Exploring Functional Graph Representations for Max Flow

The Abstraction of Functional Data Structure Design Techniques

Harrison Weinstock

Senior thesis submitted in partial fulfillment of the degree requirements for
Computer Science at Haverford College

Advised by David G. Wonnacott
Department of Computer Science
Haverford College
May 5, 2023

Contents

Abstract: This paper investigates the techniques involved in the design of functional data structures and how they are realized in a functional implementation of the Ford-Fulkerson max flow method. Specifically, we focus on functional graph representations and how some of the initial issues can be handled safely without affecting user facing code. We examine the difficulty the functional paradigm presents in working with cyclic structure and how this shows up in graph traversals. We review Erwig's induction graphs and Mokhov's algebraic graphs to see two potential solutions to the same problem of handling cyclic structures in a way that allows for easy traversal. Finally, we analyze functional implementations of the max-flow algorithm to exemplify how the tools for abstraction used in the data structure design carry over to an important and common algorithm implementation.

1 Introduction

Functional programming techniques promote an ease of reasoning and immediate concurrency that is not seen in their imperative counterparts [Oka96]. However, while the paradigm is well known and broadly used, the data structures designed within the paradigm are often underrepresented and understudied in the field of computer science. Functional data structures host many properties that imperative data structures do not, and vice-versa, which often makes it difficult to switch between the two. Specifically, the implementation of algorithms originally designed with an imperative nature in mind can be difficult to translate to an implementation utilizing functional data structures without compromising on either efficiency or readability of user-facing code.

A common example of a challenge in working with functional data structures for popular algorithms is that of representing and working with graphs. Representing cyclic structures and specifically graphs within functional programming is understudied, as seen by how graph representations are absent from some of the most well known functional data structure textbooks [Oka96]. A large explanation for this is that graphs have the potential to be cyclic and it can become difficult to traverse cyclic structures without repeating nodes without auxiliary information. In the imperative approach, acyclic traversals are achieved by marking vertices within the data structure which cannot be done when the data structure is immutable. Therefore, the functional paradigm for defining objects becomes problematic when cycles could exist since the detection of cycles cannot be easily built into the data structure in the same way.

This paper aims to investigate the problems that arise when working with functional graph representations and how they handle the cyclic structure of graphs without losing the benefits of functional design. Specifically, we focus on how these representations are able to abstract away some of these issues, such as cyclic structure, when working with functional data structures. This paper builds on previous reviews on investigating specific problems functional data structures face, by focusing on the higher level user abstraction of the solutions to these problems [Aro22] [Gor22].

The common example highlighted in previous work of how cyclic structures are a challenge to the functional programming paradigm is that of graph traversal [Aro22]. Now, consider an imperative graph traversal algorithm where we can mark each node we have already visited to avoid repeats. However, in a functional data structure, we don't want to be managing the state of the marked nodes as we program or changing the graph at all since it would be immutable.

Many functional graph data structures exist to address this issue and also promote functional implementations of classical graph algorithms. As a tool for displaying the abstraction these data structures provide, this paper aims to address the problem of implementing an efficient maximum flow algorithm using these functional data structures. Maximum flow is commonly solved via Ford-Fulkerson, an approach always

given imperative implementations. By looking at the existing tools and techniques used in the design of functional data structures, we can investigate why this algorithm might be a difficult one to translate into a functional approach and how some of these issues, such as cyclic structure, could be safely hid behind a curtain from the user. While some of these problems may be 'unavoidable', the ability to hide these challenges behind abstraction allows for the user to experience the benefits of the functional paradigm without worrying about low-level implementation details such as how to handle cyclic structure in their functional data structure.

2 Functional Data Structures and Abstraction

2.1 Introduction to Functional Data Structures

Purely functional data structures refer to a paradigm of data structure design working with *immutable* objects. These data structures are designed to to be used with a functional paradigm and therefore their design reflects these principles in their inductive approach to describing data. Note that an object is immutable if it can not be changed or modified after its creation. Thus, an immutable object is not able to track state on its own. However, when designing functional data structures, we usually require a much stronger property of *strongly immutable*: when the object itself and its corresponding sub-fields are all also immutable. While this requirement may seem strict, it is directly in line with the functional paradigm where rather than track the state of the program within the objects being used, we track the state of the program by analyzing the functions being called and their parameters.

