Vrije Universiteit Amsterdam

Bachelor Thesis

A Graph Library for Lean 4

Author: Peter Kementzey (2634625)

1st supervisor: Jasmin Blanchette
daily supervisor: Jannis Limperg
2nd reader: Anne Baanen

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

July 5, 2021

Abstract

Lean 4 is a new efficient functional programming language. I show the main design decisions in implementing a graph library for Lean 4. I go into detail about graph representations and implementation of algorithms including depth- and breadth-first traversal, topological sorting and the push-relabel algorithm for the maximum flow problem. Finally, I conclude with a benchmark against a comparable Haskell library and show that Lean 4 is significantly more efficient.

I would like to thank my daily supervisor Jannis Limperg for all his help in making the important design decisions, pointing me to good references and giving extensive feedback on the thesis. I could not have imagined a better person to work with.

I would also like to thank Jasmin Blanchette, my main supervisor, for his help in writing the thesis and preparing for the presentation.

Contents

1 Introduction

Lean is an interactive theorem prover and a functional programming language [1]. The upcoming fourth version will be released soon. In this thesis I present a graph library built for Lean 4 and benchmark it against another graph library implemented in the popular functional programming language Haskell. The motivation for this work is twofold.

First, since the new version breaks backwards compatibility the goal is to create a graph library for the new version. Having a graph library for a programming language is very helpful as graphs have many common uses in programming projects, including representing social networks, computer networks, relations and processes in various fields. It is therefore imperative to have quick and efficient methods to construct and compute properties of graphs.

Secondly, the goal is to verify the claim by the developers of Lean 4 that it is a very efficient functional programming language [2]. To that end the work contains benchmarks against a Haskell graph library.

First, in Section 2 I introduce some basic notations and graph theory. I explain the concepts necessary to understand the rest of the thesis. Then I introduce Lean 4 to place it into context. I explain the difficulties of working in an unreleased programming language. In Section 3 I discuss the choices of graph representations in computer memory. Additionally, I describe the implementation of the graph representation. Then in Section 4 I go into details about the implemented algorithms. I extensively explain traversals of graphs, I show a shortest-path search, minimum spanning tree, topological sorting and an algorithm for the maximum-flow problem. In Section 5 I finish by showing the running times of topological sorting tasks of the implemented library are much faster than `Data.Graph` in Haskell.

The code to the version of the library used at the time of writing is publicly available on GitHub.[1] It contains instructions for using the library.

2 Background

I introduce the basic notion of graph theory and notational conventions used throughout the thesis. Furthermore, I go into some detail on Lean 4.

2.1 Graphs

A graph G consists of a set V of vertices or nodes connected by a set of edges E. An edge (u, v) may be an undirected edge in which case it is equivalent to (v, u) or directed, then (u, v) is an edge from u to v. Undirected graphs may only have undirected edges, while directed graphs may only have directed edges. In the library edges always have labels that I call weights, as that is what they generally represent. A simple graph has only one edge (u, v) for any pair of vertices $u, v \in V$, for undirected graphs this means only one edge (u, v) or (v, u), while for directed graphs edges (u, v) and (v, u) without other duplicates are allowed in a simple graph. If a graph has more than one edge (u, v) it is a multigraph. $|V|$ is the size of the set of vertices called the set's cardinality. The

[1] `https://github.com/PeterKementzey/graph-library-for-lean4/tree/e49c9fa51edfb61b9c81a014fabc2069b8db88ba`

order of a graph $G = (V, E)$ is $|V|$. $|E|$ is the cardinality of the set of edges. A path P is a sequence of edges, where each edge starts in the vertex the previous edge ended in, therefore (u, v) can be followed by (v, x), $x \in V$ such that each vertex is in the path at most once. The length of a path is the sum of the weights of edges in it, assuming that summation is defined on the edge weights. The shortest path from u to v is a path from u to v with minimal length, assuming that the edge weight has a decidable less-than relation defined. The distance of vertex u to vertex v is the length of a shortest path from u to v, in undirected graphs this equals the distance from v to u. Node v is reachable from node u if there is a path from v to u. A graph is connected if all nodes are reachable from every other node. The eccentricity of a vertex is the largest distance to any edge in the graph, if the graph is disconnected it is defined as infinite. A cycle is a non-empty path from u to u, lifting the restriction of not having duplicate nodes only for the last node. A graph is cyclic if it contains cycles. A tree is an acyclic connected graph. A forest is a set of trees. A minimum spanning tree is a tree spanning a connected undirected graph whose sum of edges is minimal. A minimum spanning forest is a minimum spanning tree for each connected component of an undirected graph.

2.2 Lean 4

Lean is an open-source theorem prover originally developed at Microsoft Research and Carnegie Mellon University [1]. Lean 4 extends the theorem prover to an an efficient dependently-typed general purpose pure functional programming language in the vein of Coq. It is currently being released as milestones towards a first stable release of version 4.0.0 [3]. At the time of writing the latest milestone[2] was 4.0.0-m2 released on March 2, 2021. Additionally, a nightly build[3] is available containing the latest developments. Currently it is recommended to use the nightly release as it works with the VS Code extension and contains many features and bug fixes implemented since milestone 2.

Lean 4 uses reference counting optimized for a purely functional language with minimized reference count increments and decrements and other optimizations described by Ullrich and de Moura, the main developers of this version, in [2]. In their paper they claim that Lean 4 outperforms Haskell compiled with GHC 8.8.3 in a range of benchmarks. Additionally, they claim that in most of these benchmarks Lean 4 outperforms OCaml, Swift and Standard ML. In [2] these performance advantages are attributed to the use of the new approach to reference counting instead of a more conventional tracing garbage collector.

The main challenges of this project were the consequences of working in a pre-released programming language. While Lean 4 already has some documentation,[4] it is currently very limited in scope, only describing the most important syntax features. The standard library completely lacks documentation and provides only basic functionality. Towards the end of the project I got more familiar with the source code of the standard library and was able to implement the functionality I needed. An example of missing functionality is the disjoint-set data structure (union-find data structure), needed for Kruskal's minimum spanning tree algorithm [4]. Neither is the functionality to merge sets available so I

[2] `https://github.com/leanprover/lean4/releases/tag/v4.0.0-m2`
[3] `https://github.com/leanprover/lean4-nightly/releases`
[4] `https://leanprover.github.io/lean4/doc/`

implemented it.

Additional difficulties were caused by bugs in the compiler. During this project I discovered two bugs in Lean. In the first case the code was not compiling due to some complicated interplay of namespaces and the dot notation used to call functions from the namespace of a structure. In the second case there was an out of bounds error appearing at runtime caused by the mere presence of some closed terms in the codebase [5].

