Understanding functional and equational programming techniques on graph-based problems

Sam Aronson

Professor Dave Wonnacott (advisor)

Senior Thesis

Haverford College
Abstract

Pure-functional programming is a paradigm of programming closely related to mathematical constructs that can offer great power and expressiveness in terms of how to write and reason about programs. Additional benefits stemming from the ease of reasoning include more-reliable and maintainable code, as well as the development of advanced automated tools like identifying concurrency opportunities or generating automated testing.

However, canonical treatment of algorithm design, particularly those related to cyclic graph algorithms, is often imperative and makes heavy use of state, even for algorithms as simple as breadth-first search on cyclic graphs. Advanced techniques like “tying the knot”, or other techniques to represent edges that point to already-defined vertices are required for the self-reference necessary to build the data structures that represent graphs. This project evaluates the current state of graph representations, like those that are naively proposed, or inductive or algebraic graphs proposed for implementation in a lazy pure-functional language and like Haskell in type-safety and performance, as well as propose the implementation of efficient algorithms to solve canonical problems relating to graphs in cutting-edge sound and complete representations.

This project explores the constraints that current functional and type-safe graph representations introduce. Further, it develops equational techniques that can be used to find important information in graphs, like how to find the shortest path between vertices idiomatically in this new paradigm, and how these techniques can be further developed to allow for efficient, trustworthy graph algorithms to be developed.
Acknowledgements

My utmost of thanks to my advisor Dave Wonnacott and Jacob Gorenburg; it has been so incredibly fun to explore this somewhat obscure space of computer science with you both. Thank you to Moses Boudourides for your helpful comments on this manuscript. My friends and family have been instrumental in helping me continue to stoke and support an interest in computing, as have my previous experiences in the First Robotics Competition and programming at the Bakken Museum. Thank you for all you have done to impact my learning and enable me to engage in this project.
1. FUNCTIONAL PROGRAMMING

1.1 Why Functional Programming

Functional programming is a powerful programming paradigm that brings with it a multitude of benefits. Such benefits often stem from a focus on pure functions devoid of side effects or a reliance on mutable state. Because of this, programs written in this style are easier to reason about; this can increase both programmer confidence in the correctness of their programs as well as make available more powerful automated analysis tools like those that write automated unit tests and identify and implement opportunities for concurrency [OGS10] [BG20]. In addition, such expressions, particularly equational functional programming often more closely resembles the mathematical theory behind ones procedures compared to imperative paradigms during and after implementation, helping empower abilities to reason about the program [KT06]. One advantage of these reasoning capacities is enables the possibility of laziness, wherein only information that is needed will be calculated, allowing for more efficient computation as well as extending the possibility for what it is possible to calculate.

Unfortunately, there have been some drawbacks to using pure-functional techniques promoted in languages like Haskell. First is the recognition that nearly every program changes the state of something (take, for example, reading or writing from files through bytestreams) so additional abstractions are necessary which may obscure the operations in an unhelpful manner [OGS10]. Another concern pertains to performance. While Haskell has been found to be significantly faster than interpreted languages like Python, many canonical algorithms for solving problems efficiently have been written in an imperative form, relying heavily upon actively managing state, when those languages are compiled and more efficient in their operation, like C++ [OGS10] [KT06]. The standard approach for languages like Haskell thus far have been to use special objects to manage state, either using the monad or other specifically defined types to manage the state through a function that defines how to move between states. [BG20]. The disadvantage about this approach is that it again makes it more challenging to reason about the state of the program again, and often limits opportunities for concurrence, some of the primary advantages of this paradigm to begin with [OGS10].

For certain types of algorithms, it is easy to write in a pure-functional paradigm without using monads. Examples of such algorithms include some dynamic programming problems and problems that demand traversals of trees which do not require information about previous computation [Ehw01] [BG20]. In these cases, one is always conducting a search in one direction: the algorithm traversal will never encounter nodes of computation that have already been considered in cyclic or self-referential data structures.
1.2 Why Haskell

Haskell has three main features that make it distinct from other programming languages. First, as already touched upon, it encourages a pure-functional style that requires specifically marking any impure code that produces side effects. Beyond that, it also has an advanced type system that encourages greater precision in how types are used, enabling the programmer to have more confidence in the correctness in her program. Combined with tools like pattern matching, this encourages clean and clear code style. Finally, Haskell by default is a lazy language. What this means is that it will only evaluate an expression when it is needed, not when it is defined like in other strict evaluating languages. For example, I could define all even natural numbers as follows:

```haskell
> evens = 2: map (+2) evens
> take 10 evens
[2,4,6,8,10,12,14,16,18,20]
```

And this will generate an infinite list of even numbers. Because Haskell is lazy, I can use this to not only create other calculations or infinite lists, but ask for just a subset in any calculation and only those values will be the only ones generated. So, when I ask for 10 evens to be printed, it only prints the first ten, not all even numbers which is impossible given the fact there is an infinite number of them. This also allows us to use this in computations, like an odds function that simply maps the result of the evens function:

```haskell
> odds = map (+(-1)) evens
> take 10 odds
[1,3,5,7,9,11,13,15,17,19]
```

Take again, only asks for the first 10 values. Note that not only did it only request the first 10 values of the odds list, but in doing so, it only calculated the first 10 values of the even list, so this type of laziness can often propagate throughout a program, enabling immense expressive power. This approach can also be applied to functions and lists like those that generate all prime numbers, and many other types of infinite expressions, even infinite graphs in some cases.

