Structuring Depth-First Search Algorithms in Haskell

Xiaoyu Tongyang ✉

December 2, 2021

1 Introduction

In this paper, I will introduce the way to implement Depth-First Search in Haskell with state transformers. And I will compare the implementation in Haskell with that in a imperative language, like my Java implementation. Therefore, this paper structures as follows: In Section 2, the way to represent graphs is discussed. In Section 3, the way to construct DFS is introduced. Finally, in Section 4, two DFS-wise algorithms are presented.

2 Representing graphs

2.1 Graphs

Traditionally, we represent a graph using two ways: 1) an adjacency list or 2) an adjacency matrix. The first one, in general, requires less store space but to take longer time to find an adjacent vertex than adjacency matrix, while adjacency matrix can require an adjacent vertex at constant time, O(1), but may need more store space when dealing with a sparse graph. So in this paper, we represent a graph though an adjacency list with a standard Haskell immutable array from Data.Array model,

In this way, we know the relationship between a vertex and its adjacent ones, and then get the whole graph. In imperative language, we represent a graph with an adjacency list usually like this:

```
// Class Graph
public class Graph<E extends Node> {
    public final List<E> vertices = new ArrayList<>();

    // other code
}

// Class Vertex
public class Vertex extends Node {
    // adjacent list
    public final List<Vertex> neighbours = new ArrayList<>();

    // other code
}
```

Much like what we do in a imperative language, we construct a graph as a table indexed by vertices, where the table is an immutable array and the type Vertex may be any type belonging to the Haskell index class **Ix**:

```
-- Graph.hs
import Data.Array -- inculding Data.Ix

--The type Vertex may be any type belonging to the Haskell
   index class Ix
-- ( Ix i ) =>
```

```
type Vertex i = i

-- Graphs, therefore, may be thought of as a table indexed by
   vertices.
type Table i a = Array ( Vertex i ) a
type Graph i = Table ( Vertex i ) [ Vertex i ]
```

And vertices of a graph is just the indices of an array, so we provide vertices as an alternative for **indices**:

```
{-
 - We provide vertices as an alternative for indices, which
    returns a list of all the vertices in a graph.
 -
 - visibility: public
 -}

vertices :: ( Ix i ) => Graph i -> [ Vertex i ]
vertices = indices
```

An edge is a pair of vertices, one of which is an endpoint of the edge and the other is another endpoint. In a imperative languange:

```
public class Edge implements Comparable<Edge> {
    // the starting vertex and the ending vertex
    public final Vertex startVertex;
    public final Vertex endVertex;
}
```

In Haskell:

```
-- An edge is a pair of vertices.
type Edge i = ( Vertex i, Vertex i )

{-
-- it is convenient to extract a list of edges from the graph
 -
 - visibility: public
 -}

edges :: ( Ix i ) => Graph i -> [ Edge i ]
edges g = [ ( v, w ) | v <- vertices g, w <- g ! v ]
```

edges can extract a list of edges from the graph. To count outdegree and indegree for a vertex, we need to define a generic function **mapT** which applies its function argument to every table index entry pair, and builds a new table.

```
{-
 -To manipulate tables (including graphs) we provide a generic
    function mapT
 - which applies its function argument to every table index
    fentry pair, and builds a new table.
 -
 - visibility: public
 -}

```

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

We define type **Bounds** as a convenient way to feed a Haskell function **array** with low and ight bounds, and with **mapT**, we can define a function, **outdegree**, to detail the number of edges leaving each vertex.

```
-- Because we are using an array-based implementation
-- we often need to provide a pair of vertices as array bounds.
    So for convenience we define,
type Bounds i = ( Vertex i, Vertex i )

{-
 - builds a table detailing the number of edges leaving each
    vertex.
 -
 - visibility: public
 -}

outdegree :: ( Ix i ) => Graph i -> Table i Int
outdegree g = mapT numEdges g
    where numEdges v ws = length ws
```

the function **buildG** builds up a graph from a list of edges.

```
{-
 - To build up a graph from a list of edges
 -
 - visibility: public
 -}

buildG :: ( Ix i ) => Bounds i -> [ Edge i ] -> Graph i
buildG bnds es = accumArray ( flip ( : ) ) [] bnds es
```

And we can reverse each pair of edge and call **buildG** to build the transpose of the graph, where the direction of every edge is reversed.

```
{-
 - Combining the functions edges and buildG gives us a
 - way to reverse all the edges in a graph giving the transpose
 - of the graph:
 -
 - visibility: public
 -}

transpose :: ( Ix i ) => Graph i -> Graph i
transpose g = buildG ( bounds g ) ( reverseE g )

{-
 - We extract theedges from the original graph, reverse their
 - direction, and rebuild agraph with the new edges.
 -
 - visibility: public
 -}