3 Implementation of the Data Structure

Familiarity with the internal structure and the graph representation is necessary to understand the design decisions during the development of the library. Not only that, but the graph representation is the first and most important design decision when creating a graph library.

3.1 Graph Representation

There are two standard graph representations in computer memory, a collection of adjacency lists and an adjacency matrix [6, p. 589]. Both can be used to represent directed and undirected graphs, but adjacency matrices cannot represent multigraphs without some workarounds.

In the adjacency matrix representation a $|V|$ by $|V|$ matrix is used to represent the weight (or for a graph with no weights the presence/absence) of an edge from vertex v to u at index (v, u). A special value is used to represent no edge. If the graph is undirected then the value at (v, u) and (u, v) is always the same. The advantage of this representation is that we can determine in $O(1)$ time if a pair of vertices is connected by an edge. Determining the set of outgoing neighbors of any vertex takes $O(|V|)$ time, linear in the number of vertices. A major drawback is that the adjacency matrix is not a space efficient representation in case of sparse graphs, where $|E|$ is significantly smaller than $|V|^2$ [6, p. 589].

The alternative graph representation is to keep adjacency lists in an array of size $|V|$, one list for each vertex. An adjacency list contains the outgoing edges of the respective vertex, consisting of target vertex and weight. In this representation the set of outgoing neighbors of a vertex can be found in $O(1)$ time for simple graphs; for multigraphs it takes $O(|\{x \mid (v, x) \in E\}|)$. We can determine if there is an edge from v to u in $O(|\{x \mid (v, x) \in E\}|)$, linear in the number of outgoing edges of v.

While both representations have some advantages and disadvantages, the adjacency matrix representation is only suitable for the specific case of a dense graph (where $|E|$ is close to $|V|^2$); for a general purpose graph library the adjacency lists representation is better suited. This is not only because of the memory overhead, but also because many of the useful graph algorithms are based on traversals, therefore being able to obtain the outgoing neighbor set of a vertex quickly is more important than seeing if there is an edge from a specific source vertex to a specific target vertex. Additionally, while the adjacency matrix enables to determine if there is an edge $(u, v) \in E$ in constant time, in adjacency lists the constants of the linear asymptotic bound are usually low. Overall, therefore, a collection of adjacency lists is better suited for this library.

3.2 Graph Implementation and Basic Functionality

```
structure Edge (β : Type) where
  target : Nat
  weight : β

structure Vertex (α : Type) (β : Type) where
  payload : α
  adjacencyList : Array (Edge β)

structure Graph (α : Type) (β : Type) where
  vertices : Array (Vertex α β)

def addVertex (g : Graph α β) (payload : α) : (Graph α β) × Nat
```

Listing 1: Graph implementation

Deciding to use a collection of adjacency lists is only the first step. The specifics of this representation are important as well. We must consider what container to store the adjacency lists in; how and what data the user can store in graphs; and how the user can refer to the nodes. These questions are related; since the library has to be able to retrieve the nodes from the container using the reference of the user. Ideally the user can assign any payload of arbitrary type to the vertices. A potential feature is the ability to refer to the nodes by payload. This is convenient for some applications, but requires unique payloads.

There are two approaches to this problem. One option is to use an array, in which case we need to use the `Nat` type to refer to vertices. Another option is to use a hash map to map any hashable type to the adjacency list of its respective vertex. Both of these allow us to access the adjacency list of a specific vertex in $O(1)$ time, but an array has much lower constant factors and a more compact memory representation. In order to make it possible to refer to nodes in an array by payload we would need to limit the user to natural numbers as vertex payloads for vertices, which is unreasonable. However, requiring that the payload is hashable and each payload is unique is also suboptimal. Therefore I introduce vertex IDs, which are `Nat`s referring to an index in an `Array` containing the adjacency lists and the payload of arbitrary type. This way there are no limitations on the vertex payload and we achieve the fastest possible access time to the adjacency lists. A downside of this choice is that the user needs to keep track of the vertex ID returned by the function that adds vertices.

Listing 1 shows the graph representation we arrived at and the type signature of the function to add vertices. A `Graph` consist of an array of vertices, which in turn contain a payload and an `Array` of `Edge` types. `Edge`s contain a `Nat` representing the target vertex of the edge and a weight of arbitrary type. The `addVertex` function returns a new graph and a vertex ID.

The removal of edges and vertices poses some questions. In this implementation, there is no way to distinguish two identical edges $(v,\ u)$ that have the same weight. It is therefore not obvious what interface to provide for removing edges and whether that interface should

```
def removeAllEdgesFromTo (g : Graph α β) (source : Nat) (target : Nat)
  (weight : Option β := none) : Graph α β

def removeVertex (g : Graph α β) (id : Nat) : (Graph α β) × (Nat -> Nat)
```

Listing 2: Removal functions

remove one such edge or all. For most applications it makes more sense to remove all of the edges from vertex v to vertex u, as generally the goal of removing edges is to disconnect the two vertices. Therefore that is what the function does, with an optional parameter to only remove edges with a specific weight.

As discussed before, vertex IDs are essentially indices into the array of vertices. This makes it cumbersome to remove vertices as the IDs would change. A possible solution to this is to replace the vertex with a placeholder value, but this would make the graph representation bloated when the graph is mutated a lot. This would not only create something resembling a memory leak, but it would also require special cases for these placeholders everywhere in the library. To avoid doing this the library simply shrinks the array, while returning a mapping alongside the new graph from old vertex IDs to new vertex IDs. The function is accompanied by a warning in the documentation for the user to be careful with removing vertices from the graph, because this will change the vertex IDs.

4 Implementation of the Algorithms

The performance, usability and maintainability of the library are all important factors when making design decisions in the implementation of the algorithms. Many of the algorithms traditionally developed assume an imperative language with side effects and are hard to implement effectively in a functional programming language. This is where the domain specific language (DSL) notation embedded into Lean comes into play. In many of the algorithms below I have used it to improve the readability of the code. The DSL notation is implemented in Lean, it can basically be seen as very extensive syntax sugar, as it gets translated into core Lean 4 code.

4.1 Depth- and Breadth-first Traversal

Some sort of traversal is the core of many graph algorithms and while a lot of them require unique, specialized traversals, a significant proportion is done in a breadth-first or depth-first order. Therefore, it is important that the interface provides flexibility in how the traversal can be used to allow other algorithms to be built on top of the existing traversal.

A breadth-first traversal is a traversal in which the next node is always a successor of the least recently visited node such that no nodes are revisited. This leads to a traversal where the graph is mapped out evenly in all directions. Since the number of nodes necessary to be kept in memory grows exponentially with the branching factor, breadth-first traversal needs extensive amounts of memory.