In any self-referential case, like an infinitely constructed list or cyclic graphs, some sort of cyclic structure will be necessary. Combining with Haskell’s laziness, it is possible to define and traverse cyclic data structures recursively without descending into an infinite recursion. The technique of “tying the knot” has been often been to do so, and has been used to implement the Knuth-Morris-Pratt substring matching algorithm in Haskell as well as offering a replacement for symbol tables in construction of the abstract syntax tree in the Glasgow
Haskell Compiler, the main Haskell compiler \cite{PL95}. Any cyclic graph (graph that has an edge that points to an already-defined vertex, causing a cycle) will require some self-referential process. This paper will examine different ways to encode all graphs, including those cyclic in nature.

2. ON GRAPHS

2.1 Why graphs?

Representations of many relevant problems today can be done so using graphs. Everything from navigational and pathfinding systems, to genetics, the world wide web and logistics networks have been modeled using them \cite{KT06}. In computer science, specific applications of graphs include various representing the relations of social networks like twitter, as well as fundamental computing concepts like automata and related concepts in language theory like parsing grammars \cite{Sip12}. Scientists often can use various algorithms to describe properties of these real-life phenomenon: from how to best route one’s package or internet request to how to find the least costly or shortest path of travel. One would hope that any programming language could describe graphs given how fundamental of a concept they are. However, this concern of self-reference arises once again for any sort of cyclic graph, or graph that contains a cycle.

The goal of this project will be to investigate these techniques to see if they can be applied to other imperative algorithms, in particular graph algorithms like graph traversals or shortest-path algorithms. In doing so, what structures must be implemented, and is it possible to implement these in a manner that involves acceptable run-time performance? Other questions will also be considered, like what is the best representation of a graph to facilitate such calculations? Work by Erwig and Mokhov proposes alternative structures that may be more appropriate \cite{Erw01, Mok17}. The goal of this work is to identify how feasible implementation of these algorithms are without explicitly managing state, and what potential pitfalls of such an approach may be. In doing so, I hope to discover practices that encourage the writing of code that is easier to express, reason about, and parallelize.

First, we shall begin by defining a graph mathematically, how they are implemented in various contexts and discuss the cutting edge of graph representation. After, we will discuss how those representations support canonical graph algorithms, as well as possibilities for expanding this frontier.

2.2 Graphs, mathematically

Graphs are canonically described in mathematics as a pair of sets: a set of nodes, or vertices, with a set of edges that connect between vertices. Mathematically, this is often expressed as the pair $G = (V, E)$, where $V$ is the set of vertices, and $E \subseteq V \times V$ means that the edges of a graph are a subset containing somewhere between zero and
all possible pairs of vertices to be connected in the graph. Notice that this definition has very few restrictions on vertex connections, and even allows for an edge between a vertex and itself.

There are many different types of graphs that adapt this basic definition. First, one can distinguish between directed and undirected graphs. Directed graphs can be thought of as having edges with direction. This means that if you imagine $G = (\{a, b\}, \{(a, b)\})$, this would indicate that the graph has an edge from $a$ to $b$, but not necessarily from $b$ to $a$. Vertices are identified using some identifier. These weights often represent costs or capacities in various problems. Undirected graphs, do not have specified direction in the same manner. Therefore an edge between $a$ and $b$ also implies an edge between $b$ and $a$.

![A directed graph, with vertices a and b and an edge from a to b.](image)

![An undirected graph, with vertices a and b an edge between a and b](image)

There are also other modifications that can be made to graphs as well. For example, many relevant graphs have weights attached to their edges, which often represent cost or capacity of passing along the edge. In that case, we would designate $E \subseteq \{V \times V \times W\}$, with a unique vertex pair and where $W$ is the set of weights, likely the integers or natural numbers depending on the context. For weighted graphs, it is sometimes possible for there to be multiple edges between two vertices, this is sometimes called a weighted multigraph.

In their simplest form, graphs can also be represented in other convenient ways. One instance is as a mapping or function between a vertex and the set of vertices connected to it by an edge. Another is through a matrix representation of size $|V| \times |V|$, where each entry represents whether or not there exists an edge between the vertices, or it’s weight. These alternative isomorphic definitions can also aid in representing graphs when programming.

2.3 Graphs, programatically

There are many different ways to represent a graph, not just mathematically but for use in computer programs. Naively, one can represent a graph in a very similar manner to its mathematical definition: define a set of vertices, typically represented by a set of natural numbers, where a number is associated with each node, and a set of edges as a list of pairs of vertices, also stored in one longer list. However, in practice, this implementation is less efficient than many others. To determine, for example, if two vertices have an edge between them, in the worst
case one would need to traverse through all of the edges, or $O(m)$ time where $m$ is the number of edges and $n$ is the number of vertices.

The two most common representations of graphs are related to the two previous noted mathematical representations of a graph. Most common is the adjacency list representation, where each vertex can be mapped to a list of vertices and associated weights, if applicable. This representation is used both in standard imperative algorithm design, as well as functional settings [BG20] [KT06]. Another, less used representation is the adjacency matrix representation, which typically stores a 2D array where the $i$th and $j$th index represents whether there is an edge between the $i$th and $j$th vertices and is typically more efficient for highly connected graphs [KT06]. These representations alleviate some of the access inefficiencies from the naive implementation.