In order to grasp how to work with immutable data structures, consider an immutable binary search tree implementation. Suppose we have some tree T and we want to add an item a to it. To do so, we will traverse the tree such that we end up at some node p which is who we want to define as the parent of our new item a. As we traverse p we define a new tree T' such that when we reach p, we can redefine p to have a as a child. Then, we have that T' is the new version of T containing a.

While strong immutability may initially appear as a handicap in the design of efficient and effective data structures, it instead guarantees very strong properties not provided by the imperative counterpart [Oka96]. The first property worth mention is the idea of *persistence*.

Most common imperative data structures can be thought of as *ephemeral*, which means that any change to the object destroys the previous version of the object [DSST89]. For example, if we consider the standard implementation of an array, changing the array gives us a new array. Consequently, the old array no longer exists in memory.

In contrast, persistent data structures allow access to previous version of the object [DSST89]. Now, consider a functional and therefore immutable data structure. Because it is immutable, we aren't going to modify the object itself, but rather use it

as a building block to build other objects. For example, if we consider an immutable array A and we want to change it somehow. We must build a new array A' constructed from A. However, since we never modified A, we still have access to it even after making the change to construct A'. Thus, one can see that immutability guarantees persistence. This idea of building A' from A gets into another core idea of functional data structures in that they are almost always defined recursively.

More formally, a recursively defined data structure is commonly referred to as *inductive* [Oka96]. The property of inductive when referring to a data structure can be thought of as defining the data structure piece by piece where each piece only has the information relevant to that piece. A classic example of this property is the linked-list.

Another property of functional data structures beyond the implications of their immutability is that their design is based around functional programming paradigms. While many paradigms exist, one of large importance is that of *lazy evaluation*. Lazy evaluation is an implementation level behavior of a programming language where parameters are computed when they are needed. Lazy evaluation is most often paired with the *non-strict* paradigm. Consider the example of the AND function and how in most languages if the first term evaluated to true, the following terms are not evaluated. However, for other operations, such as multiplication, the same approach cannot be taken to evaluation. If we consider the expression $x * (1/x)$ and set it to 0 the moment we knew $x = 0$, we would be incorrect in computing the whole term since the second term is undefined. Thus, we must define multiplication in a *strict* manner such that it always looks at the sub-expressions being evaluated.

Lazy evaluation can be contrasted with *eager* evaluation where all parameters are evaluated when they are seen. Lazy evaluation naturally comes with the non-strict paradigm while eager evaluation naturally comes with the strict paradigm.

Highlighting how lazy evaluation differs from eager evaluation is important because it motivates much of the design choices behind purely functional data structures. Additionally, one can see that lazy evaluation and inductive data structures are a very natural pairing of paradigms.

Finally, before moving on to more complicated examples of functional data structures, consider this set of more well-known examples [Oka96]:

- Binary search trees.
- A stack.
- A queue or double-queue.
- Random access lists.

2.2 Example: Persistent Union-Find

Recall that the goal of a union-find data structure is to store a collection of disjoint sets with the core operations being that of *create*, $find$, and *union*. We can describe the *create* method as a way to initialize the partition, the *union* method a way to combine two disjoint sets within a partition, and $find$ as a way to locate which set an element is located in within a partition.

An optimal imperative data structure is known and has been thoroughly analyzed that uses arrays to represent the partitions [AAHJUJ$^+$83]. The core idea being that we store each disjoint set as a tree where all non-root nodes point to their parent, and the root points to itself. This root node is then called the *representative* of the set. To demonstrate the idea, consider the example where the set $\{0, \ldots, 10\}$ is partitioned into zero-valued, even, and odd integers with 0, 1, and 2 being the representatives. The trees are depicted in figure 2, with the array representing the tree being in figure 1.