A depth-first traversal is a traversal where the strategy to choose the next node to

be visited is to choose an unvisited successor of the most recently visited node that has any unvisited successors. This leads to the traversal visiting nodes following a path until reaching a leaf node, after which the traversal is continued at the closest predecessor that has a branching point with an unvisited node. This is a much more memory-efficient way of inspecting the graph than breadth-first search is, as only the single path currently being explored has to be kept in memory, an upper bound therefore is the longest path in the graph. In the case of depth-first search, the notion of leaving a node makes sense for certain algorithms. A node is considered to be left when all of its successors, including the ones that are indirect successors, have been visited.

There is a natural way of implementing these two traversals with a shared algorithm, with the only difference being the container used for storing and retrieving the following node to visit. The idea is that each time a node is visited all of its successors are added to the container, then the next node that is removed from the container is the one that should be visited next. To achieve a depth-first order we need to use a stack, while to achieve a breadth-fist order we must use a queue as a container. This aligns with what I explained before; that breadth-first search uses the least recently visited node, while depth-first search uses the most recently visited node to continue the traversal. This makes a very elegant solution to the problem. The container becoming empty means that all reachable nodes from the source node have been visited. The notion of leaving a node in a depth-first traversal can be implemented in this algorithm by adding the node back to the stack when visiting it before adding its successors. This time the node will have to have a special flag signifying the next time it is read from the container that the node is being left rather than visited, which in the case of a stack will happen after all of the successors and their successors are removed recursively, yielding the desired order of leaving and visiting nodes. Using the notion of leaving a node has no meaningful interpretation in a breadth-first search.

```
universes u v
structure Container (β : Type u) (γ : Type v) where
  container : γ
  addFun : β -> γ -> γ
  addAllFun : Array β -> γ -> γ
  removeFun : γ -> Option (β × γ)
-- Standard library
def dequeue? (q : Queue α) : Option (α × Queue α)
def pop (s : Stack α) : Stack α
def peek? (s : Stack α) : Option α
-- Not standard library
def pop? (s : Stack α) : Option (α × (Stack α))
```

Listing 3: Container and relevant `Stack` and `Queue` functions

The main challenge in implementing this was the container. Fortunately, the standard library of Lean 4 already has a `Stack` and a `Queue` implemented, however, these have a slightly different interface. The solution was to create a `Container` wrapper around the two containers and extend the interface of the `Stack` of the standard library to be

compatible with that of the `Queue`. While the `Queue` provides a function `dequeue?` which returns a tuple of an element removed from the `Queue` and the `Queue` without the element in it, the `Stack` provides separate functions `peek?` and `pop` that do the same things, respectively. These functions had to be combined into a `pop?` function as seen in Listing 3.

While this algorithm is a very elegant one and shares code between the two different implementations, during the benchmarking it turned out that it has one major drawback, that it is very inefficient. As explained above, when visiting a node, all of its successors will be added to the container, which then ensures a first-in first-out or last-in first-out order. In Figure 1 when a traversal is started at node `0` then `1` and `2` are added to the container. Then, without loss of generality we can assume `1` is removed from the container next. Since node `2` has not yet been visited we cannot know that it should not be added to the container anymore, therefore we add it another time. This pattern causes node `2` to be duplicated in the container, it can occur both in depth- and breadth-first traversals. The reason for the inefficiency of this algorithm is that in large complex graphs many of the nodes will have been added to the container many times before being visited, but once they are visited it is not trivial to filter them out from the container anymore, they simply have to be ignored when removed from the container again. Note that this does not make the logic wrong; we just have to check for each removed node if it has already been visited. Specialized algorithms for breadth- and depth-first traversals can make assumptions that the general algorithm cannot, helping them to avoid this duplication issue.

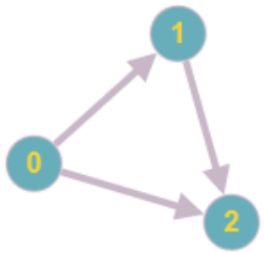

Figure 1: Example traversal

First I explain the specialized implementation of depth-first search. After that I show breadth-first search and point out some of the unique assumptions not valid in the shared code. A depth-first traversal can often have multiple source nodes, even all of the nodes in the graph if the user wants to visit all nodes at least once and always follow a depth-first order from those starting nodes. Therefore, the basis of the depth-first traversal is a loop through the `sources` parameter, as seen in Listing 4. At each source node, the function is recursively called using the targets of the adjacency list of the node as the `sources` argument. Before visiting (calling the user's `visit` function on) the current source node we need to check its visited flag for two reasons. Firstly, since the sources array in the recursive calls is effectively the adjacency list of a node in the case of a multigraph there

```
private def depthFirstTraverseAux (g : Graph α β)
(visit : Nat -> γ -> γ × Bool) (leave : (Nat -> γ -> γ ))
(state : γ) (sources : Array Nat)
(visited : Array Bool) : Nat -> γ × Bool × Array Bool
  | 0 => (state, true, #[])
  | n + 1 => do
    let mut visited := visited
    let mut state := state
    for id in sources do
      if visited[id] then continue else
      visited := visited.set! id true
      let (newState, terminate?) := visit id state;
      state := newState
      if terminate? then return (leave id state, true, #[]) else
      let adjacencyList := (g.vertices[id].adjacencyList.map
        (λ edge => edge.target)).filter (!visited[.])
      let (newState, terminate?, newVisited) :=
        g.depthFirstTraverseAux visit leave state adjacencyList visited n
      visited := newVisited
      state := leave id newState
      if terminate? then return (state, true, #[])
    return (state, false, visited)

def depthFirstTraverse (g : Graph α β) (sources : Array Nat)
(startingState : γ ) (visit : Nat -> γ -> γ × Bool)
(leave : Nat -> γ -> γ  := (λ _ x => x)) : γ :=
  (g.depthFirstTraverseAux visit leave startingState
   sources (mkArray g.vertexCount false) (g.vertexCount)).1
```

