FUNCTIONAL GRAPH ALGORITHMS

Pulkit Agarwal (200050113)

May 2022

Contents

1 Introduction

One of the major attractions that functional programming paradigms offer is the ability to write concise and elegant code. It handles various data types, such as lists and trees, quite efficiently by using inductive definitions. As a simple example, here is a possible way to define a tree.

```
data Gtree a = Node a [Gtree a]
```

On the other hand, the same is not so easy for general graphs. The problem arises due to cycles in the graph. Unlike trees, which have the concept of leaves and root, there is no such start and end for a general graph. So any recursive definition will revisit old vertices on discovering a cycle. The first part of this report discusses the implementation of inductive graphs suggested by Martin Erwig in [1] and a few advantages as well as shortcomings of this approach.

2 Why Inductive Definitions?

There has been a lot of research prior to the work in [1], which tries to implement general graphs using functional programming. In particular, there are 2 major issues that need to be handled. The first one is to overcome the problem that cycles can create by a robust definition for the graph data type. To clarify, the problem is not in making a graph data structure (you could simply use a list of lists as an adjacency list type structure). Instead, we want to use the functional style to represent the graph (and the reason to do so is in the second problem). The second is to make the data type in such a manner that it is possible to implement different graph algorithms (such as DFS), and also to do so efficiently, i.e. with a time complexity comparable with the imperative programming style.

2.1 Dealing with cycles

For the first problem, a possible approach is to exploit the lazy nature of `Haskell`. In particular, note that the issue is that we can run into infinite loops due to cycles. So if the laziness could be incorporated in a way that the infinite structure is never fully calculated, then we will be able to get a representation for graphs.

A method to do so is the "tying the knot" technique. Think of a cycle like a thread that has been knotted, which is unrolled lazily by the `Haskell` compiler. Having mutually recursive definitions (say by using the let or where clause), can help ensure that the recursion is unrolled lazily. Consider the following directed graph given in [3]:

```
data Vertex a b = Vertex { vdata :: a, edges :: [Edge a b] }
data Edge   a b = Edge   { edata :: b, src :: Vertex a b, dst :: Vertex a b }

type Myvertex = Vertex String ()
type Myedge   = Edge   String ()

e :: Myvertex -> Myvertex -> Myedge
e = Edge ()

v :: String -> [Myedge] -> Myvertex
v = Vertex

mygraph5 = map vv [ "one", "two", "three", "four", "five" ] where
    vv s = let vk = v s (zipWith e (repeat vk) mygraph5) in vk
```

Here we have a graph, with vertices labeled by strings (namely `"one"`, `"two"`, and so on), and having unlabeled edges. `e` and `v` are functions that form an edge between 2 vertices, and a vertex given its label and adjacent edges, respectively. The final graph mygraph5 consists of a complete graph of size 5.

Notice that the definition of the map function `vv` in mygraph5 involves the statement

```
let vk = v s (zipWith e (repeat vk) mygraph5) in vk
```

In each unrolling of the let clause, the function `v` is called with arguments `s` (i.e. the label currently being mapped, such as `"one"`) and `(zipWith e (repeat vk) mygraph5)`. The `zipWith` function only zips the two lists (with function `e`) till one of them gets exhausted (note that the first list `repeat vk` is infinite, while the second one is finite). Due to this lazy evaluation, in each step of the map function, a vertex gets created with edges to all other vertices, and the cycles in the graph do not cause an issue.

However, this approach fails to give efficient algorithms, as discussed in the next section.

2.2 DFS Algorithm

The traditional DFS algorithm, implemented in various imperative and object oriented languages, requires maintaining a state for each of the vertices, which informs the algorithm about whether the vertex has been visited before or not. The problem with any extension of this algorithm to a functional program is that maintaining this state in functional set data structure is no longer a constant time operation, since there is no way to access an element at a particular index of a list in constant time. This is why the "tying the knot" approach fails (or any trivial adjacency list or adjacency matrix representation fails). The PhD. Thesis of Daniel Jonathan King in [6] addresses this disparity. Consider the following implementation of the dfs algorithm:

```
dfs :: Graph -> [Vertex] -> [Vertex] -> ([Tree Vertex], [Vertex])
dfs g ms [] = ([], ms)
dfs g ms (v: vs) = if v `elem` ms
                   then dfs g ms vs
                   else let (ts, as) = dfs g (v: ms) (g! v)
                            (us, bs) = dfs g as vs
                        in (Node v ts: us, bs)
```