```

```
reverseE :: ( Ix i ) =>  Graph i -> [ Edge i ]
reverseE g = [ ( w, v ) | ( v, w ) <- edges g ]
```

This means we can compute indegree of a vertex with the combination of **transpose** and **outdgree**.

```
{-
 - Now by using trensposeG
 - diately define an indegree table for vertices:
 -
 - visibility: public
 -}

indegree :: ( Ix i ) => Graph i -> Table i Int
indegree g = outdegree ( transpose g )
```

2.2 Example

Then, to make what we've talked about a real sense, let's go over an example to illustrate more on those. Assume we have a directed graph:

```
graph = buildG ('a', 'j')
               [ ('a','j'), ('a','g'), ('b','i'),
                  ('b','a'), ('c','h'), ('c','e'),
                  ('e','j'), ('e','h'), ('e','d'),
                  ('f','i'), ('g','f'), ('g','b') ]
```

which is shown in Figure 1.

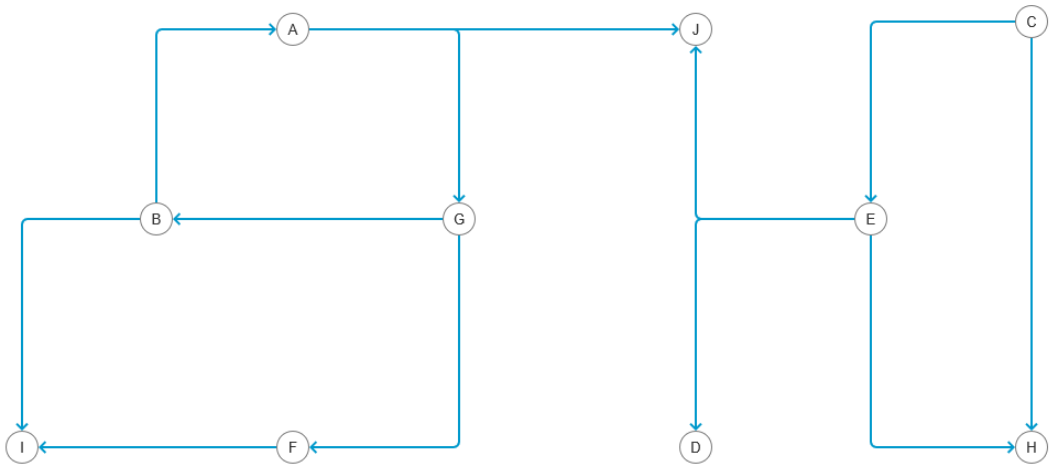

Figure 1: A directed graph

First, we want to get all vertices and edges of the graph:

```
ghci> graph = buildG ('a', 'j') [('a','j'),('a','g'),('b','i'),
    ('b','a'),('c','h'),('c','e'), ('e','j'),('e','h'),('e','d
   '), ('f','i'),('g','f'),('g','b')]
ghci> vertices graph
"abcdefghij"
ghci> edges graph
[('a','g'),('a','j'),('b','a'),('b','i'),('c','e'),('c','h'),('
   e','d'),('e','h'),('e','j'),('f','i'),('g','b'),('g','f')]
```

We got all the vertices in an array and all the edges as a pair of two vertices connected each other. Second, we want to see outdegree and indegree for every vertex in the graph:

```
ghci> outdegree graph
array ('a','j') [('a',2),('b',2),('c',2),('d',0),('e',3),('f
    ',1),('g',2),('h',0),('i',0),('j',0)]
ghci> indegree graph
array ('a','j') [('a',1),('b',1),('c',0),('d',1),('e',1),('f
    ',1),('g',1),('h',2),('i',2),('j',2)]
```

So, as the output shown above, vertex 'a' has outdegree 2, which is correct, it has two outgoing edges, connecting vertices J and G. Also, it has indegree 1, which is the incoming edge connecting vertex B.