Listing 4: Depth-first traversal

may be duplicates. Secondly, even if we were to ensure that there are no duplicates in the sources array, each node in the `sources` array will already have been visited if it is reachable from a source node preceding it in the `sources` array. And since this check is already present it is generally not worth removing the duplicates from the adjacency lists before the recursive call. The library is basically making the assumption that in most graphs that users will be traversing there will be at most a few pairs of vertices v and u that contain multiple edges from v to u and so it would not be more efficient to construct a hash set and then revert it to an array in order to remove these duplicates. This has the added benefit of following the order in which the user added the edges in when traversing the adjacency lists which may be important in certain cases. After the recursive call exploring the current node's adjacency list is done, we can `leave` the node and go on to the next one. This algorithm works excellently with the adjacency list representation as it can basically loops through these arrays in place, rather than constructing a `Stack` of nodes. Additionally, as explained above, in the shared-code implementation we can only

filter out those nodes from the adjacency list that have already been visited before the node is visited, while in this case each node of the adjacency list is explored first and if the following nodes in the `sources` array have already been visited through this exploration they are simply ignored.

```
private def breadthFirstTraverseAux (g : Graph α β)
(visit : Nat -> γ -> γ × Bool) (state : γ) (startingSources : Array Nat)
(sources : Array Nat) (visited : Array Bool) : Nat -> γ
  | 0 => state
  | n + 1 => do
    let mut visited := visited
    let mut state := state
    let mut nextSources : Std.HashSet Nat := Std.HashSet.empty
    for id in sources do
      visited := visited.set! id true
      let (newState, terminate?) := visit id state;
      state := newState
      if terminate? then return state else
      let adjacencyList :=
        (g.vertices[id].adjacencyList.map (λ edge => edge.target))
      for targetId in adjacencyList do nextSources :=
        nextSources.insert targetId

    let sourcesArray : Array Nat := nextSources.fold
      (λ arr id => if visited[id] then arr else arr.push id) #[]
    let startingSources := startingSources.filter (!visited[.])
    match (sourcesArray.isEmpty, startingSources.isEmpty) with
      | (false, _) => g.breadthFirstTraverseAux visit state
        startingSources sourcesArray visited n
      | (_, false) => g.breadthFirstTraverseAux visit state
        startingSources.pop #[startingSources.back] visited n
      | (true, true) => state
```

Listing 5: Breadth-first traversal

In a breadth-first traversal, the notion of the `sources` array has a different interpretation. At first, it may only contain a single source node, in each following iteration, it will contain all unvisited nodes from all of the adjacency lists of the nodes in the `sources` array in the previous iteration. The basis of this traversal is also a loop through this array but in this case rather than making recursive calls inside the loop the `sources` for the next iteration have to be compiled from all the adjacency lists as seen in Listing 5. Since this list will be the combination of the adjacency lists of many nodes, I assumed it is likely that it will contain many nodes multiple times. Therefore, rather than using an array for this compilation, the algorithm uses a `HashSet` from the standard library. Contrary to depth-first search in this step it is not worth filtering the already visited nodes from the adjacency list before adding them to the set as this operation would be repeated multiple

times for the same node because many of the nodes will probably be in the adjacency list of multiple nodes. Instead, once all sources have been visited in the loop a `fold` is used on the set which turns it into an array and filters the already visited nodes at the same time, only needing to check the visited flag for each node a single time. This guarantees that the `sources` array of the next iteration does not contain duplicates or already visited nodes, and we do not visit any other nodes while visiting these `sources`, so there is no need to check at the beginning of the loop if the visited flag is set. In the first iteration, there is only one node so both these assumptions hold. In the shared code, however, the same assumptions could not be made. Generally, a breadth-first traversal is started from a single node, however, I have provided the possibility to start from multiple sources as it may still be useful in some cases. In the case of breadth-first search, we cannot use the initial array of starting sources as the first `sources` array as this would start a breadth-first search in all of these nodes simultaneously, not one by one. In each iteration, if the `sources` array is empty we must remove a node from the starting sources and check if it has already been visited, to make sure that the assumption I mentioned earlier is maintained. If it is unvisited, then a breadth-first traversal can be started from the node. The traversal will be completed once all starting nodes have been visited.

Both traversals provide a similar interface. They take a `visit` function parameter and a starting state, these work similarly as a fold function on an array. `visit` takes a vertex ID and the current state and returns a new state and a Boolean flag meant to signal that the traversal has reached its goal and can terminate, useful for example when trying to determine the reachability of a specific target vertex. Depth-first traversal additionally provides a `leave` function with the same interface except there is no termination flag. In a breadth-first search when the termination flag is set by the `visit` function the traversal can be immediately terminated. In depth-first search, however, the nodes that have not yet been left must still be left, but no further nodes should be visited. Therefore, in depth-first search, the termination flag must be propagated up through the recursion stack using the return value.

Overall, we can see that there are plenty of differences and assumptions in the specialized algorithms that make them much more efficient than the shared code. Especially the exploration first approach in depth-first traversal and the hash set approach in breadth-first traversal that help reduce the repeated visiting of nodes reduce the running times significantly. Nevertheless, both the container based and the specialized versions of the traversal implementation stayed in the library. I used the two to verify that the results of a depth-first traversal using the shared and the separated code match. In a breadth-first traversal the order may be somewhat different due to the use of the hash set in the efficient version, so the queue-based version can be used if it is important to the user that the order of nodes visited is chosen based on the order of edges added. The flexibility of the `visit` and `leave` functions makes it possible to use these traversals for simple tasks such as determining the reachability of a node or finding a path but also for somewhat more complex algorithms, like calculating a topological sorting while doing cycle detection.

4.2 Shortest Path Tree using Dijkstra's Algorithm

Finding shortest paths is one of the most common problems in graph theory. I have implemented the classic algorithm conceived by E. W. Dijkstra [7]. This algorithm solves

the single-source shortest-paths problem. If we do not specify a target node the algorithm will compute a shortest-path tree, a tree rooted at the source vertex which spans across the reachable part of graph G from the source vertex such that it contains the shortest path from the source vertex to any other reachable vertex in G. Alternatively, we may specify a target vertex, then the algorithm can terminate once that vertex has been found and only returns a shortest path from source to target. Dijkstra's algorithm only works on graph with no negative weights [6, p. 658]. This algorithm chooses the next nodes to visit in a very specific order, so unfortunately it cannot be implemented using the breadth-first traversal from Section 4.1.

The main design decisions in implementing this algorithm were about the representation of the shortest-path tree and shortest paths in general, and by extension, the combination of the specific target code with the general code. First, I will show the implementation of the `Path` type, then I will explain the `ShortestPathTree` type. In the end I will detail the other design decisions in the implementation.

```
inductive Path (β : Type) : Bool → Type where
  | vertex : Nat -> Path β false -> Path β true
  | edge : β -> Path β true -> Path β false
  | empty : ∀ {b}, Path β b
```

Listing 6: inductive path structure

Paths in this library are represented by an inductive type with three constructors, one for edges, one for vertices, and one for the empty path and end of path as seen in Listing 6. A Boolean value is used to enforce that vertices and edges can only be added to the path alternatingly. This inductive construction gives a nice way for the user to pattern match on the path to traverse it as seen in Listing 7

```
match p with
  | vertex id restOfThePath => match restOfThePath with
    | edge weight restOfThePath => -- code
    | empty => -- code
  | empty => -- code
```