For imperative or some functional implementations, this can be adequate. However, often times one wants to be able to leverage the type system to ensure the correctness of what has currently been implemented. For example, one may want a guarantee that what has been generated when constructing a graph, is in fact a graph. However, if it is possible to generate edges in the program that do not correspond to vertices that exist in the programmed graph, this would be an invalid graph by the aforementioned definitions. Therefore, a different construction of graph that maintains its property of being a graph at all times is desirable, particularly for strongly typed languages like Haskell where the type checker is relied upon heavily to provide confidence in the correctness of a program.

In addition, traversal can be challenging if not impossible to implement both idiomatically and efficiently. Take the naive approach that is promoted by Bird and Gibbons in their introduction to graph algorithms, that includes weighted edges [BG20]:

```haskell
  type Graph = ([Vertex],[Edge])
  type Edge = (Vertex, Vertex, Label)
  type Vertex = Int
  type Label = Char
```

It should be unsurprising that we define a Graph as a list of vertices and edges, an edge by its two vertices and weight, labelling vertices by integers. While typically not defined as separate types, a similar treatment is often used in imperative algorithm design as well. Consider how the following graph would be represented in this manner:
This would correspond to the following graph representation:

\[
(\{1, 2, 3, 4\},
\{(1, 2, "a"),
(2, 3, "d"),
(3, 4, "c"),
(3, 1, "e"),
(4, 4, "b"),
(4, 13, "invalid!"))
\]

However, one of the strengths of Haskell is its type system. Because we are simply defining vertices as integers, the type system would accept a representation like this:

\[
(\{1, 2, 3, 4\},
\{(1, 2, "a"),
(2, 3, "d"),
(3, 4, "c"),
(3, 1, "e"),
(4, 4, "b"),
(4, 13, "invalid!"))
\]

Note that because there is no vertex "13", this is an invalid graph, as not all of the edges are contained in \(V \times V \times W\), with a unique vertex pair, which was required by the mathematical representation and definition of a graph. Therefore, the invariant desired cannot be confirmed using this implementation of a graph. Given
the importance of correctness that is placed in the development of programs in Haskell and other strongly typed
functional programming languages, one would hope to develop a notion of a graph which will continue to be a
graph in all cases. Instead, if one wanted to confirm the validity of the graph, one would have to traverse each
edge to confirm that its associated vertices are valid. More likely, and with many of the most common graph
representations in Haskell, one would encounter a runtime error in the execution of the program in this case,
which is far from ideal. [Mok17] We will now present two alternative graph representations that can be used,
before discussing how to work with these representations, as well as the naive one.

Another issue to consider is the imperative nature of many types of graph traversals. Canonically, when
implementing a depth-first or breadth-first traversal of a graph, a “mark and sweep” approach is used. That
is, once a vertex has been visited, it it “marked,” typically in an array representation of the vertices, indicating
to the program that if it is visited again, its adjacent vertices should not be pursued in the search. This
prevents the traversal from encountering an infinite loop. However, in a functional program, state management
is more challenging; at the very least, the “marked” nodes would have to be passed long using either a monad
or some other type of function. Additionally, most data-structures used functionally, like balanced binary trees,
would have an $O(\log n)$ lookup time, as opposed to $O(1)$ of random-access arrays. [Erw01] Therefore, alternative
representations to solve both the traversal problem and the type problem have been proposed.

3. ERWIG’S INDUCTIVE GRAPHS

In his paper “Inductive graphs and functional graph algorithms,” Erwig proposes an inductive definition of
graphs. [Erw01] The hope is that like other inductive data types, like (linked) lists, or even trees, a special type
of graph wherein the functional definition is in terms of other, subtrees. Because of the potentially cyclic, or
self-referential nature of graphs, this is a more challenging but still accomplishable task. His main proposition
is to still represent nodes with integer identifiers, but to introduce with every edge the relevant new context, or
edges that need be introduced.

3.1 Graph Construction

Therefore, we can represent it using the following types in Haskell:

```haskell
    type Node = Int
    type Adj b = [(b, Node)]
    type Context a b = (Adj b, Node, a, Adj b)
```

Consider the representation of the simple 3-node cyclic graph offered above. In this case, we can represent it as
so
We can also ensure by the end of the representation, that we have consistency with the requirements of such a representation. Note that such consistency could not be ensured until the end, so it is possible to produce partial graphs, or graphs wherein edges refer to vertices that do not exist. Importantly, the type system will be unable to do this checking for the graph, which is often one main advantage of using Haskell.

The advantage of such a representation is that it is incredibly easy to traverse such a graph, without using standard techniques like marking already-visited edges as is done in standard graph searches imperatively. Because each edge is only represented once in the graph, traversals and mappings can be executed inductively without the concern of infinite recursion without keeping track of what has already been accessed.

Furthermore, Erwig has also demonstrated how basic tasks like breadth-first and depth-first traversals can be executed, as well as additional functions like reversing edge direction (take an edge from $a \rightarrow b$ and convert it to $b \rightarrow a$).

Work by Dexter et al have validated this approach by demonstrating that inductive graphs can efficiently and effectively be used for high-performance applications. They developed the DELTA-GRAFH library that uses this structure for high-performance cases. Many of the benefits of functional parallelism can be leveraged with this approach. This helps demonstrate the viability of such an approach both theoretically and empirically.

