Structuring Depth-First Search Algorithms in Haskell

David J. King

Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, UK
gnik@dcs.gla.ac.uk

John Launchbury

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
Beaverton, Oregon 97006–1999, USA
jl@cse.ogi.edu

Abstract

Depth-first search is the key to a wide variety of graph algorithms. In this paper we express depth-first search in a lazy functional language, obtaining a linear-time implementation. Unlike traditional imperative presentations, we use the structuring methods of functional languages to construct algorithms from individual reusable components. This style of algorithm construction turns out to be quite amenable to formal proof, which we exemplify through a calculational-style proof of a far from obvious strongly-connected components algorithm.

Classifications: Computing Paradigms (functional programming); Environments, Implementations, and Experience (programming, graph algorithms).

1 Introduction

The importance of depth-first search (DFS) for graph algorithms was established twenty years ago by Tarjan (1972) and Hopcroft and Tarjan (1973) in their seminal work. They demonstrated how depth-first search could be used to construct a variety of efficient graph algorithms. In practice, this is done by embedding code-fragments necessary for a particular algorithm into a DFS procedure skeleton in order to compute relevant information while the search proceeds. While this is quite elegant it has a number of drawbacks. Firstly, the DFS code becomes intertwined with the code for the particular algorithm, resulting in monolithic programs. The code is not built by re-use, and there is no separation between logically distinct phases. Secondly, in order to reason about such DFS algorithms we have to reason about a dynamic *process*—what happens and when—and such reasoning is complex.

Occasionally, the *depth-first forest* is introduced in order to provide a *static* value to aid reasoning. We build on this idea. If having an explicit depth-first forest is good for reasoning then, so long as the overheads are not unacceptable, it is good for programming. In this paper, we present a wide variety of DFS algorithms as combinations of standard components, passing explicit intermediate values from one to the other. The result is quite different from traditional presentations of these algorithms, and we obtain a greater degree of modularity than is usually seen.

Of course, the idea of splitting algorithms into many separate phases connected by intermediate data structures is not new. To some extent it occurs in all programming paradigms, and is especially common in functional languages. What is new, however, is applying the idea to graph algorithms. The difficulty is always to find a sufficiently flexible intermediate value which allows a wide variety of algorithms to be expressed in terms of it.

There is another challenge here, however. Graph algorithms have long been poorly handled in functional languages. It has not been at all clear how to express such algorithms without using side effects to achieve efficiency, and lazy languages by their nature have to prohibit side-effects. So, for example, many texts provide implementations of search algorithms which are quadratic in the size of the graph (see Paulson (1991), Holyer (1991), or Harrison (1993)), compared with the standard linear implementations given for imperative languages (see Manber (1989), or Corman *et al.* (1990)).

In our work there is one place where we do need to use destructive update in order to gain the same complexity (within a constant factor) as imperative graph algorithms. We make use of recent advances in lazy functional languages which use monads to provide updatable state, as implemented within the Glasgow Haskell compiler. The compiler provides extensions to the language Haskell (Hudak *et al.* 1992) providing updatable arrays (Launchbury and Peyton Jones 1994), and allows these state-based actions to be encapsulated so that their external behaviour is purely functional. Consequently we obtain linear algorithms and yet retain the ability to perform purely functional reasoning on all but one fixed and reusable component.

Most of the methods in this paper apply equally to strict and lazy languages. The exception is in the case when DFS is being used for a true *search* rather than for a complete *traversal* of the graph. In this case, the co-routining behaviour of lazy evaluation allows the search to abort early without needing to add additional mechanisms like exceptions. We give an example of this in Section 6.

In summary the main contributions of this paper are:

- We provide implementations of DFS algorithms in linear time in Haskell. We are careful to provide real code throughout, and avoid resorting to pseudo-code.
- We construct the algorithms using reusable components, providing a greater level of modularity than is typical in other presentations.
- We provide examples of correctness proofs. Again, these are quite different from traditional proofs, largely because they are not based upon reasoning about the dynamic *process* of DFS, but rather about a static *value*.

This paper is organised as follows. Section 2 introduces a data type for graphs and some standard functions which will be used in subsequent algorithms. Section 3 introduces depth-first search. Section 4 describes the Haskell implementation of several algorithms that use depth-first search which includes: topological sorting, strongly connected components, as well as others. Section 5 describes the linear implementation of depth-first search in Haskell. Section 6 describes some more complex algorithms that use depth-first search, including edge classification and biconnected components. Section 7 discusses the complexity of the algorithms. Finally, Section 8 discusses related work.