Index	0	1	2	3	4	5	6	7	8	9	10
Value(parent)	0	1	2	1	10	7	2	1	6	3	2

Figure 1: Array for Union Find Partition

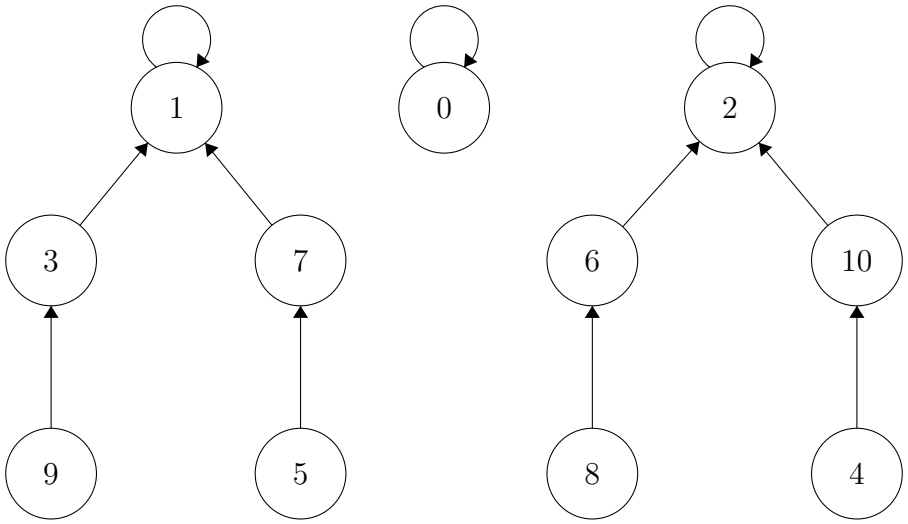

Figure 2: Trees for Union Find Example

These trees are then represented in arrays where each index i represents a value in some set, and the corresponding value at index i denote the parent of index i in its corresponding tree. We can describe these arrays as a type of partition and give the signatures for the core operations of this data structure in figure 3.

create: (**int** n) → **partition** P
union: (**partition** P, **int** rep1, **int** rep2) → **unit**
find: (**int** item, **partition** P) → (int rep)

Figure 3: Imperative Union Find Outline

We adopt the notation of **unit** to refer to an in-place modification from Conchon's paper on a functional union find data structure. [CF07].

Now, the question presents itself of how to create a functional counterpart to represent the union-find data structure. This section highlights some of the core tactics used by Conchon to propose such a functional data structure [CF07]. The first piece of this representation that must change is that the *union* operation is in in-place adjustment, meaning that we directly modify the data structure. However, we want our functional data structure to be immutable. Therefore, we want our partitions to be immutable arrays. Thus, we must adjust *union* to return a new partition, rather than adjusting the existing one in place. This gives us,

union: (**partition** P, **int** rep1, **int** rep2) → **partition** P'

Now, this gets us closer to the functional representation we want, however there are other subtle areas where in-place modifications are done on the partitions. In the optimization of the efficiency of the data structure there is a *path compression* step where when *find* is called on any value i, we modify the partition array in place [CF07]. Specifically, as we work our way up the partition tree containing i, following the parent node pointers, we modify the pointers of the nodes of the tree such that their parent pointers point to the representative for the partition. This step is done to minimize the number of pointers one must follow to find an element in future $find$ invocations.

One simple idea is to modify $find$ to return a new partition, but this modifies user-facing code and is not intuitively in line with the purpose $find$ serves. Ideally, we want to hide this optimization behind an abstracted interface so that the user-facing code does not change and the user is able to reason without worry of the optimization techniques used.

Conchon proposes the use of an auxiliary function, $findAux$ that creates a new persistent array with path compression applied as well as return the representative containing the item requested. This auxiliary function is then called within the $find$ method such that user facing code is not affected and we are able to create a new partition array, achieving a best of both worlds scenario [CF07].

Another modification Conchon uses is the use of a *functor* to hide the use of persistent arrays within the data structure. Functors are an abstract module that describes any sort of object that can be mapped over, such as arrays or lists. [has]. Functor's are a very powerful abstraction tool in representing data and when provided

with more structure become what is known as a *Monad*. Mondas will be expanded on in the next section.

In addition to the tools outlined above, Conchon proposes a few other minor techniques that allow them to argue that the functional data structure presented in the paper is an equally efficient alternative to the standard imperative implementation [CF07]. However, with the functional version we are guaranteed the strong property of persistence from the immutability of the partition arrays, which is not true of the imperative design.