3 Depth-first search

3.1 Specification of DFS

After successfully constructing a graph, it's time to do DFS. Traditionally, depth-first search is as a process which may loosely be described as follows. Initially, all the vertices of the graph are deemed "unvisited", so we choose one and explore an edge leading to a new vertex. Now we start at this vertex and explore an edge leading to another new vertex. We continue in this fashion until we reach a vertex which has no edges leading to unvisited vertices. At this point we backtrack, and continue from the latest vertex which does lead to new unvisited vertice[1]. For example, we implement this idea in Java like:

```
/**
 * DFS to find all reachable vertices and count their number
 * */

public int DFS( Vertex vertex, boolean[] visited ) {
    // base case
    // visited before?
    if ( visited[ vertex.ID ] ) return 0;
    visited[ vertex.ID ] = true;

    // recursion procedure
    int totalVertices = 0;
    for ( Vertex neighbour : vertex.neighbours )
        totalVertices += DFS( neighbour, visited );

    return totalVertices + 1;
}
```

We use a boolean array to mark if a vertex has been visited before, and start at a vertex and recursively visit their neighbors until all vertices are visited. However, in haskell implementation, we will take advantage of the concept, the spanning forest defined by a depth-first traversal of a graph, to do so, like this one in Figure 2:

The (solid) tree edges are those graph edges which lead to unvisited vertices. The remaining graph edges are also shown, but in dashed lines. These edges are classified according to their relationship with the tree, namely, forward edges (which connect ancestors in the tree to descendants), like CH, back edges (the reverse), like BA, and cross edges (which connect nodes across the forest, but always from right to left), like FI[1].

Next, we define data structures for DFS, that is, **Tree** and **Forest**. A forest is a list of trees, and a tree is a node containing some value, together with a forest of sub-trees. Both trees and

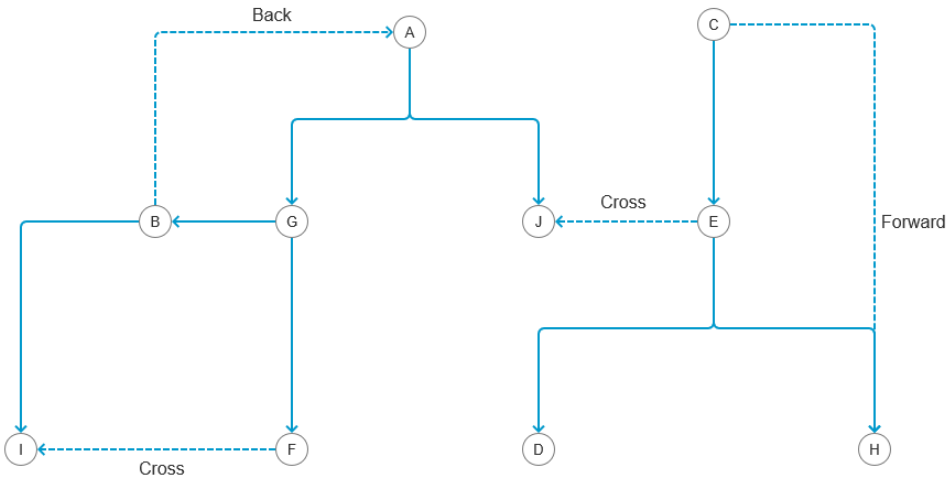

Figure 2: A depth-first fortest of the graph

forests are polymorphic in the type of data they may contain[1].

```haskell
-- A forest is a list of trees, and a tree is a node containing
    some value, together with a forest of sub-trees.
data Tree a = Node a ( Forest a ) deriving Show
type Forest a = [ Tree a ]
```

Notice, to make easy to represent Tree or Node in textual context, I will use the following format: *vertex's name: [its children]*. For example, for Vertex E in the graph, it should be: E: [H: []], which means vertex E has one child, H and H has no children.

3.2 Implementing DFS

The overall idea to translate a graph into a DFS spanning tree is first to generate spanning tree starting with a vertex (maybe infinite if the graph is cyclic) , and then prune duplicate subtree. We define a function **generate** which, given a graph g and a vertex v builds a tree rooted at v containing all the vertices in g reachable from v[1].

For pruning, we take advantage of state transformers and and mimic the imperative technique of maintaining an array of booleans, indexed by vertices. To do this, we need to implement a data type **Set** to have the ability above:

```haskell
-- ( Ix i ) =>
type Set s i = STUArray s ( Vertex i ) Bool

{-
 - array to mark visiting status for each vertex
 -
 - visibility: private
 -}

mkEmpty :: ( Ix i ) => Bounds i -> ST s ( Set s i )
mkEmpty bnds = newArray bnds False

{-
 - check the visiting status of v
 -
 - visibility: private
```

```
 -}

contains :: ( Ix i ) => Set s i -> Vertex i -> ST s Bool
contains m v = readArray m v

{-
 - mark v as visited
 -
 - visibility: private
 -}

include :: ( Ix i ) => Set s i -> Vertex i -> ST s ()
include m v = writeArray m v True
```