### 3.2 Graph manipulation

This section will discuss the algorithms that Erwig introduces in his paper, as well as those from the follow-on.

#### 3.2.1 Graph Mapping

As a preliminary example for the type of expressive power that inductive graphs have, particularly in the context of Haskell’s pattern matching, Erwig demonstrates how to implement a graph mapping called \texttt{gmap}. Using the pattern matching definition, we can construct a function that traverses over the contexts to produce a new graph.

\[
\text{gmap} :: (\text{Context } a \ b \rightarrow \text{Context } c \ d) \rightarrow \text{Graph } a \ b \rightarrow \text{Graph } c \ d
\]

\[
\text{gmap } f \ \text{Empty} = \text{Empty}
\]

\[
\text{gmap } f \ (c \ & \ g) = f \ c \ & \ \text{gmap } f \ g
\]
To walk through this Haskell definition, the first line can be interpreted as the type declaration, in this case asking for a function that maps a context to a different context, and a graph. It will then return a graph (this is represented in this way using the method of currying). The next two lines consider the pattern matching cases, first that one should map an empty graph to another empty graph. Then, the inductive definition that maps a graph defined by \( c \& g \), where the context \( c \) is applied to \( f \) and \( gmap \) is recursively called on the rest of the graph’s context.

This definition is very similar to what one would define as a tree or list mapping in Haskell, or any other inductively defined datatype. Using \( gmap \) also allows for easy definition of many types of graph traversals and manipulations. For example, one can reverse the direction of all edges in the graph using \( gmap \) as follows:

\[
grev :: \text{Graph } a \, b \rightarrow \text{Graph } a \, b
\]
\[
grev = gmap \text{ swap where swap } (p, v, l, s) = (s, v, l, p)
\]

In this case, we define the function \( \text{GREV} \) as a function that takes in and returns a graph, and does so by mapping the function \( \text{swap} \), which reverses the order of the context, making it such that all outward facing edges from each node become inward facing ones. This type of definition allows for easy graph traversal of many basic needs. It also allows for relatively simple proofs of correctness as well.

Erwig also demonstrates different forms of folding over the graph as well, as well as other functions like those that easily calculate things like a vertex’s degree. These require another feature called active graph patterns, which would find the term representation where some arbitrary vertex \( V \) is inserted last, so it has the full context of its own representation. Such a representation will typically be stored in either a balanced BST or in functional arrays. Depending on the number of node deletions, the way to find \( &v \) or the context most relevant to \( v \), can be done in \( O(uc \log c) \), where \( c \) is the size of the context of the whole graph, and \( u \) is the number of updates to the graph with optimizations. An optimization using the functional array implementation can bound access at \( O(n) \) in the worst case, with performance usually being better. In some cases it is the case that without modification access of \( &v \) context can occur in \( O(1) \). While this access time is less optimal than those of adjacency matrices, gathering this information is typically not prohibitively more inefficient in most cases where that information is gathered. While the BST implementation has worse worst-case runtime complexity, in practice Erwig claims it is often a better alternative.

### 3.2.2 Implementing Depth-First and Breadth-First Search

A main motivating concern for Erwig is to construct a representation that can idiomatically and efficiently traverse graphs; Erwig’s implementation also allows for effective depth and breadth-first traversals. This is a
helpful contribution as most algorithms will traverse the graph in such a manner, at least as typically presented. As he notes in his paper, using a traditional “mark and sweep” approach will typically give one asymptotically worse results, as instead of a constant access array to mark already visited nodes, binary search trees are often used which have $O(\log n)$ lookup time instead of $O(1)$. Using Erwig’s representation, he proposes the following traversals for breadth-first and depth-first search. To begin, we shall discuss the proposed depth-first search that will produce a list of nodes in depth-first order.

```haskell
dfs :: [Node] → Graph a b → [Node]
dfs [ ] g = [ ]
dfs (v:vs) (c &v g) = v:dfs (suc c++vs) g
dfs (v:vs) g = dfs vs g
```

Here’s how this traversal occurs, assuming $g$ is connected. Considering the base case, once there are no more nodes to traverse, $dfs$ will return no more nodes. To begin, a list containing a single node will be inputted, and we will consider either the second or third case in the traversal. Either the first node in the traversal list will be found in its active context, that is, the form in which the full information about its adjacency will be found. It will then prepend all of its successors (using the $suc$ function), so the children will be accessed before the siblings. If the context $v$ cannot be matched, which will occur when $v$ has already been found, then the traversal will continue with the next sibling. This allows us to fully traverse $g$ in a depth first manner. Furthermore, one can easily implement breadth first search by traversing over siblings before successors, or replacing the list in the second pattern to be $(vs++suc c)$. Erwig offers implementations that improve the computational complexity, as appending to a list naively will result in $O(n)$ time, as opposed to $O(1)$. This also illustrates the power of such an inductive expression.

### 3.2.3 Additional algorithmic capabilities

In addition, Erwig demonstrates additional standard graph algorithms, like Dijkstra’s shortest path algorithm, finding minimum spanning trees, and maximum independent node sets. These implementations, along with additional work conducted by Dexter et al demonstrate the wide-ranging viability of this implementation for even high performance cases, as well as the advantages when treating graph algorithms in an idiomatically functional manner. [DLC16]

### 4. MOKHOV’S ALGEBRAIC GRAPHS

Andrey Mokhov also proposes another representation of graphs, attempting to address the type problem as discussed before wherein the graph type is always a valid graph. Previous work by Gibbons and others presents
attempts at building algebras to define certain types of graphs, like directed acyclic graphs, however, this is the first noted work to generalize such an approach to all graphs [Gib95].