2.3 Monads

Alongside our functional data structures exist a design principle called *monads*. Monads can provide modularity, flexibility, and readability improvements when used correctly [has]. Monads are typically used as a higher level design to work with functional data structure and are not themselves a data structure.

Before formalizing the idea of what a monad exactly is, let us first consider a simple example and the purpose that it serves.

Suppose we are trying to evaluate basic arithmetic on integers. Consider the pseduocode in figure 4.

Type $Op = + \mid - \mid * \mid \div$
Type $Expr = (n \in \mathbb{Q}) \mid (Expr\ Op\ Expr)$
Function *Evaluate* ($Expr$: E)
if $E \in \mathbb{Q}$ **then**
  Return E
else
  Define E_1, E_2, and Op such that $E = E_1\ Op\ E_2$
  Let $E_1' = Evaluate\ E_1$
  Let $E_2' = Evaluate\ E_2$
  Return $E_1'\ Op\ E_2'$
end if

Figure 4: Eval. First Version

This code provides a recursive design for evaluating expressions. However, there is a fatal flaw in the implementation above. Suppose we are evaluating the expression $(2+(1\div(5-5))$ and let us assume strict evaluation for the purpose of ease of reasoning. Then, we will evaluate the sub expression $(5-5)=0$, then plug it into our larger expression for $(1\div 0)$. This will result in an error since our current procedure assumes all expressions will evaluate successfully. We can fix this with the following addition before the final return statement in the code of figure figure 5.

. . .
if Op is $\div$ and E_2' is 0 **then**
  Return *undefined*
end if
if E_2' is *undefined* or E_1' is *undefined* **then**
  Return *undefined*
end if
Return E_1' Op E_2'

Figure 5: Eval. Second Version

This is now a correct program since it can handle the case where we divide by 0 and will return *undefined* as appropriate. Additionally, it handles the case where we get an undefined sub expression, and must carry it up to the top level expression.

However, there is still a flaw in this program. The clear and concise reasoning has been blurred away by the excessive number of return operations to handle all possible errors. To begin to fix this, let us define a new *MaybeNum* type in the code of figure 6.

Type MaybeNum$\langle n \rangle$:
  value(n),
  undefined.

Figure 6: $MaybeNum$ Definition

Note that the *value* operator here is a simple function that returns the number value of n. This operator is commonly referred to as a *unit* operator. With this extension, we can introduce the bind operator notation >>= from Haskell [has]. The >>= in this context can be read such that $a >>= f$ means 'if a is defined, compute $f(a)$'. Now consider the following final version in the code of figure 7.

. . .
if Op is $\div$ and E_2' is 0 **then**
  Return *undefined*
end if
Return $E_1' >>= (\lambda E_1^* : E_2' >>= (\lambda E_2^* : \text{value}(\frac{E_1^*}{E_2^*}))$

Figure 7: $Eval$. Final version

Now, this reads the same as the error handling above but gives us a succinct notation and structure to express our solution. Rather than worry about the error

handling, we can store that information within the types and abstract away the error handling.

We can then define the $MaybeNum$ monad to be the following combination:

- The **Monadic TypeL** MaybeNum.
- The **Unit:** value().
- The **Bind Operator:** >>=.

This only shows one such example of a monad $MaybeNum$. However, one could define a monad to serve other purposes with the structure provided as long as it satisfies the core requirements of a monad in that the *unit* operations is a left and right identity while the *bind* operator is essentially associative [Wad90]. Later on, we will see how this design principle can help us abstract away the cyclic problem in graphs.

3 Functional Graph Data Structures

Let us begin with a simple example undirected weighted graph we want to represent depicted in figure 8.