Listing 7: Pattern matching on Path β true

Shortest-path trees are somewhat more complicated and contain various useful information. To make this user-friendly, the `dijkstra` function returns a wrapper structure with some interface functions to get various information out of it. When computing Dijkstra's algorithm the resulting format for each vertex provides its predecessor in the shortest path leading to it and its distance from the source. This is essentially the information needed for this to be a representation of a shortest-path tree, however, to be able to construct paths from this efficiently it helps to know the edge weight to the predecessor of the nodes as well, so this information is also added when updating tentative distances in the algorithm. The `structure DijkstraVertex` contains these pieces information, a `ShortestPathTree` is an array of Dijkstra vertices. Theoretically this representation could be used by the

```
structure DijkstraVertex where
  predecessor : Nat
  distance : Option Nat := none
  edgeWeightToPredecessor : Nat := 0

structure ShortestPathTree where
  dijkstraVertices : Array DijkstraVertex

def distanceToVertex (t : ShortestPathTree) (id : Nat) : Option Nat

def eccentricity (t : ShortestPathTree) : Option Nat

def successorsOfVertexDeprecated (t : ShortestPathTree)
(id : Nat) : Array Nat

def predecessorOfVertex (t : ShortestPathTree) (id : Nat) : Option Nat

def pathToVertex (t : ShortestPathTree) (id : Nat) : Option (Path Nat true)
```

Listing 8: Shortest path tree structure and interface functions

user directly, but there are some functions provided to hide this abstraction. In this representation the distance to a node from the source node and the predecessor of a node in the shortest path to it can be determined in constant time. Constructing the shortest path to a vertex can be done in linear time in the length of the path. Additionally, the eccentricity of the source vertex can be determined in $O(|V|)$ time. A function which determines the successors of a vertex in the shortest path tree is also provided, however, there is a warning for the user not to use it to construct paths as it is not very efficient, it also takes $O(|V|)$ time while the predecessor could be found in $O(1)$ time. Listing 8 shows the type signarutre of these functions.

When the user only cares about the shortest path to a specific target value, it is wasteful to compute the entire shortest-path tree. To short-circuit this calculation only a very slight modification has to be made, essentially the same code can be used. Therefore, an option is provided in the auxiliary function which the function `dijkstraWithTarget` uses, that has a different type signature. The auxiliary function still returns a shortest-path tree, but it is not complete, just enough for the `pathToVertex` interface function of the tree structure to construct the path.

Dijkstra's algorithm is based on a while loop, which can alternatively be represented as a recursive function, generally in neither of these cases can the termination of the program easily be proven. However, the functional programming language Lean requires that terminations is always proven. There is a method to work around this issue commonly referred to as the fuel pattern described in [8]. The idea is to find an upper bound on iterations, use it as an argument `n` for the recursive function and decrease it in each iteration, as seen implemented in Listing 9. This guarantees termination as the number of recursive calls is limited. While this is not always possible to do, for Dijkstra's algorithm

it is fairly straightforward, as the number of recursive calls is limited by the cardinality the unvisited set, which in turn is limited by the order of the graph.

```
private def dijkstraAux (current : Nat) (unvisited : Std.HashSet Nat)
(dijkstraVertices : Array DijkstraVertex) : Nat -> Array DijkstraVertex
  | 0 => dijkstraVertices
  | n + 1 => _ -- Rest of the Dijkstra logic
```

Listing 9: Simplified function showing fuel pattern

The last problem in this implementation was the type of the edge weights in the graph. As mentioned earlier, one of the limitations on the edge weight imposed by the algorithm is that it only works on non-negative edge weights. Additionally, there has to be a meaningful addition and a less than operation that are compatible with each other, as these have to be used to compute the distance of a shortest path. While this would theoretically be possible to do with positive rational numbers and other types as well, this was not yet implemented and may be added in the future, for now, the algorithm only works on graphs with natural numbers as edge weights, and there is a function provided which can map all edge weights of a graph to a different type. This restriction also guarantees that there are no negative edge weights in the graph.

4.3 Minimum Spanning Tree using Kruskal's Algorithm

Kruskal's algorithm finds a minimum spanning forest, a collection of minimum spanning trees for each connected component in the graph [4]. If the graph is connected, it finds a single minimum spanning tree. This is only well-defined on undirected graphs; hence this functionality is only available in the `UndirectedGraph` namespace.

This algorithm requires the edge weights to have a less-than operation. Lean 4 has a `[LT β]` type class, however, it does not refer to a decidable relation, so instead, following the Lean 4 standard library's example I used a function `(lt : β -> β -> Bool)` argument. Additionally, since the algorithm uses sets of edges, the edge weights are limited to `Hashable` types, so they can be collected in `HashSet`s.

Undirected graphs are implemented as a simple wrapper around the directed graph we have seen in Listing 1. Most functionality redirects to the directed implementation, however, there are some differences. A minor example is the in- and out-degree of vertices in directed graphs only translates to a single degree function in undirected graph. While a minimum spanning tree is only defined for undirected graphs, a topological sorting is only defined for directed graphs.

4.4 Topological Sorting

A topological ordering of a graph is a linear ordering of its vertices such that for all directed edges (u, v) in E, u comes before v in the ordering. A topological sort can therefore only be made on directed acyclic graphs (DAG). A DAG may have one or more correct topological orderings. Algorithms exist that perform cycle detection while constructing a topological ordering. An algorithm with cycle detection will not return a result if the graph contains a cycle. Algorithms without cycle detection should therefore only be used on graphs

that are guaranteed to be acyclic, in that case the lack of cycle detection will enable a significant performance boost. Topological sorting has many uses, a common one being the representation of dependencies in a task or scheduling problem by edges and then performing a topological sorting to determine the order of sub-tasks.

Topological sorting can easily be implemented based on depth-first search following the algorithm described by Tarjan [9]. The algorithm I implemented is based on that but slightly adapted to be compatible with the traversal interface of this library. First, I describe the base algorithm, then the implementation without cycle detection, then I explain how cycle detection is added to this.

The algorithm uses a notion of permanent mark and temporary mark defined for all vertices in the graph, initially none of them are marked. The main body of the algorithm is based on a while loop which selects a node without a permanent mark while there are any. The `visit` subroutine is called on this node and the loop is repeated. In the `visit` subroutine we first check if the node has a permanent mark, if it does then we return. We also check if it has a temporary mark, if it does then the graph contains a cycle, therefore we return an error or `none`. Then we mark the node with a temporary mark and visit each of its successors recursively (thereby doing a depth-first exploration). After the recursive calls have been done, we remove the temporary mark and add a permanent mark to the node. In the end we insert the node to the beginning of the resulting ordering.

We can see that the permanent mark is used to do a depth-first traversal without revisiting nodes, while the temporary mark is used to detect cycles. Each node that is on the path the traversal consists of at the moment is marked with a temporary mark, so if one of these nodes is visited again there is a cycle in the graph. As the permanent mark is only used to keep track of the depth-first traversal we do not need it, since we use an already implemented depth-first traversal.