4.1 Graph Construction

Mokhov identifies the issue of the possibility that in generating an graph it is inconsistent. This is still an issue with Mokhov’s construction, as one can create an invalid graph by defining a context with a never-defined node. His representation constructs a graph in such a way that it is impossible to build an invalid graph, and eliminates this issue of partial functions [Mok17].

He does so by defining a graph using the following definition:

```haskell
data Graph a = Empty
          | Vertex a
          | Overlay (Graph a) (Graph a)
          | Connect (Graph a) (Graph a)
```

The empty Graph represents $G = (\emptyset, \emptyset)$, that is the graph without any vertices or edges. Vertex a represents the graph $G = (\{a\}, \emptyset)$, or the graph containing the vertex $a$ and no edges. Note that in Haskell, “a” represents a token of some unspecified type, so it could be represented as an integer or character, or any other datatype. Typically, natural numbers are idiomatic. Overlay is the function that takes the union of their vertices and edges, while Connect is a function that produces the graph that is the union of their vertices, and generates edges between all vertices in each subgraph. Using these properties and subgraphs that share vertices, Mokhov is able to prove that this construction is isomorphic to the standard $G = (V, E)$ construction. In other words, this construction is sound (no non-graph can ever be constructed) and complete (all graphs can be constructed using this method). In other words all possible graphs and only graphs can be constructed using these definitions.

He also proposes a library of functions that can manipulate these graphs, and also offers extensions for such things like directed and undirected graphs, reflexive graphs (that allow a vertex to connect to itself). However, progress in published literature is rather light, so future work is necessary to determine if standard graph algorithms can be written in this representation with reasonable or equivalent time complexity to standard imperative implementations, or even functional implementations like those traditionally offered by Bird and Gibbons or work using Erwig’s inductive construction.
4.2 Graph manipulation

4.2.1 Graph mapping

Mokhov also proposes a gmap function. However, because the main focus of this type of implementation is that which manipulates and then overlays vertices, the same type of “context mapping” that manipulated edges before does not apply in this way. Instead, gmap is defined as being able to manipulate vertex values. One interesting consequence of this is that vertices can actually be merged (by mapping to vertices to the same ID number); however the same types of manipulations are not possible. Generally, this is a shortcoming of Mokhov’s implementation. That is, there are relatively few ways the graph can actually be manipulated in helpful ways directly (some interesting constructions can be created, but not the types of traversals and manipulations that are helpful in understanding properties of graphs like DFS and BFS).

The primary method of ”graph mapping” is through a graph transposition. This is where a function can be defined over the four algebraic base operations to modify how a graph is being treated. One advantage is that this approach will instantly “modify” the graph, or at least how it is interpreted, without any overhead. Of course, traversals will still take the same amount of time. For a base example, consider the transposition that reverses the direction of all vertices as presented in Mokhov’s paper:

```haskell
instance Graph g => Graph (Transpose g) where
  type Vertex (Transpose g) = Vertex g
  empty = T empty
  vertex = T . vertex
  overlay x y = T $ overlay (transpose x)(transpose y)
  connect x y = T $ connect (transpose y)(transpose x)
```

What this does, in short, is defines every operation identically to its typical implementation, except for connecting, which will connect vertices in the opposite direction. This will connect all vertices such that any edge is reversed. This same type of approach can be used to merge vertices, by defining a function over the vertex operation, remove vertices, and make many other types of graph representations. Again, these operations can often happen faster than traditionally given the structure of this data.

Note that by defining this mapping in particular ways, there is extremely expressive powers that can be one method by which properties of graphs usually determined by algorithms like Kruskal’s may be able to be defined. However, at this point there is no published literature directly working with the core of this construction. While algorithms have not been explicitly generated through through these functions, other methods using equational reasoning can be leveraged by building off of the four algebraic constructors.
4.2.2 Graph composition

This section will discuss the equational reasoning possibilities with this approach, which provide much of the power of both proving properties about graphs, as well as allowing for manipulations. The primary advantage is, recalling + to be the “overlay” function and \( \rightarrow \) to be the “connect” function, any algebraic graph can be represented as:

\[
g = \sum_{v \in V_g} v + \sum_{(u,v) \in E_g} (u \rightarrow v)
\]

Using this decomposition can allow us to use more traditional graph algorithms to understand information about these graphs. This can then use more traditional techniques to conduct traversals and gather information about the graphs. Note that this can be generated in \( O(n + m) \) time, however using laziness it doesn’t necessarily mean the entire relation is evaluated. This is one avenue to be able to use already existing graph libraries and developed algorithms, but it circumvents the type safety proposed in this model.