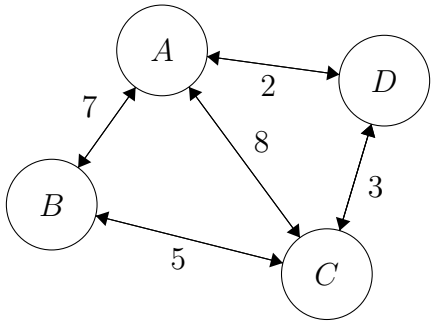

Figure 8: Graph Example

3.1 Algebraic Graphs

The first solution to this problem we will analyze is that of Andrey Mokhov's algebraic graphs [Mok17]. Mokhov defined a functional graph data structure from the basis of a graph algebra. Mokhov's describes a set of four core graph construction primitives that allows the construction of any graph. Let us define the set of all graphs as $G^* = \{G = (V, E) : \text{G is a graph}\}$. We can define then describe Mokhov's data structure as follows:

- G_e: an empty graph or the identity graph in this algebra.

- $(\{v\}, \varnothing)$: a single vertex graph with vertex v.
- $U : G^* \times G^* \to G^*$: an overlay operation such that for graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we have $U(G_1, G_2) = (V_1 \cup V_2, E_1 \cup E_2)$.
- $C : G^* \times G^* \to G^*$: a connect operation such that for graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we have $C(G_1, G_2) = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(v_1, v_2) : \forall v_1 \in V_1, \forall v_2 \in V_2\}$.

Notice that the connect and overlay operations are very similar with the distinction being that connect forms new connections between all vertices in the two sub-graphs. An interesting property of Mokhov's algebraic graphs are that they guarantee type safety [Mok17] [Aro22]. That is, it is not possible to construct an invalid graph from these primitives. Specifically, they handle the case where a vertex referenced in any edge is not a vertex in the graph, which is not guaranteed by most imperative graph data structures or even most functional data structures.

Now, returning to our example from earlier, we represent the unweighted version as the tree in figure 9 where circle nodes represent operations and square represent single vertex graphs.

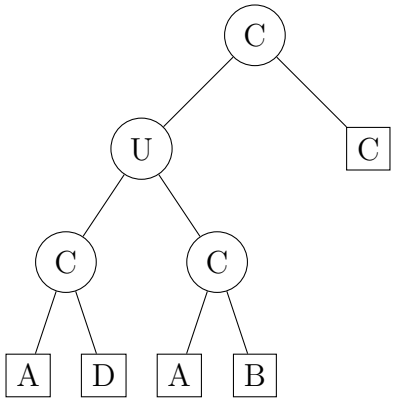

Figure 9: Algebraic Representation in a Tree

Notice however, that this construction is not unique. In fact, there are many possible constructions. Consider the following as an alternative in figure 10.

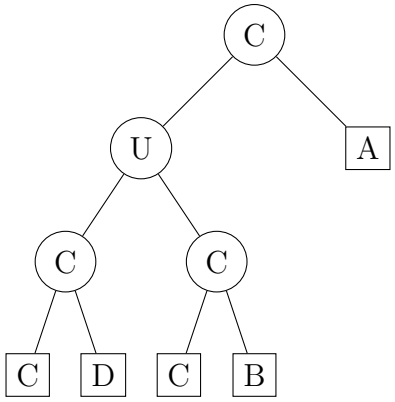

Figure 10: Alternative Algebraic Representation

We will see later how these ideas can be expanded to represent weighted graphs.

3.2 Induction Graphs

Now let us turn to another functional graph representation. Recall the discussion of inductive data types and notice that graphs are not inherently inductive. However, Erwig proposed the idea that we pretend they are in order to define an inductive data structure to represent them [Erw01].

The core idea is that we can think of a graph inductively by defining it as either the empty graph or a fixed vertex with its edges combined with the rest of the graph. This idea is analogous to that of a linked list except that the head of the list becomes the node and its edges, or the context as Erwig calls it, and the rest of the list is now the rest of a graph. We can define the type for integer labeled vertices formally in figure 11.

Type *Node* = Int
Type *Adj* b = [(b, *Node*)]
Type *Context* a b = (*Adj* b, *Node*, a, *Adj* b)
Type *Graph* a b = *Empty* | *Context* a b & *Graph* a b

Figure 11: Induction Graph Pseudocode

Now, if we enumerate the vertices in alphabetical order, we can represent our original graph as depicted in figure 12.

$$([('B', 7), ('C', 8), ('D', 2)], 1, 'A', [('B', 7), ('C', 8), ('D', 2)]\&$$
$$([('C', 5)], 2, 'B', [('C', 5)])\&$$
$$([('D', 3)], 3, 'C', [('D', 3)])\&$$
$$([], 4, 'D', [])\&$$
$$Empty$$

Figure 12: Induction Graph Example

Because the graph is undirected, the adjacent edges must be duplicated in the context.

