Project Report

Liziqiu Yang

August 1, 2023

Contents

1 Overview

In this report, we choose [1] for implementation. In this paper, the authors propose a linear structure for graph type constructors and some algorithms on graphs, using such a linear structure can simplify formal proofs on the one hand, and on the other hand, it can be elegantly implemented in functional programming to realize some generic graph algorithms, and this structure can also instantiate some type classes on graphs, such as functor, monad and so on. On the other hand, it can be used to implement algorithms in functional programming in a very elegant way, and it can also be used to instantiate type classes on graphs, such as functor, monad, etc. In this paper, we present the results of our work on graph types. In this paper, we generalize the structure mentioned in the paper to support graphs with arbitrary nodes, and we analyze the benefits of using this structure to implement algorithms.

2 Implementation

2.1 Type definition

In this paper, we analyze [1], which describes a graph structure and graph type definitions that are friendly to proofs and functional programming, and the types of graphs defined by the authors in the paper are as follows.

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

For this type, the author most important is the introduction of Context for the definition, which Node is simply defined as Int type, where Context type represents the node's parent and child nodes, and Adj's b represents the attributes on the edge (such as the edge's weight, the edge's name and so on). We extend Node to facilitate some algorithmic operations later.

```
data Node a = Node {  name   :: String,
                      value  :: a
                   }  deriving(Show)

instance (Eq a) => Eq (Node a) where
    (Node n1 v1) == (Node n2 v2) = (n1 == n2) && (v1 == v2)
```

Then you can define the Graph, the general Graph is an unordered structure, because the graph itself is discrete and does not have a definite order, and given the Context can be given to a definite order of the graph, we define the type as follows.

```
data Graph  a b = Empty | Cons (Context a b) (Graph a b) deriving(Show)
(&) :: Context a b -> Graph a b -> Graph a b
(&) x y = Cons x y
```

Observe that the above structure is actually very similar to the definition of a List, which is defined as follows.

```
data List a = Empty | Cons a (List a) deriving(Show)
```

Comparing the definitions of Graph and List, we can find that the author defines the type of graph as a chain structure, which transforms the original unordered graph into a linear structure.

And storing the graph as an adjacency list requires much less space than an adjacency matrix, and the adjacency list allows us to access the children of a node in O(1) time complexity instead of requiring $\Omega(n)$ time complexity to scan.

2.2 Some functional graph functions

Using this graph definition, we can implement some additional functions as follows.

```
gmap :: (Context a b -> Context c d) -> Graph a b ->   Graph c d
gmap f Empty = Empty
gmap f (Cons c g) = (f c) & gmap f g

grev :: Graph a b -> Graph a b
grev = gmap swap where swap (u,n,v) = (v,n,u)

ufold :: (Context a b -> c -> c) -> c -> Graph a b -> c
ufold f u Empty = u
ufold f u (Cons c g) = f c (ufold f u g)
```

In the above function we have implemented **gmap**,**grev**,**ufold**. gmap maps a function to each Context in the graph, which is similar to map on a List, and since each Context contains a Node, it can also be processed for nodes, and the same can be done for edges. grev flips the directed graph, and $grev \circ grev = id$. ufold is similar to lfold and requires an initial value to be defined.

2.3 Functor

Earlier we defined gmap on Graph, since the definition of this Graph is similar to the chained definition of List, we can also instantiate Functor on this class, for this we need to check Functor Law.

Proof. **Case 1** fmap id = id, for any graph g,we have:

$$\begin{aligned} \text{fmap id g} &= && \text{fmap id (c \& g)} \\ &= && \text{c \& fmap id g} \\ &\dots \\ &= && \text{g} \end{aligned}$$

□

By the above proof fmap id = id holds.

Next we need to show that fmap (f . h) == fmap f . fmap h holds.

Proof. **Case 2**

$$\begin{aligned} \text{fmap (f.h) g} &= && \text{fmap (f.h) (c \& g)} \\ &= && \text{(f.h) c \& fmap (f.h) g} \end{aligned}$$

Next we consider fmap f . fmap h g, (h c) & g = fmap h g ,then consider fmap f (h c) & g = fmap f . h c & g ,so it follows that fmap (f.h) == fmap f . fmap h holds.

□

Similarly, if we think of Graph as a chain of lists, and Context as an element in the chain, we can show that Graph is in fact a singleton.