To make our life easier, you may think this set is similar to the boolean array in imperative language, like Java:

```
// type Set s i
boolean[] set = new boolean[ 10 ];

// mkEmpty
// all elements in the array are false by default

// contains
boolean status = set[ 0 ];

// include
set[ 0 ] = true;
```

But the array in haskell can mimic the array in imperative language but not lose nice property in haskell, like purity. With these, we define **prune** as follows:

```
{-
 - The prune function begins by introducing a fresh state
    thread,
 - then generates an empty set within that thread and calls
 - chop. The final result ofprune is the value generatedby
 - chop, the final state being discarded.
 -
 - visibility: private
 -}

prune :: ( Ix i ) => Bounds i -> Forest ( Vertex i ) -> Forest
   ( Vertex i )
prune bnds ts = runST $ do
    marks <-  mkEmpty bnds
    chop marks ts
```

The prune function begins by introducing a fresh state thread, then generates an empty set within that thread and calls chop. The final result ofprune is the value generated by **chop**, the final state being discarded[1].

```
{-
 - When chopping a list of trees, the root of the first is
    examined.
 - If it has occurred before, the whole tree is discarded.
 - If not, thevertex isaddedto the set represented bym, and
```

```
 - two further calls tochop aremade in sequence.
 - The first, namely, chop m ts, prunes the forest of
    descendants
 - of v, adding all these to the set of marked vertices.
 - Once this incomplete, thepruned subforest is named
 - as, andtheremainder of the original forest is chopped. The
 - 349 result of this is, in turn, named bs, and the resulting
    forest
 - is constructed from the two.
 -
 - visibility: private
 -}

chop :: ( Ix i ) => Set s i -> Forest ( Vertex i ) -> ST s (
   Forest ( Vertex i ) )
chop m [] = return []
chop m ( ( Node v ts ) : us ) = do
    visited <- contains m v
    if visited then
        chop m us
    else do
        include m v
        as <- chop m ts
        bs <- chop m us
        return ( ( Node v as ) : bs )
```

When chopping a list of trees, the root of the first is examined. If it has occurred before, the whole tree is discarded. If not, the vertex is added to the set represented by m, and two further calls to chop are made in sequence[1].

3.3 Examples

To make the two functions more understandable, I will talk through two examples to illustrate more on this.

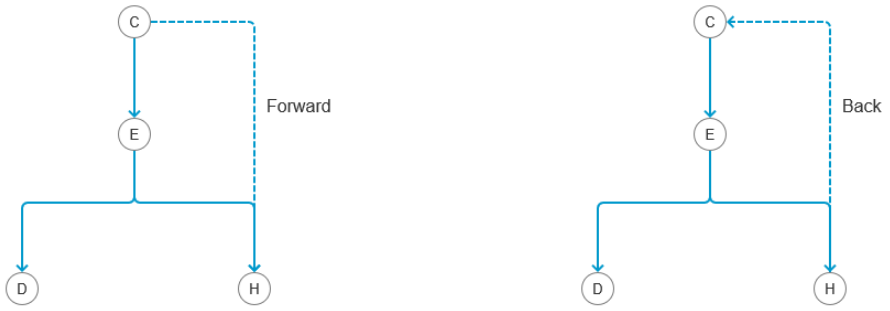

Figure 3: Two examples of generating and pruning

In the example on the left, assume we start with vertex C, we will generate this:

```
C: [ E: [ D: [], H: [], ]
         H:[] ]
```

After pruning, this structure won't change since no cycles in it. In other example, starting with vertex C as well, we'll generate this:

```
C: [ E: [ D: [], H: [ C: [ ... ] ], ]  ]
```

After pruning, the duplicate node C in H will be removed:

```
C: [ E: [ D: [], H: [], ]  ]
```

3.4 DFS

At this point, we have all components to provide the definition of DFS.

```
{-
 - A depth-first search of a graph takes a graph and an initial
 - ordering of vertices. Allgraph vertices inthe initial
    ordering
 - will be in the returned forest.
 -
 - visibility: public
 -}

dfs :: ( Ix i ) => Graph i -> [ Vertex i ] -> Forest ( Vertex i
    )
dfs g vs = prune ( bounds g ) ( map ( generate g ) vs )


{-
 - Sometimes the initial ordering of vertices is not important.
 -
 - visibility: public
 -}

dff :: ( Ix i ) => Graph i -> Forest ( Vertex i )
dff g = dfs g ( vertices g )
```

The argument vs is a list of vertices, so the generate function is mapped across this (having been given the graph g). The resulting forest is pruned in a left-most top-most fashion by prune[1]. If the initial ordering of vertices does not matter, we use **dff** instead of **dfs**.