While the representation is not as elegant as that of Mokhov's, it makes up for it with its ability to naturally store weighted edges, a limitation of Mokhov's original design [Erw01] [Mok17]. However, we will see later an extension of Mokhov's that allows for weighted or labeled edges.

3.3 Solving the Cyclic Problem

3.3.1 Imposing direction

As was highlighted earlier, representing graphs as immutable objects presents a natural struggle in handling how we perform acyclic traversals of cyclic graphs. However, both presentations of functional graph data structures above solve this problem via the same general idea: imposing direction on the graph. Notice that a graph is inherently directionless in that no vertex or edge is given priority over any other. However, this can be changed. In the example of Mokhov's algebraic graphs, we are able to represent the graph as a tree, an acyclic object, of algebraic operations. Similarly, Erwig's induction graphs treat the graph similar to a linked list and impose an ordering of the vertices base on the order their contexts are added to the graph.

3.3.2 BFS and DFS on Inductive Graphs

For Erwig's induction graphs, breadth first search (BFS) and depth first search (DFS) have been implemented as efficiently as on an imperative data structure counterpart [Aro22]. Specifically, let us describe the ideas behind the DFS implementation on Erwig's induction graphs and the adaptation for BFS will become apparent.

We can describe the goal of DFS as to output a list of nodes in depth-first order starting at some given node. As input, we will have the argument graph G we want to search and a list L of nodes in the graph, where the first node in the list corresponds to the initial vertex in our depth-first order list. This list of nodes is referred to as a *traversal list* and allows us to track which nodes are yet to be visited [Erw01]. As a base case, if the traversal list is empty, we are done, since this implies we have visited all of the nodes of the graph. Otherwise, there exists some first node v in L.

We then make use of *active pattern matching* to find the context of v in G [Erw01]. Active pattern matching is an extension of pattern matching that modifies the objects based on how the pattern was matched. Specifically, in the case of Erwig, we use pattern matching to efficiently determine the context of our vertex v, or in other words all edges going in and out of v. Then, we make this an 'active' pattern by removing these edges as well as the vertex v itself from the corresponding graph representation automatically upon matching the pattern. The optimization of the implementation of such active pattern matching is beyond the scope of this paper, but is briefly touched on in Erwig's paper [Erw01].

Before moving on, however, an important aspect worth highlighting is that this active pattern matching scheme utilizes a monad, specifically the maybe monad example from earlier, to easily work with matching cases that could correspond to a matched case, or nothing. Using a monad here allows us to avoid additional casing on each match, and use the >>= notation to feed it into the following logic. Thus, we once again find the basic functional data structure design techniques useful in aiding the abstraction and design of more complicated objects.

Now, continuing with the description of how DFS is performed on an inductive graph, recall that this context given to use by this active pattern matching scheme gives us all incoming and outgoing edges with v in the graph. Then, using this context, we can continue our search on the remaining graph. We make use of a queue data structure to store the successors of v. Specifically, we add them to the queue in such a way that we search the successors of v before traversing the other nodes in the queue [Erw01]. This allows us to ensure a depth-first ordering. Notice that by modifying this to search the nodes in the queue first, we will achieve a breadth first ordering. Finally, if at any point, we run out of successors and the pattern matching fails to find more contexts, we apply the same approach with the next node v' in L.

Note that this last case of working with the next vertex in the traversal list only applies when the graph is disconnected. In the case of a connected graph, the context of any given node should be reachable from the context of any other, and therefore we would eventually find it. Therefore, if it is known that the graph is fully connected, then this *traversal list* can be replaced by a starting node.

4 Different Solutions to Max-flow/Min-Cut Problem

Before exploring functional adaptations of common algorithms to solve max flow, it is important that we highlight what those common approaches are, and the challenges that may arise when implementing them on functional data structures.