4.3 Efficiency Comparison

While Erwig’s inductive graph representation has shown how to represent graphs in ways that are ergonomic to manipulate inside functional programming idioms, it is still not a type-safe solution, that is, it does not guarantee the invariant that what is represented is necessarily a graph where \( E \subseteq V \times V \). Fortunately, Andrey Mokhov’s novel work has been developed such that a static guarantee of this invariant can be made. Additionally, libraries leveraging this algebraic approach have been developed in many languages, including Haskell, TypeScript, Adga, and F#.\[^{Mok17}\]

This leads to the question, how efficiently are these libraries compared to standard approaches. Previous work has began preliminary benchmarking the alga library, which utilizes Mokhov’s approach against other common Haskell graph libraries.\[^{HPMM+21}\] This work demonstrated that alga often has comparable performance to other Haskell libraries. No benchmarking across this alga library and other approaches has been conducted yet. In addition, even relatively standard algorithms such as Dijkstra’s Algorithm have not yet been Incorporated into the Alga library using the standard representation. This paper shall explore why that may be the case.\[^{Kum19}\]
5. RESEARCH GOALS

There are two goals of this project. First is to begin to gesture towards an answer to the question of what is the performance cost of correctness against other graph representations; second, it is hoped to begin to explore algorithmic design approaches that lead to efficient and idiomatic algorithms and graph traversals in an algebraic graph representation.

To work towards these goals, algorithm design will be attempted to be designed for Mokhov’s Alga library. There are many common algorithm implementations, including Dijkstra’s algorithm which are not yet available as a part of the graph library. For example, it is more costly in Alga to find the neighbor set of a vertex than in other canonical representations as it requires a traversal of the entire graph representation. If such specific traversals can be avoided, and instead a full traversal of the tree is only required once, it may lead to better algorithmic performance. This project shall explore whether such hypothesis are born out in practice, and in what situations that may be possible.
6. IDIOMATIC TRAVERSAL

One advantage of the inductive graph approach is that simple functional traversals can be conducted to check the graph. Of course, breadth-first and depth-first traversals are essential not just for those algorithms, but for many others that rely upon such traversals, like Dijkstra’s. In fact, most algorithms that require a vertex’s neighbors can typically be characterized using such a traversal.

6.1 Considering breadth-first traversal

Consider this breadth-first search traversal as introduced in [Erw01]:

```haskell
bfs :: [Node] -> Graph a b -> [Node]
bfs [] g = []
bfs (v:vs) (c &v g) = v:bfs(vs++suc c) g
bfs (v:vs) g = bfs vs g
```

This function takes in a list of nodes and a graph and returns a list of nodes in the order in which they are traversed. The first list is the list of vertices to traverse, and should be initiated with the start node for the traversal. In the case that no nodes are left to be traversed, the empty list will be returned. For the next case, an “active pattern” is used to determine whether there is a graph context involving the first vertex on the list. If that is the case, the successors of that vertex will be pushed to the end of the stack. If no such context can be found, then the vertex has no neighbors, and the next vertex in the queue will be explored. It should also be clear that with a change as to the order of traversal, this algorithm can easily be ported to be depth-first instead.

This allows for a simple and idiomatic graph traversal. If the graph is represented by a search tree, contexts can be found in \( \log n \) time, whereas it can be found in constant time by representing it using arrays, although in practice search trees have been found to often be faster, particularly without tedious tuning processes [Erw01]. This allows for efficient traversals, in some cases asymptotically identical to standard traversals.

Consider the equivalent breadth-first traversal in Mokhov’s representation, developed for this project:
bfs :: (Eq a) => [a] -> [a] -> Graph a -> [a]
bfs [] _ g = []
bfs (v:vs) p g = v : bfs (filter (\x -> x `notElem` p) (vs ++ vsuc c)) (v:p) g
  where c = context (== v) g

With some minor stylistic differences, this should look fairly familiar. In this case, an empty list of nodes to
traverse returns an empty list of traversed nodes. Like in the other case, we again add any available successors
to the relevant node’s context. However, we also need to do a manual check to ensure that a node hasn’t been
traversed (this is where the filter comes into play). This marks the first note of difference between the two
approaches.

Whereas Erwig’s approach can leverage active patterns which automatically remove already visited nodes,
this is not the case for Mokhov, requiring this to be done manually. In this case, the filter check will naively take
up to \( n \) time to check to see if any successors have already been visited, however this could easily be reduced to
a maximum of \( \log n \) using a balanced search tree. Of course, of interest is what active patterns actually do. In
this case, they are typically implemented by activating a “flag” after being visited the first time, like a standard
approach would do. Note that this is an imperative implementation that removes many of the advantages of
functional programming like persistence and ease of establishing the problem for concurrency in exchange for
single-core asymptotic efficiency; but is also not implemented in a pure-functional manner. This implementation
is because the function represents one “step” in the algorithm, not requiring state and allowing each step to be
persistently encoded.

A final note on time complexity should be considered, and that pertains the context-getting line:

where c = context (== v) g

While this functionality is built in to the algebraic-graphs package, it does so in an exceedingly inefficient way
that requires a full traversal of the entire representation, leading to a time complexity typically referred to as
\( O(s) \). While there is theoretically no bound as to the size of the representation in relationship to the nodes and
edges, one can assume that any reasonable relationship wouldn’t superfluously encode edges twice unless it was
more efficient to do so than individually connecting each edge, leading to what I will refer to as the assumption
of reasonable bound of \( s \leq 4e \). In this case, we can assume that in association with each edge \((u, v) \in E, \) that
it overlays a connect between the two vertices, adding a size of at most four to each edge. It is clear that doing a full traversal of the representation is less than ideal, and less efficient than, say a adjacency list representation which offers much faster access to the list of successor nodes. This leads to the question as to whether this is avoidable in this representation.