The visit procedure in this algorithm aligns well with the `visit` interface of depth-first traversal explained in Section 4.1. However, since this visit subroutine also executes the dept-first traversal we must split the steps into `visit` and `leave` functions to leave handling of the traversal to the already implemented code. `visit` should contain everything before visiting the successors while `leave` should contain everything after that.

```
def topSortUnsafe (g : Graph α β) : Array Nat :=
  let res := g.depthFirstTraverse g.getAllVertexIDs
             initialState (visit g) leave
  res.reverse
  where
    initialState := #[]
    visit (g : Graph α β) (id : Nat) (s : Array Nat) : Array Nat × Bool :=
      (s, false)
    leave (id : Nat) (s : Array Nat) : Array Nat := s.push id
```

Listing 10: Topological sorting without cycle detection

For a topological sorting without cycle detection all we need is the `leave` function to construct the result. While Lean provides the `List` type, `Array` which is implemented through `std::vector<T>` in C++ performs much better [10], so I use an `Array Nat`

instead. For arrays, however, prepending is not as efficient as appending. Therefore the nodes are appended in `leave` and the result is reversed in the end. `visit` in this case is just the identity function which also returns the Boolean flag not to terminate the traversal. Listing 10 shows the implementation.

```
private structure State where
  temporaryMark : Array Bool
  res : Array Nat
visit (g : Graph α β) (id : Nat) (s : Option State) : Option State × Bool :=
  let state := s.get!
  if g.vertices[id].adjacencyList.any
    (λ edge => state.temporaryMark[edge.target])
      then return (none, true) else
  let updatedState := { state with
    temporaryMark := state.temporaryMark.set! id true
  }
  (some updatedState, false)
leave (id : Nat) (s : Option State) : Option State :=
  match s with
    | some state =>
      some { state with
        temporaryMark := state.temporaryMark.set! id false
        res := state.res.push id
      }
    | none => none
```

Listing 11: State structure and visit and leave functions for cycle detection

To perform cycle detection, we must use the temporary mark, but we also need to keep the result array. I create a simple structure for the traversal state containing these two as seen in Listing 11. `visit` marks the nodes with the temporary mark, leave removes the mark. Since in this implementation already visited nodes will not be visited again, even if they do not currently have the temporary mark, we must inspect the successors of the visited node to see if they have a temporary mark. If they do, we may terminate the traversal. To account for this possibility, I use an `Option State`, returning `none` if the graph contains a cycle.

4.5 Push-Relabel Method for Maximum-Flow Problem

A flow-network is a directed graph where each edge weight represents a capacity with a source and a target or sink vertex. A flow in a flow-network is an assignment of flow values to each edge, a value between 0 and the capacity of the edge, such that at each vertex except the source and the sink, the incoming flow is equal to the outgoing flow. At the source there is no incoming flow, while at the sink there is no outgoing flow. We can think of this flow for example as traffic on a map, water in a pipe network or electrical current on wires. The overall flow rate in the network is the overall outgoing flow from the source or the overall incoming flow to the sink, these values are equal in any valid flow [6, p.

710]. The maximum flow problem is the task of finding a valid flow in the network whose overall flow rate is maximal.

This is a non-trivial problem, and many algorithms exist to solve it, so the main design decision was the choice of algorithm. I considered the options described in [6]. The first algorithm ever published for this problem was the Ford-Fulkerson method [11], a family of algorithms with varying ways of finding a so-called augmenting path [6, p. 725]. The naïve approach is to choose these paths arbitrarily. This is usually referred to as the Ford-Fulkerson algorithm and runs in $O(|E|f*)$ where $f*$ is the maximum flow of the flow-network. The Edmonds-Karp algorithm [12] is another implementation of the Ford-Fulkerson method which chooses augmenting paths using breadth-first search, this improves the complexity to $O(|V||E|^2)$ [6, p. 722]. While these first algorithms have the benefit of simplicity, a variety of improved algorithms have been discovered since then. A notable family of algorithms is the push-relabel method, a naïve implementation of which is Goldberg's generic maximum-flow algorithm, which improves the time complexity to $O(|V^2||E|)$ [13]. The push-relabel family of algorithms can also be improved by changing the operation and vertex selection rules. A more advanced algorithm is the push-relabel algorithm with the relabel-to-front selection method also described in [13] that runs in $O(V^3)$. The fastest currently known algorithm is due to Orlin and runs in $O(|V||E|)$ [14].

Unfortunately, the algorithm described by Orlin is very complex. It uses advanced heuristics and other optimizations and is therefore out of scope for this project, as it would have taken too much time to implement. I chose the push-relabel algorithm with the relabel-to-front selection method as a good balance between complexity and efficiency. This algorithm could be broken down into 20 subfunctions to make the problem manageable. Explaining the specifics of the algorithm is out of scope for this thesis.

To further simplify the implementation I restrict the problem in three ways, without loosing too much in generality. First, capacities in maximum flow problems are often integral in practice [6, p. 725]. Additionally, according to the integral flow theorem, if all capacities are integral, then the resulting maximum flow is also integral. Limiting the algorithm to only work on graphs with edge weight types of `Nat` allows the implementation to be considerably simpler. Second, any flow-network that has multiple edges between the same pair of vertices, either in the same direction (parallel edges) or in opposite directions (anti-parallel edges) can be converted into an equivalent flow-network without such edges [6, p. 711]. In case of parallel edges, the capacities can simply be added together, while in case of anti-parallel edges an additional vertex can be added to the graph, through which the edge in one of the two directions can be redirected. Disallowing parallel and anti-parallel edges therefore also does not limit the functionality of the algorithm, however, allows for further simplifications. Finally, flow-networks with multiple sources or sinks can easily be converted into an equivalent flow-network with a single source and sink [6, p. 713]. Therefore these problems are usually considered for a single source and sink. To allow multiple sources we may add an additional source vertex and connect it to each source with and edge of infinite capacity, likewise, to augment multiple sinks we may add an additional sink vertex and connect every augmented sink to the added sink with edges of infinite capacity. These transformations can be done automatically. Due to time constraints they have not yet been implemented in the library. Future work can be done on this.

5 Benchmarks

To evaluate the performance of both Lean 4 and the library implementation, I benchmarked the library against a Haskell library. I give a short introduction and explanation about the chosen Haskell library and benchmarks. Then I explain the limitations of these measurements. Finally, I present the results and give a short evaluation. The code used to measure these running times is available on GitHub and can be reproduced simply by running the script.[5] The tests were run on a Windows 10 system running Ubuntu 20.04.2 LTS in WSL 2, using Lean version 4.0.0-nightly-2021-07-01 and the GHC compiler version 8.10.4 in a laptop with an Intel Core i7-1065G7 processor and 16 GB of RAM.