4 DFS Algorithms

In this chapter, I'll introduce two algorithms utilizing DFS implemented above.

4.1 Algorithm 1. Depth-first search numbering

The first one is Depth-first search numbering, which is very important since many other algorithms are based upon it.

We can express depth-first ordering of a graph g most simply by flattening the depth-first forest in preorder. Preorder on trees and forests places ancestors before descendants and left subtrees before right subtrees[1].

```
{-
 - We can express depth-first ordering of a graph g most
 - simply by flattening the depth-first forest in preorder.
    Preorder
 - on trees and forests places ancestors before descendants
 - and left subtrees before right subtrees4:
```

```
 -
 - visibility: private
 -}

preorder :: Tree a -> [ a ]
preorder ( Node a ts ) = [ a ] ++ preorderF ts

{-
 - entry function to call preorder
 -
 - visibility: private
 -}

preorderF :: Forest a -> [ a ]
preorderF ts = concat ( map preorder ts )

{-
 - obtaining a list of vertices in depth-first order
 -
 - visibility: public
 -}

preOrd :: ( Ix i ) => Graph i -> [ Vertex i ]
preOrd g = preorderF ( dff g )
```

For the example above, we can test this and get:

```
ghci> graph = buildG ('a', 'd') [('a','b'),('b','c'),('b','d')
    ,('a','d')]
ghci> graph
array ('a','d') [('a',"db"),('b',"dc"),('c',""),('d',"")]
ghci> ( preOrd graph )
"adbc"
```

However, it is often convenient to translate such an ordered list into actual numbers. For this we could use the function **tabulate**[1].

```
{-
 - However, it is often convenient to translate such an ordered
 - list into actual numbers. This zips the vertices together
     with the positive integers
 - 1, 2, 3,..., and (in linear time) builds an array of these
 - numbers, indexed by the vertices.
 -
 - visibility: public
 -}

tabulate :: ( Ix i ) => Bounds i -> [ Vertex i ] -> Table (
    Vertex i ) Integer
tabulate bnds vs = array bnds ( zip vs [ 1.. ] )

{-
 - (it turns out to be convenient for later algorithms if such
 - functions take the depth-first forest as an argument, rather
 - than construct the forest themselves.)
 -
 - visibility: public
```

```
 -}

preArr :: ( Ix i ) => Bounds i -> Forest ( Vertex i ) -> Table
    ( Vertex i ) Integer
preArr bnds ts = tabulate bnds ( preorderF ts )
```

Also, we can test it with the same example:

```
ghci> tabulate (bounds graph) ( preOrd graph )
array ('a','d') [('a',1),('b',3),('c',4),('d',2)]
```

As we can see, the order of visiting vertices by DFS is $a \to d \to b \to c$, which is consistent with the result.

4.2 Algorithm 2. Finding reachable vertices

Finding all the vertices that are reachable from a single vertex v demonstrates that the dfs doesn't have to take all the vertices as its second argument. Commencing a search at v will construct a tree containing all of v's reachable vertices. We then flatten this with preorder to produce the desired list[1].

```
{-
 - Finding all the vertices that are reachable from a single
     vertex
 - v demonstrates that the dfs 'doesnt have to take all the
 - vertices as its second argument. Commencing a search at v
 - will construct a tree containing all of 'vs reachable
     vertices.
 - We then flatten this with preorder to produce the desired
 - list.
 -
 - visibility: public
 -}

reachable :: ( Ix i ) => Graph i -> Vertex i -> [ Vertex i ]
reachable g v = preorderF ( dfs g [ v ] )

{-
 - One application of this algorithm is to test for the
     existence
 - of a path between two vertices:
 -
 - The elem test is lazy: it returns True as soon as a match
 - is found. Thus the result of reachable is demanded lazily,
 - and so only produced lazily. As soon as the required vertex
 - is found the generation of the DFS forest ceaaes. Thus
 - dfs implements a true search and not merely a complete
 - traversal.
 -
 - visibility: public
 -}

path :: ( Ix i ) => Graph i -> Vertex i -> Vertex i -> Bool
path g v w = w elem ( reachable g v )
```

Also, we can test it with the same example above:

```
ghci> reachable graph 'a'
"adbc"
ghci> path graph 'a' 'c'
True
ghci> path graph 'a' 'h'
False
```

As we can see, all reachable vertices from vertex a is a, d, b and c, and path from vertex a to c is reachable, but the path from vertex a to h is not.

References

[1] David J. King and John Launchbury. Structuring depth-first search algorithms in haskell. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 344–354, New York, NY, USA, 1995. Association for Computing Machinery.