What this means is that every step of this algorithm takes $O(s)$ time, meaning that instead of a BFS taking $O(n + m)$ time as it does canonically, it will take $O(s \ast (m + n))$ time, which will often be significantly worse and is not ergonomic considering how the graph is represented. Thus, this paper shall investigate if there are traversals that better perform and reflect the data structure as it is constructed.

6.2 The proximate neighbors problem

This challenge relating to finding the proximate neighbors is a significant one; given the fact that many common graph queries are related to its relations to its neighbors. Other greedy algorithms like Dijkstra’s, or the A* algorithm for pathfinding all rely upon easy access to neighbors. This is part of the reason why Erwig’s graph representation relies so heavily upon surfacing neighbors.

It should be clear that there is no guarantee until the full graph traversal that there are not additional neighbors to a given node $v \in G$. This is because at any arbitrary point in the representation, a “connection” can be built between $v$ and $v'$, meaning that the entire graph needs to be traversed to ensure that this is not the case.

There are two questions at this point to ask: first, how might we efficiently ascertain neighbors in this paradigm? Next, need we have the set of neighbors for all common queries on graphs? Or can there be some evaluations conducted independent of the neighbors? This second question shall be considered in the next section, but first, a note on the former.

There are two potential approaches that have been taken thus far for more efficient neighbor traversals. The first option is to build the adjacency map after the graph has been constructed in canonical form; in this case, it will take $O(s + m \log m)$ time to traverse the representation and construct the graph. This will generally be preferable than requiring an $s$-length traversal any time neighbors are generated, because more information can be collated during that one pass. [Mok22]

Alternatively, one can actually build using the four algebraic constructors directly to an adjacency map. However, because trees need to be built representing the neighbors, connecting or overlaying graphs will take $O((n + m) \ast \log(n))$ time, which need to be done any time those operations are built into the graph. These will then allow standard algorithmic approaches to be used, while still maintaining the invariant that only graphs
can be built. As it should be clear, there is a performance cost to generating graphs in this manner compared to canonical imperative implementations.

6.3 Shortest-paths without complete neighbor information

The main question to thus address is whether we can approach problems like finding the least costly path through a graph pure-functionally, equationally utilizing the algebraic graph representation. One common shortest-paths algorithm is inspired by the works of Floyd and Warshall [KT06]. As described, it establishes a dynamic programming problem over the graph. The algorithm works by considering sets of vertices increasing in size. At each step, it establishes the shortest possible path from every vertex to every other vertex using only the vertices already considered, with the shortest path being $\infty$ if there is no feasible path. Unlike Dijkstra’s algorithm, this algorithm finds the shortest path between all vertices to all vertices, as opposed to Dijkstra’s which finds the shortest path from one vertex to all vertices. It also can consider edges with a negative weight, which Dijkstra’s cannot.

Already, this algorithm considers vertices in an arbitrary manner as opposed to any sort of edge-related traversal like a breadth or depth first search. Therefore, it appears to be a reasonable candidate for the type of traversal that Mokhov’s representation most closely enables, because doing one traversal of the representation will present edges in an arbitrary order.

To do so, we shall establish attributes to the Mokhov representation of the graph, which will take the form of a tree. Consider the following graph, and one of its representations as an algebraic graph.
We will start by beginning to define a number of relevant attributes to the graph, so that we can build up the relevant information needed to find the best paths. These attributes can be built from the bottom of the tree up, and be calculated at each intermediate step for every subgraph built while constructing the complete graph.

First, we need to define the vertices that can be found for each subgraph.

For nomenclature, let $G_x$ be a graph with associated vertices $V_x$, and edges $E_x$, which is a set of tuples $(u, v, c)$, that is the pair that contains start of the edge, end of the edge, and associated cost of the edge between.

First, we begin by defining the vertex attribute. This attribute calculates all of the vertices defined in the the given subgraph; that is, the graph defined by the node and any of its children.

We define vertex as follows: For connect and overlay, the vertices are the union of the vertices of each of the two constitutive subgraphs of the children. For Vertex, we define the vertices to be the set including just the vertex that has been introduced. For Empty, there are no vertices in the graph.

At this point, we can also define the set of edges that will be introduced at any given point as well. We define them using the following formulae:
For the edges, we recognize that the empty graph and the graph constructed with one Vertex will not have any edges (although connecting the “vertex” graph with itself will result in a graph with a self-referential edge). Overlaying two graphs will result in an edge set that is the union of the two subgraphs. Connecting two graphs will maintain all already-established edges, but also include new edges defined by the connect, that is the cartesian product of the sets of vertices in the left and right-hand side subgraphs, at cost $e$ (the arbitrary associated cost with the connect). Once we have established the edge sets, we can then construct the paths.

The set of paths are the sequences of edges that form a followable path from one vertex to another. The goal is to find the least costly paths to get from any vertex to another. For one-vertex and empty graphs, the optimal paths are trivial to generate. For other graphs, the set of paths can be calculated using a fixed point calculation; at each stage new paths will be generated and may allow for the generation of even longer paths. Once all paths have been generated for acyclic graphs, the fixed point calculation can finish. For cyclic graphs, there are an infinite number of paths because any cycle can be followed ad infinitum. Instead, following from the problem definition of Floyd’s algorithm, if we assume no negative cycles (that is, paths that start and end at the same point with a negative total cost), then we know that any path that includes a cycle is not optimal or equivalent to an optimal path without a cycle; therefore, we can generate all paths smaller than the size of the edge set, and collect a set of all paths without a cycle (although it is possible that paths with cycles may still be included). When generating the paths, the cost of the paths are also calculated (that is, the sum of the edge costs of the path).