4.1 Ford Fulkerson Method and Edmond Karp Algorithm

The Ford Fulkerson method is a greedy approach to solving for the maximum flow of a network. Ford Fulkerson is commonly referred to as a method rather than an algorithm because the original approach did not specify some of the details of its implementation [FF56]. Specifically, recall the general idea of Ford Fulkerson: while there exists an augmenting path P from the source vertex s to the sink vertex t, increase the flow along that path. We say that a path P is augmenting if more flow can be sent through the network. In other words, the capacity of each edge on the path exceeds the current flow on each edge [FF56].

However, the algorithm for finding such an augmenting path is not specified leading to different algorithmic complexities depending on how the path is chosen. In one of the most common implementations, the Edmond-Karp implementation, this path is found by utilizing a breadth first search (BFS) resulting in a complexity of $O(nm^2)$ where $n = |V|$ and $m = |E|$ [EK72]. Additionally, the Edmonds-Carp implementation has been shown to guarantee termination, which cannot be done on the general Ford-Fulkerson method [CLRS22]. In the original Ford Fulkerson method, we can describe the complexity as $O(fm)$ where f is the maximum flow on the graph, however this becomes problematic if we do not know an explicit bound on the maximum flow f. Additionally, there exists Dinic's algorithm, a mispronounced algorithm from Israeli computer scientist Yefim Dinitz, that uses a combination of depth first search (DFS) and BFS to achieve a complexity of $O(n^2m)$ [Din06]. Thus, depending on the sparsity of the graph, the best implementation of the Ford Fulkerson method varies.

5 Max-Flow Implementations using Functional Data structures

5.1 Induction Graphs Haskell Implementation

In his original paper, Erwig proposes Haskell code to implement common graph algorithms such as DFS, BFS, and minimum spanning tree on inductive graphs via a Haskell library he created [Erw01]. While max flow was not an originally proposed graph algorithm that Erwig demonstrated, his Haskell library now supports two separate implementations of maximum flow on inductive graphs [EM22].

The main implementation is a standard Edmond-Karp implementation utilizing a functional paradigm. The core of the algorithm is depicted below in Haskell code of figure 13.

To breakdown the logic of this program, we start with the inputs. We have that gr is a graph object where we assume the edge labels are of the form (max capacity, current capacity, residual capacity), and residual capacity represents the weight of the edge in the residual graph. Thus, notice that this approach is storing the residual

```
mfmg :: (DynGraph gr, Num b, Ord b) => gr a (b,b,b) -> Node -> Node -> gr
    a (b,b,b)
mfmg g s t
  | null augPath = g
  | otherwise   = mfmg (updateFlow augPath minC g) s t
  where
    minC       = minimum (map ((\(_,_,z)->z).snd)(tail augLPath))
    augPath    = map fst augLPath
    LP augLPath = lesp s t gf
    gf         = elfilter (\(_,_,z)->z/=0) g
```

Figure 13: Haskell Induction Graph Max-Flow [EM22]

graph and the original graph within the same object to avoid duplicating edges and increasing storage space. Next, we have our source and sink nodes s and t.

Then, with all of this information, the logical flow of the algorithm is very straight forward. We set the base case to be no augmenting path exists, and recursively update the graph with augmenting paths while they do exist. Notice that this function stays true to the functional paradigm and does not directly update the object with which it is computing since objects are immutable in this setting, but rather returns a new version of the object with the updated information.

While the Haskell code itself for this implementation can be intimidating to those unfamiliar with the language, the logical flow that the program presents is very succinct and directly in line with that for the Ford-Fulkerson method. The details of the implementation are within the code below the *where* line, when the program explicitly states how to compute the specific values involved in the computation, such as getting the augmenting path via BFS between the source and the sink in the residual graph.

Notice that the user facing code for working with the inductive graphs is not affected by the functional data structure. That is, similar to the example with union find, the specific implementations that guarantee us the immutability and therefore persistence of these data structures is abstracted away from user facing code allowing the programmer to reason at a high level without worrying about low-level issues [CF07].