`Data.Graph`[6] is a graph library implemented in Haskell focused exclusively on depth-first search algorithms [15]. It uses an adjacency lists representation near identical to this Lean library and is therefore the best suited graph library in Haskell for comparison. This minimizes the differences in implementation details and allows us to focus on the difference in the language used. Additionally, according to [16] it is the fastest Haskell graph library in most applications, including topological sorting and reachable nodes; both of these are depth-first search-based algorithms. The library is limited to specific applications, it does not support mutation of the graph in any form, not even creating copies. It only allows integer vertex payloads. Topological sorting and reachable nodes are overlapping functionality between the two libraries, so I used those for comparison.

These results have some limitations due to the lack of a benchmarking suite in Lean. The standard library provides a function to get the current time in milliseconds, but nothing more is provided. Haskell, on the other hand, has the Criterion package,[7] I only used the `getTime` method to keep the results comparable.

Topological sorting can only be done on directed acyclic graphs (DAG). For these benchmarks I created a simple DAG generator. It decides on a possible topological ordering of the given number of nodes at random, and then connects random pairs of vertices in a direction that keeps the topological ordering valid. The results contain graphs from six different categories:

1. 2 000 nodes and 1 000 000 edges
2. 2 000 nodes and 4 000 000 edges
3. 20 000 nodes and 1 000 000 edges
4. 20 000 nodes and 4 000 000 edges
5. 20 000 nodes and 7 000 000 edges
6. 30 000 nodes and 7 000 000 edges

Each graph category has five graphs and each graph is sorted five times. The results are the mean of the five iterations over the five graphs. Lean unsafe is the running time of the Lean topological sorting function without cycle detection, Lean safe is with cycle

[5] `https://github.com/PeterKementzey/lean-graph-benchmarking`
[6] `https://hackage.haskell.org/package/containers-0.6.4.1/docs/Data-Graph.html`
[7] `https://hackage.haskell.org/package/criterion`

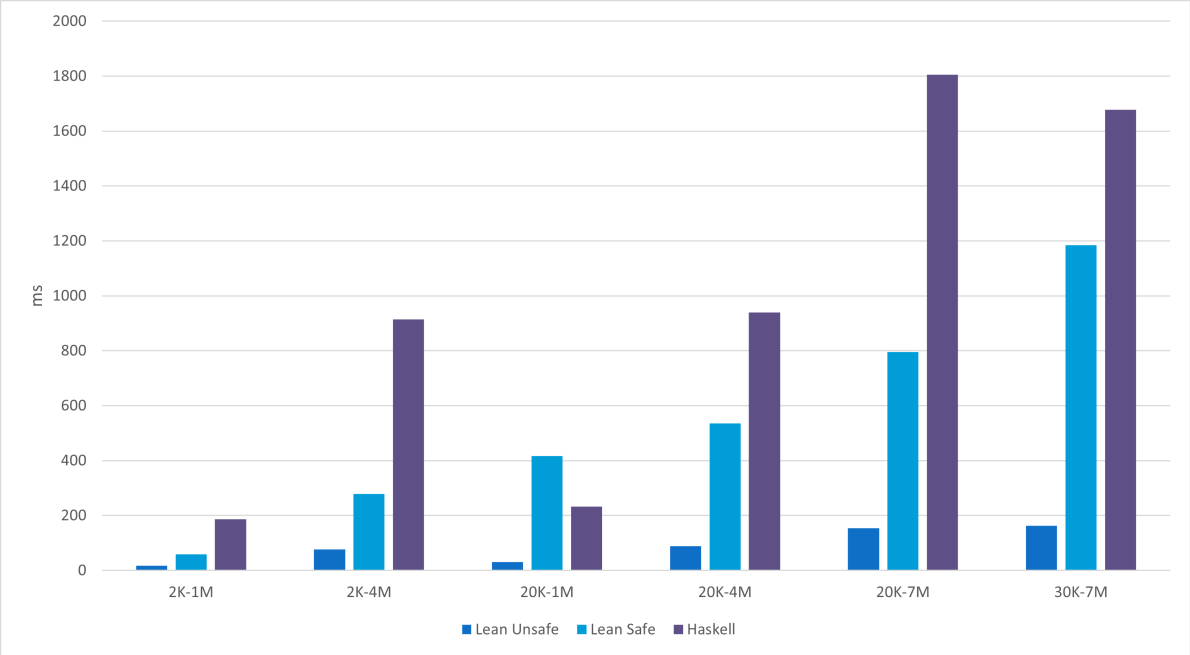

Figure 2: Topological sorting mean running times

detection. The Haskell library does not provide cycle detection. The results are shown in Figure 2.

We can see that Lean 4 outperforms Haskell significantly in every category. Additionally, even the safe Lean code is faster than Haskell in most categories. The unsafe Lean code's running time mainly depends on the number of edges in the graph; the number of nodes does not have a significant impact. In the case of Haskell, we see similar trends, but counterintuitively Haskell occasionally takes less time to sort a larger graph with the same number of edges, as seen in the last two categories. The safe Lean code's running time grows with both the number of edges and nodes increasing. This results in Haskell being faster in the category of 20 000 nodes and 1 million edges.

Both libraries provide a method that finds all reachable nodes from a specific source. To test this functionality, I have used graphs representing websites from the University of Notre Dame [17], Stanford University [18] and a combined Berkeley-Stanford webpage collection [18] [19]. Nodes represent webpages and directed edges represent hyperlinks in them. The results are the mean time it takes to find all reachable nodes from a specific source over five runs. The `Data.Graph` library is focused on depth-first search algorithms only, but Lean can provide this in a breadth-first order as well. As seen in Figure 3, Lean is clearly faster in every category and scales better on larger graphs. Remarkably, Lean using breadth-first search is also faster than Haskell which uses depth-first search.