3 Algorithm

For the Algorithm section, in the code given by the original author we can see a clean implementation of dfs and bfs.

```
dfs :: Graph gr => [Node] -> gr a b -> [Node]
dfs [] _            = []
dfs _ g | isEmpty g = []
dfs (v:vs) g        = case match v g of
                        (Just c, g') -> v:dfs (suc c ++ vs) g'
                        (Nothing, _) -> dfs vs g

bfs :: Graph gr => [Node] -> gr a b -> [Node]
bfs [] _            = []
bfs _ g | isEmpty g = []
bfs (v:vs) g        = case match v g of
                        (Just c, g') -> v:bfs (vs ++ suc c) g'
                        (Nothing, _) -> bfs vs g
```

The original authors implemented the DFS (Depth-First Search) and BFS (Breadth-First Search) algorithms in very short code by using pattern matching and recursion. These concise implementations maintain the efficiency of the algorithms as they still have the same time complexity

The same shortest-path related algorithms can still be implemented in a very simple way, even if the algorithms themselves are complex.

```
dijkstra :: (Real b, Graph gr) => H.Heap b (LPath b) -> gr a b -> LRTree b
dijkstra h g | H.isEmpty h || isEmpty g = []
dijkstra h g =
  case match v g of
    (Just c, g')  -> p:dijkstra (H.mergeAll (h':expand d p c)) g'
    (Nothing, g') -> dijkstra h' g'
  where (_, p@((v, d):_), h') = H.splitMin h
```

In particular, the Dijkstra algorithm finds the shortest path by continuously selecting the r-path of the node with the smallest label, then adding its successor to the front of the path and updating the label. The algorithm uses a heap data structure to manage the nodes to keep the node with the smallest label in front. During the execution of the algorithm, each node will be selected only once as they are labeled as permanent nodes and the labeled nodes are ignored in subsequent selections. For nodes in the heap, storing different root paths does not affect correctness because the algorithm only selects the root path with the smallest cost. Overall, this version of Dijkstra's algorithm runs in O(m log m), where m is the number of edges, i.e., the size of the graph. This is because the time complexity of the heap operation is $O(logm)$and there are at most m nodes in the heap. In the worst case, when the graph is a dense graph, the algorithm runs in $O(m + mlogn)$, where n is the number of nodes and m is the number of edges. In dense graphs, the number of edges m can be close to n^2, at which point heap operations become the main time consumer. In a sparse graph, the number of edges m is much smaller than n^2, and the running time of the algorithm is mainly determined by the heap operation, so it can be approximated as $O(mlogn)$.

The author has also implemented the prim algorithm for minimum spanning trees and the topological sorting algorithm, as follows, which is still very concise.

```
topsort :: Graph gr => gr a b -> [Node]
topsort g = reverse . concatMap postorder . dff (nodes g) $ g
```

We summarize a comparison of the time complexity of the algorithms implemented by the authors with the time complexity of the imperative language, as shown in the following table.

programming paradigm	BFS	DFS	Dijkstra	Prim
functional programming	$O(n+m)$	$O(n+m)$	$O(m+mlog(n))$	$O(mlog(n)$
Command Programming	$O(n+m)$	$O(n+m)$	$O(m+nlog(n))$	$O(mlog(n)$

As we can see from the table above, using the defined Inductive graph for the algorithm to implement the graph not only guarantees correctness but also code elegance while maintaining the time complexity of the algorithm and even improving it a bit.

4 Summary

Implementing graph algorithms using inductive graph structures not only ensures correctness but also produces elegant code. In addition, these algorithms maintain time complexity and even improve

compared to their counterparts in imperative languages. The functional programming paradigm allows us to effectively utilize pattern matching, recursion, and higher-order functions, which improves the elegance and efficiency of the implementation. This report emphasizes the advantages of functional programming for graph algorithms and the importance of using appropriate data structures to achieve correctness and efficiency. By adopting this linear structure for graph types, functional programming enthusiasts can further explore the appeal of functional graph algorithms and their applications in various fields.

[2]

References

[1] M. Erwig, "Inductive graphs and functional graph algorithms," *J. Funct. Program.*, vol. 11, no. 5, p. 467–492, sep 2001. [Online]. Available: https://doi.org/10.1017/S0956796801004075

[2] "GitHub - sevanspowell/inductive-graphs: A simple implementation of the inductive graphs detailed in Martin Erwig's paper "Inductive Graphs and Functional Graph Algorithms" (https://web.engr.oregonstate.edu/ er $--$ *github.com*," , [Accessed 01-08-2023].