At this point, all potentially-optimal paths have been generated, and the minimum cost path between every set of pairs can be found from the set and generated, finding the minimum distance path between all reachable pairs of vertices in the graph $G$, doing so using equational reasoning on the graph.

We have now demonstrated the viability of using equational reasoning on an algebraic graph representation to find common queries, like the least costly path through the graph. While the approach is limited to calculations that are not strictly reliant on doing a breadth-first or depth-first traversal, it is possible for algorithms that can be developed using an arbitrary traversal. At this point, the future possibilities of this project will be
explored, including efficiency implications and data structure when it comes to making the calculations for this shortest-distance algorithm that has been developed.
7. FUTURE DIRECTIONS

There are a number of future directions that can be taken with this original proof-of-concept algorithm. While it demonstrates the viability of calculating some things of interest directly from an algebraic tree representation, there are many efficiency gains to be considered. First, the mode of edge storage should be considered, potentially using a more efficient structure like a balanced search tree. Next, the algorithmic complexity of generating the possible and best paths should be further investigated; it is quite likely that there are efficiency gains to be found by more carefully calculating only paths that are possibly optimal. Further, the performance of this approach should be better characterized, particularly using experimental methods given the subtleties of lazy evaluation and the possibilities of even greater efficiencies.

Beyond shortest paths, a further question remains to be answered as to how many different queries can take this arbitrary traversal approach to see how far Mokhov’s representation can be extended for graph comprehension. Further research should be conducted into whether other queries can also be expressed without reference to neighbor sets for other common graph queries, like max-flow min-cut for instance.

8. CONCLUSION

This thesis has explored idiomatic functional graph representations. In particular, it has explored algebraic type-safe representations. Constraints on finding neighbors have been explicated for type-safe representations. Further, a process for finding the shortest paths in a graph has been developed that is idiomatic and equational on the graph representation. Hopefully these techniques can be further adopted to demonstrate how to write code for graphs that is not just reasonably efficient, but easy to understand and develop from first principles and thus be trustworthy as it is written.
Annotated Bibliography

[BG20] Richard Bird and Jeremy Gibbons. *Algorithm design with Haskell*. Cambridge University Press, Cambridge ; New York, NY, 2020. "This book is a textbook that covers many topics that a standard algorithm design course, however in Haskell. In doing so, it employs lazy, pure functional design but often resorts to state management. I will use this book to understand when monads are currently employed, as well as understand the current approaches that are being used in lazy pure functional algorithm design.”.

[DLC16] Philip Dexter, Yu David Liu, and Kenneth Chiu. Lazy graph processing in Haskell. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, page 182–192, New York, NY, USA, 2016. Association for Computing Machinery. This paper uses an inductive graph definition similar to Erwig to develop the type of high-performance graph library that could be used for many applications, like those in social networks. The authors simulate Twitter usage to demonstrate viability quantitatively of performance viability.

[Erw01] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, September 2001. This paper proposes an alternative way of storing and thinking about graphs instead of the standard $G = (V,E)$ and proposes how this structure can lead to new approaches in functional algorithm design. I will use this paper as a basis to think about more complex types of graph algorithms, like network flow.

[Gib95] Jeremy Gibbons. An initial-algebra approach to directed acyclic graphs. In Bernhard Möller, editor, *Mathematics of Program Construction*, pages 282–303, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. This paper both describes initial algebraic work on directed acyclic graphs, as well as describes other work in this area. While this is novel work, it does not begin to cover the general case like Mokhov proposes.


[KT06] Jon Kleinberg and Éva Tardos. *Algorithm design*. Pearson/Addison-Wesley, Boston, 2006. This book is a text that is used in many undergraduate algorithm design courses. It treats algorithms topics with a fairly standard imperative perspective. I will use this book both to understand what the standard imperative approaches to many algorithms are and their performance, as well as use it to understand the few places where recursion or functional programming may be employed.

This paper provides a formal explanation of monads and state management techniques in Haskell. I will use this to help understand the mathematical backing for monads and the justification in that use of technique in the design of the language and approaches taking to implementing solutions in Haskell.

Andrey Mokhov. Algebraic graphs with class (functional pearl). In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, ICFP 17, pages 2–13, Oxford UK, September 2017. ACM. This conference proceeding also addresses ideas about how to represent graphs for ease of functional algorithm problem solving. I will also use this text to further my own approaches when attempting to apply these techniques to more complex graph problems.


Bryan O’Sullivan, John Goerzen, and Donald Bruce Stewart. Real world Haskell: code you can believe in. O’Reilly, Sebastopol, CA, 1. ed., [nachdr.] edition, 2010. This is a book that introduces Haskell to programmers. It both helps provide background information on why Haskell is a helpful language as well as many bits of a practical introduction to the language as I look to implement algorithms in it.


Twan van Laarhoven. Knuth-Morris-Pratt in Haskell. This blog post provides an example of an algorithm that requires self-reference and how Haskell’s lazy evaluation can work to successfully use the ‘tying the knot’ technique. I will use this to see if it can also be used for more advanced graph-based algorithms.