Review: Graph Algorithms with a Functional Flavour

Yicong Luo
yicong.luo@epfl.ch

Qingyi He
qingyi.he@epfl.ch

November 30, 2022

1 Introduction

Prior research on pure functional graph algorithms has been limited, but graph algorithms have a wide range of applications. This work represents the graph in a purely functional way. And it focuses mostly on depth-first search since it can be utilized to construct a number of effective graph algorithms.

The approach of this work managed to achieve standard complexity as the imperative way does while maintaining a high level of modularity. As commonly used in other purely functional algorithms, it applies the idea to graph algorithms, of separating the algorithms into several phases and connecting them using explicit intermediate values.

2 Preliminaries

Commonly, there are three ways to represent a directed graph. The first one is the most functional way, by using cyclic structures. Another one is highly imperative, it describes the graph as explicit mutable nodes, which is convenient when there is a need to change the graph. And the last one is used by this paper, it represents a graph as an immutable array of edges, namely the adjacency list. This approach is at the border of imperative programming and functional programming and can be used in both paradigms.

2.1 Adjacency List

The adjacency list can be noted as an immutable array, indexed by the vertices. Each element of this list is a list of vertices, which are adjacent to the vertex, the index of this entry. Obviously, the size of an adjacency list is linear with the size of the graph.

Then, a graph can be represented as follows.

```haskell
type Table a = Array Vertex a
type Graph   = Table [Vertex]
```

While the name of each vertex is not important, we can simply assume they are named contiguously in an ordered type (e.g. NAT). Then, the list of vertices can be represented by the starting point and the ending point:

```haskell
type Bounds = (Vertex, Vertex)
```

This paper defined a generic function mapT, which applies a function to each table and generates a new table. It consume a function for a single vertex and convert its table to another one:

```haskell
mapT :: (Vertex -> a -> b) -> Table a -> Table b
mapT f t = array (bounds t) [(v, f v (t!v) | v <- indices t]
```

And a function transfers a list of vertices to a look-up table, which provides their index.

```haskell
tabulate :: Bounds -> [Vertex] -> Table Int
tabulate bnds vs = array bnds (zip vs [1..])
```

It also provides two functions to convert between the adjacency list and a list of all the edges this graph contains. This conversion can be accomplished in linear time.

```haskell
type VE = (Bounds, [(Vertex, Vertex)])

edges :: Graph -> VE
edges g = (bounds g, [(v, w) | v <- indices g, w <- g!v])

buildG :: VE -> Graph
buildG (bnds, es) = accumArray snoc [] bnds es
    where snoc xs x = x:xs
```

3 Body

3.1 Simple Operations on Graph

With those given predefined functions, we can apply simple Operations to the graphs.

3.2 Deep-First Search

The following procedure can be used to intuitively describe the deep-first search. Mark each node as "unvisited" to begin with, then pick one origin node to begin the exploration. Every time, we'll aim to hop over to one of the nodes that are right next to the one we're at right now but haven't yet been to. If every neighboring node is marked as visited, we go back and carry on exploring at the node we just left.

If every node that is reachable from the chosen source node has been checked out, we choose another unvisited node to continue the search. This process terminates when all the nodes of this graph are visited.

3.2.1 Specification in the purely functional way

In a purely functional way, the depth-first search should be seen as a value, not a process as in the imperative view. This paper chooses to represent a depth-first search as a spanning forest, a particular sub-graph of the given graph, which contains all the vertices and the edges that we visited when searching.

Then the edges of the original graph can be classified into the following sets. The tree edges are included in the spanning forest. The back edges go in the opposite direction to the tree edges. The forward edges, go from lower nodes to deeper nodes in the same spanning tree. The cross edge connects vertices across the forest.

The forest can be represented as follows:

```
data Tree a = Node a (Forest a)
type Forest a = [Tree a]
```

And the depth-first search converts a graph to a forest, as:

```
dfs :: Graph -> [Vertex] -> Forest Vertex

dff :: Graph -> Forest Vertex
dff g = dfs g (indices g)
```

3.3 Implementing Depth-First Search

In the transition from a graph to a spanning tree, this paper first generates and then prunes, as in most of the other lazy functional programming.

Given a graph and a set of root vertices, it first generates a forest that contains all the vertices and edges of the graph and then prunes the overlapping part.

3.3.1 Generating

The function for generating is defined as below, given a graph g and the root set v, builds a tree rooted at v.

```
generate :: Graph -> Vertex -> Tree Vertex
generate g v = Node v (map (generate g) (g!v))
```

3.3.2 Pruning

The function pruning will eliminate the repeated subtrees. It maintains a finite set that mimics the imperative technique of maintaining a boolean array to record the status.

The set is represented as a mutable array.

```
type Set s = MutArr s Vertex Bool

mkEmpty :: Bounds -> ST s (Set s)
mkEmpty bnds = newArr bnds False

contains :: Set s -> Vertex -> ST s Bool
contains m v = readArr m v

include :: Set s -> Vertex -> ST s ()
include m v = writeArr m v True
```

Then prune function is defined as follows and its final result is the value generated by chop.

```
prune :: Bounds -> Forest Vertex -> Forest Vertex
prune bnds ts = runST (mkEmpty bnds CthenST' \m ->
                        chop m is)
```

```
chop :: Set s -> Forest Vertex -> ST s (Forest Vertex)
chop m [] = returnST []
chop m (Node v ts : us)
    = contains m v 'thenST ~ \visited ->
        if visited then
            chop m us
        else
            include m v 'thenST' \_ ->
            chop m ts 'thenST' \as ->
            chop m Us CthenST, \bs ->
            returnST ((Node v as) : bs)
```

And then, the dfs function is defined by combining those two parts together.

```
dfs g vs = prune (bounds g) (map (generate g) vs)
```

4 Conclusion

This paper investigates several graph search algorithms that attain standard complexity while vastly improving on conventional imperative presentations. It builds the algorithms specifically from reusable parts, offering a higher level of modularity than is customary elsewhere. Additionally, it gives instances of soundness proofs that are considerably different from conventional proofs, primarily because they are based on reasoning about a static value rather than the dynamic process of graph traversal.

It gives us a way to implement the graph algorithm in a purely functional way, without any increase in complexity. It allows us to verify the graph algorithms also.