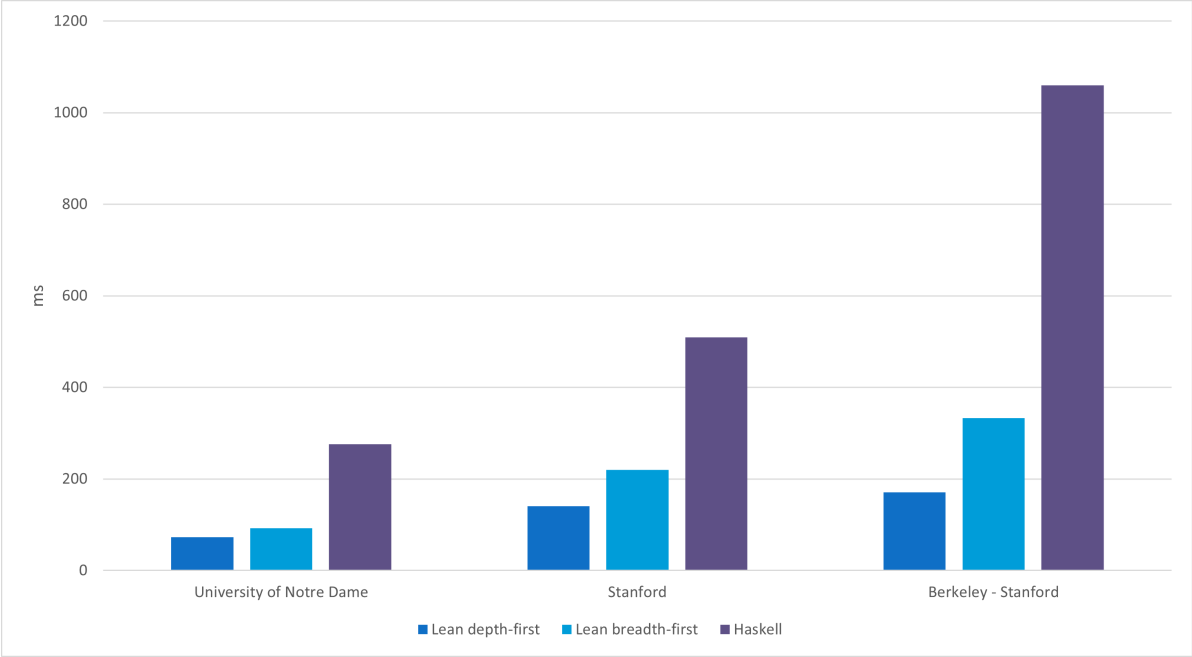

Figure 3: Reachable mean running times

6 Conclusions

Lean 4 is a theorem prover and purely functional programming language that claims to be one of the most efficient in its class, outperforming the state-of-the-art Haskell compiler. I implemented a graph library in Lean 4 and ran benchmarks comparing it to a Haskell library to independently verify the claims mentioned in [2].

The representation of a graph in computer memory is the first and most important design decision in designing a graph library. We concluded that adjacency lists stored in an array are better suited for a general-purpose graph library than adjacency matrices. One of the most basic algorithms for a graph library are the depth- and breadth-first traversal. While the classic implementation using a stack for depth-first and a queue for breadth-first traversal is very elegant, many optimizations can be done when implementing the two individually. This results in much more efficient code and has the additional benefit of making it resemble the implementation in the `Data.Graph` Haskell graph library. We implemented more algorithm implementations, including Dijkstra's shortest-path algorithm, Kruskal's minimum-spanning-tree algorithm, topological sorting based on the implemented depth-first traversal and the push-relabel algorithm for the maximum-flow problem with the relabel-to-front selection method.

To conclude, we looked at benchmarks comparing the running times of topological sorting and reachable nodes to the implementations in the `Data.Graph` Haskell library. The benchmarks showed that the Lean 4 implementation impressively outperformed Haskell, corroborating the Lean developers' claim [2].

References

[1] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The Lean Theorem Prover (system description)," in *Automated Deduction - CADE-25, 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, 2015.

[2] S. Ullrich and L. de Moura, *Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming*, 2020. arXiv: 1908.05647 [cs.PL].

[3] L. de Moura, S. Morrison, J. Avigad, S. Kong, G. Ebner, and S. Ullrich, *Readme*. [Online]. Available: https://github.com/leanprover/lean4/blob/master/README.md.

[4] J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956, ISSN: 00029939, 10886826. [Online]. Available: http://www.jstor.org/stable/2033241.

[5] P. Kementzey, *Unexpected 'error: index out of bounds' caused by the mere presence of a function · Issue #534 · leanprover/lean4*, Jun. 2021. [Online]. Available: https://github.com/leanprover/lean4/issues/534 (visited on 06/20/2021).

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd. The MIT Press, 2009, ISBN: 0262033844.

[7] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959, ISSN: 0945-3245. DOI: 10.1007/BF01386390. [Online]. Available: https://doi.org/10.1007/BF01386390.

[8] A. Bove and V. Capretta, "Modelling General Recursion in Type Theory," *Mathematical Structures in Computer Science*, vol. 15, no. 4, pp. 671–708, 2005. DOI: 10.1017/S0960129505004822.

[9] R. E. Tarjan, "Edge-Disjoint Spanning Trees and Depth-First Search," *Acta Informatica*, vol. 6, no. 2, pp. 171–185, Jun. 1976, ISSN: 1432-0525. DOI: 10.1007/BF00268499. [Online]. Available: https://doi.org/10.1007/BF00268499.

[10] S. Ulrich and L. d. Moura, *Lean Manual: Array*. [Online]. Available: https://leanprover.github.io/lean4/doc/array.html (visited on 06/19/2021).

[11] L. R. Ford and D. R. Fulkerson, "Maximal Flow Through a Network," *Canadian Journal of Mathematics*, vol. 8, pp. 399–404, 1956. DOI: 10.4153/CJM-1956-045-5.

[12] J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, Apr. 1972, ISSN: 0004-5411. DOI: 10.1145/321694.321699. [Online]. Available: https://doi.org/10.1145/321694.321699.

[13] A. V. Goldberg and R. E. Tarjan, "A New Approach to the Maximum-Flow Problem," *J. ACM*, vol. 35, no. 4, pp. 921–940, Oct. 1988, ISSN: 0004-5411. DOI: 10.1145/48014.61051. [Online]. Available: https://doi.org/10.1145/48014.61051.

[14] J. B. Orlin, "Max Flows in O(Nm) Time, or Better," in *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '13, Palo Alto, California, USA: Association for Computing Machinery, 2013, pp. 765–774, ISBN: 9781450320290. DOI: 10.1145/2488608.2488705. [Online]. Available: https://doi.org/10.1145/2488608.2488705.

[15] D. J. King and J. Launchbury, "Structuring Depth-First Search Algorithms in Haskell," ACM Press, 1995, pp. 344–354.

[16] A. Moine, *Benchmarking Haskell Graph Libraries*, Jul. 2018. [Online]. Available: https://blog.nyarlathotep.one/2018/07/benchmarking-haskell-graph-libraries/ (visited on 06/19/2021).

[17] R. Albert, H. Jeong, and A.-L. Barabási, "Diameter of the World-Wide Web," *Nature*, vol. 401, no. 6749, pp. 130–131, Sep. 1999, ISSN: 1476-4687. DOI: 10.1038/43601. [Online]. Available: https://doi.org/10.1038/43601.

[18] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, *Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters*, 2008. arXiv: 0810.1355 [cs.DS].

[19] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford Large Network Dataset Collection*, http://snap.stanford.edu/data, Jun. 2014.