5.2 Algebraic Graphs for Haskell

There also exists an extensive library for the use of Mokhov's algebraic graphs within Haskell. This library extends the ideas of algebraic graphs presented in the paper to allow for the implementation of networks, or weighted graphs. This extension is done by changing how the connect operation works within the algebra to have it also take

in a weight or label parameter [Mok]. That is, we now formally define the connect operation for integer labeled edges as,

$$C : G^* \times G^* \times \mathbb{Z} \to G^*\text{: a connect operation such that for graphs } G_1 = (V_1, E_1) \text{ and } G_2 = (V_2, E_2) \text{ as well as a weight or label parameter } w, \text{ we have } C(G_1, G_2) = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(v_1, v_2) : v_1 \in V_1, v_2 \in V_2\}) \text{ where the weight or label on each new edge created is } w.$$

This simple extension is what allows us to have weighted networks in Haskell. Therefore, since this library also includes BFS, it is clear that this package contains the necessary tools to build a maximum flow implementation.

Conclusion

We have seen that the max-flow problem has been efficiently implemented using functional data structures for the graph representation. Moreover, the induction graph implementation as well as the persistent union find representation exemplify how strong functional data structure properties can be guaranteed without adversely affecting user-facing code. Specifically, persistence can be guaranteed without blurring away the logic of the problem at hand. In the union-find example, it was the auxiliary function in the data structure implementation that guaranteed us persistence without compromising on optimally or usability.

Similarly, we reviewed the technique of monads and how they allow the abstraction of data structure logic to avoid disrupting the logical flow of the problem. All of these techniques form an arsenal of powerful tools that can be used in data structure implementation to allow the user-facing code to focus strictly on the logical flow of the problem and avoid handling low level details or optimizations. In other words, these are tools of abstraction in the field of functional programming that assist in the ease of reasoning provided.

As was seen in the max-flow example, classical algorithms can be implemented using these functional data structures to allow for the ease of reasonability of recursive design principles to become present in user-facing code without compromising on the efficiency of such implementations under the hood. Once the building block of BFS and DFS can be defined on the data structure, we can utilize them to implement many classical algorithms on functional data structures. In that sense, the traversal primitive is tied to the data structure and is what promotes the abstraction from the data structures.

To conclude, this paper focuses on the abstraction provided by techniques in designing functional data structures. However, we do not deeply address the specific optimizations made to avoid compromising on efficiency when choosing functional data structures. Many of the functional data structures presented in the paper include creative solutions to improving efficiency that were not explicitly investigated. As

an example for future work, consider Erwig's active pattern matching scheme to define traversals on induction graphs and how it can be optimized to give us a linear traversal overall. Digging deeper into this question will require more analysis tools such as exploring amortized bounds and more advanced memoization techniques. Therefore, further research into these techniques and their application on functional data structures pose a promising area of future investigation.

References

[AAHJUJ+83] V Aho Alfred, E Hopcroft John, D Ullman Jeffrey, V Aho Alfred, H Bracht Glenn, D Hopkin Kenneth, C Stanley Julian, Brachu Jean-Pierre, Brown A Samler, Brown A Peter, et al. *Data structures and algorithms*. USA: Addison-Wesley, 1983.

[Aro22] Samuel Aronson. Understanding functional and equational programming techniques on graph-based problems, 2022.

[CF07] Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 37–46, 2007.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[Din06] Yefim Dinitz. Dinitz'algorithm: The original version and even's version. In *Theoretical computer science*, pages 218–240. Springer, 2006.

[DSST89] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989.

[EK72] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[EM22] Martin Erwig and Ivan Lazar Miljenovic. Fgl haskell code, Sep 2022.

[Erw01] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[FF56] Lester Randolph Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

[Gor22] Jacob Gorenburg. Cyclic structures and lazy evaluation, 2022.

[has] Haskellwiki: All about monads.

[Mok] Andrey Mokhov. Alga haskell library.

[Mok17] Andrey Mokhov. Algebraic graphs with class (functional pearl). *ACM SIGPLAN Notices*, 52(10):2–13, 2017.

[Oka96] Chris Okasaki. Functional data structures. In *International School on Advanced Functional Programming*, pages 131–158. Springer, 1996.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 61–78, New York, NY, USA, 1990. Association for Computing Machinery.