Assume that the Graph data structure has been defined in some manner. The notation `g! v` gives all the vertices adjacent to `v` in graph `g`, which requires doing lookup in a list. As lookup is a linear time operation, so the overall time complexity of this approach is $O(|V|(|V| + |E|))$ instead of the usual $O(|V| + |E|)$. King has used state monads to implement this lookup step in constant time, but this makes the program much more complex as well as closer to an imperative definition. The work of Erwig tries to keep the code as clean as possible by making a new way to define the Graph data type.

3 Inductive Graphs

As described in the previous section, recursive definitions similar in taste to those for trees could make the process of dealing with graphs in functional programming much less painful. Erwig has defined the Graph data type as follows (a full implementation is available in `fgl` library, which uses similar definitions).

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

data Graph a b = Empty | Context a b & Graph a b
```

Here, `Graph a b` represents a multi-graph with labeled vertices of type `a` and labeled edges of type `b`. To make a graph, we inductively add vertices to it, with the initial graph being `Empty`. In each addition, the added Context consists of all edges coming into the vertex from previously added vertices, and all edges going out of it to these vertices. In particular, although the above term representation in itself is insufficient, it works under the invariant that every node in the `Context` of a newly inserted node is already present in the graph.

Although, the actual implementation in the library is abstract, the above algebraic interpretation can help prove some important properties. As an example, Erwig proves in [2] that the above data type is complete, or equivalently it is able to represent all possible labeled multi-graphs. Here is a quick sketch of his proof to this theorem.

Theorem. *Each labeled multi-graph can be represented by a graph term.*

Proof. The main point is that we can first add all vertices, and then add all edges with the help of pattern matching. Addition of node/edge can be done using the following auxiliary functions:

```
addnode (v,l) g = ([],v,l,[]) & g

addedge (v,w) ((p,u,l,s) & g) =
    if u = v then (p,u,l,w:s) & g
             else (p,u,l,s) & (addedge (v,w) g)
```

□

A stronger version of the above completeness theorem says that we can in fact represent a graph `g` with the starting `Context` having any arbitrary node `v` of `g`. This also shows that the representation of a graph is not unique (which is expected since it depends on how we choose our sequence of vertices while adding inductively).

3.1 Pattern Matching

The above definition allows us to make various functions by basic pattern matching. As an example, the author extends the `map` function to `gmap` function on the graph (note that, although technically not a list, the graph data type does act like a list to some extent). Here, `gmap` applies a function `f` on each `Context` in the graph. So, for example, the function which interchanges the in-edges and out-edges in a `Context` can be used to implement a function `grev` to reverse the edges in a graph.

Erwig also shows the benefit of the functional paradigm by proving different properties, such as the fusion property for `gmap`, which states that `gmap f.gmap f' = gmap(f.f')`. However, the actual implementation of these functions are slightly different. In particular, since & is no longer a constructor, but a function in the `fgl` library, so we can not simply do pattern matching. Instead he defines a function `matchAny` which extracts out any random `Context` from a graph. Although this will not cause an issue in the `grev` function (which does not depend on the order of `Context`), but some other interpretation of `gmap` can give wrong answers if it has a dependence on the order in which the `Context` come.

3.2 Active Pattern Matching

To handle the problem described above, Erwig uses the concept of Active Patterns, which he himself defined in [5]. The basic idea is to get a form of pattern matching on the abstract graph implementation in `fgl` which is sure to have the obtained element as the first element (i.e. the matched `Context` is the first one). Unfortunately, active patterns are not available in `Haskell`, and so currently functions are written by explicitly searching for the `Context` of a given node `v`.

3.3 Depth First Search

Erwig has given implementations of various algorithms in [1]. We will look at one of them, which is DFS. Again instead of the constant time active pattern matching $\&^v$ operation, the actual implementation is done via case matching for the `Context`. The algorithm works as follows:

```
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, the algorithm returns the order in which the vertices are visited, and takes as input list of vertices still left to be visited and the graph on which search is taking place. In particular, if `v` is not present in this graph, then we simply apply `dfs` for the remaining vertices. Else we add `v` to the the depth-first order output, and apply `dfs` on the graph after removing the `Context c` corresponding to `v`, and adding the successors of `v` to the start of the list of nodes left to be visited.

Note that, we are simply removing any node that we visit from the graph, thus ensuring that the successors are all unvisited. Also, adding the successors at the start causes the search to be depth-based. To make this into `bfs`, all that needs to be done is add the successors at the end (however this change would not give an efficient `bfs` algorithm since the append operation is linear time in the first argument; to get efficiency one would need to use some constant-time queue implementation).

4 Going Forward?

Although pathbreaking in its approach, the actual implementation of inductive graphs in the `fgl` library do have some shortcomings. Doczkal addresses some of them in [7]. Due to the transformative nature of active pattern matching, it is not possible to implement these inductive graphs as an algebraic data type. So the package uses balanced binary search trees to actually represent graphs. However, although reasonably efficient for small and medium size graphs, the actual implemented algorithms do not meet imperative time complexity. To actually reach this complexity, one would either need to perform multi-threaded operations, or work within a monad, which destroys the functional flavor being suggested by Erwig.

A second issue is discussed in [8]. This time, it covers the fact that the theoretical representation itself has some holes, namely the dependence on the graph invariant. As an example, consider the following code:

```
gErr :: Gr Char Char
gErr = mkGraph [(1,'a')] [(1,2,'p')]
```

Here, we are forming an edge between nodes with indices 1 and 2, even though there is no node with index 2. Ideally, this should produce a compile-time error, but this code compiles correctly. [8] handles such problems by defining a new data type, namely inductive triple graphs. Here, they remove the need for an index for each node, and instead require that each node and edge have a unique label (although the label can be same for a node and an edge). The representation for this triple graph is

```
type TContext a = (a, [(a,a)], [(a,a)], [(a,a)])

data TGraph a = Empty | TContext a & TGraph a
```

Here, `TContext` consists of the predecessors, successors and relations for each node/edge, where relations is the two endpoints if the label is for an edge. The authors have also used this representation to implement different graph algorithms, and making functions similar to those in `fgl`.

This problem of partial functions in `fgl`, i.e. issues with inserting an edge with one of the vertices missing, is also addressed by Andrey Mokhov in [4]. He defines algebraic graphs and provides a clean methodology to work with them, based on rigorous mathematical algebra of programs.

5 Algebraic Graphs

Algebraic graphs are defined for graphs with labeled vertices and unlabeled edges, and it has the following representation:

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

where `Empty` and `Vertex a` are graphs with no node, and a single node, while `Overlay` and `Connect` are composing functions. Formally, we have

`Overlay` (V_1, E_1) (V_2, E_2) = $(V_1 \cup V_2, E_1 \cup E_2)$

`Connect` (V_1, E_1) (V_2, E_2) = $(V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)$

The crucial property of this data type is that non-graphs are not representable by it, while every graph is. We also note that `Connect` is the only source of edge insertions.

To show completeness of this method (i.e. all graphs are representable), note that we can make the set of vertices as a `Graph` using `overlay` on all `Vertex` occurences. Similarly, the set of edges can also be formed as a `Graph`, by using `Connect` on the corresponding vertices, and then combining them using `Overlay`. Finally, we can take a union of these 2 `Graph` instances, which forms the complete graph.

The advantage of this representation is that it gives fewer opportunities for usage errors. However, as of now it lacks methods to efficiently implement graph algorithms, and also does not support labeled edges.

6 References

For the full list of references, please take a look at *references.txt*. The below list consists of references which are mentioned in the report.

1. Erwig, M. (2001) *Inductive Graphs and Functional Graph Algorithms*
2. Erwig, M. (1997) *Functional Programming with Graphs*
3. Tying The Knot - Stack Overflow
4. Mokhov, A. (2017) *Algebraic Graphs with Class (Functional Pearl)*
5. Erwig, M. (1995) *Active Patterns*
6. King, D. J. (1996) *Functional Programming and Graph Algorithms*
7. Doczkal, C. (2005) *A Functional Graph Library*
8. Gayo, J. E. L. (2014) *Inductive Triple Graphs*