2 Representing graphs

In order to meet our goal of not resorting to pseudo-code, we need to begin with some boring details. There are many ways to represent (directed) graphs. For our purposes, we use an array of adjacency lists. The array is indexed by vertices, and each component of the array is a list of those vertices reachable along a single edge. This adjacency structure is linear in the size of the graph, that is, the sum of the number of vertices and the number of edges. By using an indexed structure we are able to be explicit about the sharing that occurs in the graph. Another alternative would have been to use a recursive tree structure and rely on cycles within the heap. However, the sharing of nodes in the graph would then be implicit making a number of tasks harder.

So we will just use a standard Haskell immutable array. This gives constant time access (but not update—these arrays may be shared arbitrarily).

We can use the same mechanism to represent *undirected* graphs as well, simply by ensuring that we have edges in both directions. An undirected graph is a symmetric directed graph. We could also represent *multi-edged* graphs by a simple extension, but will not consider them here.

Graphs, therefore, may be thought of as a table indexed by vertices.

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

The type `Vertex` may be any type belonging to the Haskell index class `Ix`, which includes `Int`, `Char`, tuples of indices, and more. Haskell arrays come with indexing (`!`) and the functions `indices` (returning a list of the indices) and `bounds` (returning a pair of the least and greatest indices). We provide `vertices` as an alternative for `indices`, which returns a list of all the vertices in a graph.

```
vertices :: Graph -> [Vertex]
vertices = indices
```

Sometimes it is convenient to extract a list of edges from the graph, this is done with the function `edges`. An edge is a pair of vertices.

```
type Edge = (Vertex,Vertex)

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

To manipulate tables (including graphs) we provide a generic function `mapT` which applies its function argument to every table index/entry pair, and builds a new table.

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

The Haskell function `array` takes low and high bounds and a list of index/value pairs[1], and builds the corresponding array in linear time. 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 = (Vertex,Vertex)
```

Using `mapT` we could define,

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

which builds a table detailing the number of edges leaving each vertex.

To build up a graph from a list of edges we define `buildG`.

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

Like `array` the Haskell function `accumArray` builds an array from a list of index/value pairs, with the difference that

[1] Actually Haskell uses the `Assoc` type, which is equivalent, but introduces an unnecessary new notation.

accumArray accepts possibly many values for each indexed location, which are combined using the function provided as accumArray's first argument. Here we simply build lists of all the values associated with each index. Again, constructing the array takes linear time with respect to the length of the adjacency list. So in linear time, we can convert a graph defined in terms of edges to the vertex table based graph. For example,

```
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')]
```

will produce the array representation for the graph shown in Figure 1.

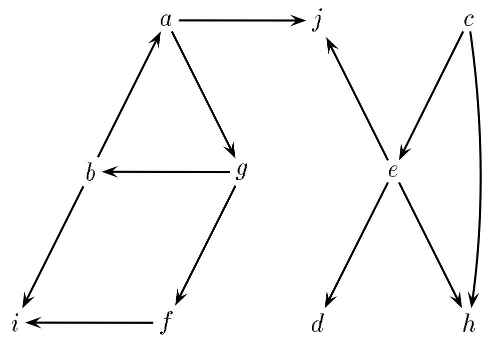


Figure 1: A directed graph

Then, to find the immediate successors to 'e', say, we compute:

```
graph ! 'e'
```

which returns ['d', 'h', 'j'].

Combining the functions edges and buildG gives us a way to reverse all the edges in a graph giving the *transpose* of the graph:

```
transposeG :: Graph -> Graph
transposeG g = buildG (bounds g) (reverseE g)

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

We extract the edges from the original graph, reverse their direction, and rebuild a graph with the new edges. Then, for example,

```
(transposeG graph) ! 'e'
```

will return ['c']. Now by using transposeG we can immediately define an indegree table for vertices:

```
indegree :: Graph -> Table Int
indegree g = outdegree (transposeG g)
```

This example gives a first feel for the approach we advocate in this paper. Rather than defining indegree from scratch by, for example, building an array incrementally as we traverse the graph, we simply reuse previously defined functions, combining them in a fresh way. The result is shorter and clearer, though potentially more expensive (an intermediate array is constructed). There are two things to say about this additional cost. Firstly, the additional cost only introduces a constant factor into the complexity measure, so the essence of the algorithm is preserved. Secondly, recent work in the automatic removal of intermediate structures (deforestation) comes a long way to removing this problem.

3 Depth-first search

The traditional view of 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 vertices.

Eventually we will reach a point where every vertex reachable from the initial vertex has been visited. If there are any unvisited vertices left, we choose one and begin the search again, until finally every vertex has been visited once, and every edge has been examined.

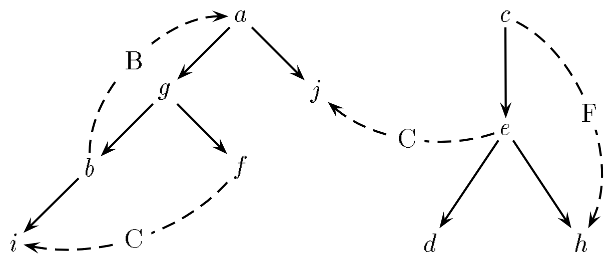


Figure 2: A depth-first forest of the graph

In this paper we will concentrate on depth first search as a specification for a *value*, namely the *spanning forest* defined by a depth-first traversal of a graph. Such a forest for the graph in Figure 1 is depicted 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), *back edges* (the reverse), and *cross edges* (which connect nodes across the forest, but always from right to left). This standard classification is useful for thinking about a number of algorithms and later, in Section 6, we give an algorithm for classifying edges in this way.

3.1 Specification of depth-first search

As the approach to DFS algorithms which we explore in this paper is to manipulate the depth-first forest *explicitly*, the first step, therefore, is to construct the depth-first forest from a graph. To do this we need an appropriate definition of trees and forests.

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

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

A depth-first search of a graph takes a graph and an initial ordering of vertices. All graph vertices in the initial ordering will be in the returned forest.

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

This function is the pivot of this paper. For now we restrict ourselves to considering its properties, and will leave its Haskell implementation until Section 5.

Sometimes the initial ordering of vertices is not important. When this is the case we use the related function

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

What are the properties of depth-first forests? They can be completely characterised by the following two conditions.

(i) The depth-first forest of a graph is a spanning subgraph, that is, it has the same vertex set, but the edge set is a subset of the graph edge set.

(ii) The graph contains no left-right cross-edges with respect to the forest.

Later on in the paper, we find it convenient to talk in terms of *paths* rather than single edges: a path being made up of zero or more edges joined end to end. We will write $v \longrightarrow_{\texttt{g}} w$ to mean that there is a path from v to w in the graph `g`. Where there will be no confusion we will drop the graph subscript.

The ban on left-right cross edges translates into paths. At the top level, it implies that there is no path from any vertex in one tree to any vertex in a tree that occurs later in the forest. Thus[2],

Property 1
If (`ts++us=dff g`), then $\forall v \in \texttt{ts} \,.\, \forall w \in \texttt{us} \,.\, v \not\longrightarrow w$ $\square$

Deeper within each tree of the forest, there *can* be paths which traverse a tree from left to right, but the absence of any graph edges which cross the tree structure from left to right implies that the path has to follow the tree structure. That is:

Property 2
If the tree (`Node x (ts++us)`) is a subtree occurring anywhere within `dff g`, then

$$\forall v \in \texttt{ts} \,.\, \forall w \in \texttt{us} \,.\, v \longrightarrow w \Rightarrow v \longrightarrow x$$

$\square$

So the only way to get from v to w is via (an ancestor of) x, the point at which the forests that contain v and w are combined (otherwise there would be a left-right cross edge). Thus there is also a path from v to x.

The last property we pick out focusses on `dfs`, and provides a relationship between the initial order, and the structure of the forest[3].

Property 3
Let a and b be any two vertices. Write $\longrightarrow$ for edges in the graph `g`, and $\leq$ for the ordering induced by the list of vertices `vs`. Then

$$\begin{array}{ll} & \exists t \in \texttt{dfs g vs} \,.\, a \in t \wedge b \in t \\ \Leftrightarrow & \exists c \,.\, c \longrightarrow a \;\wedge\; c \longrightarrow b \;\wedge \\ & \quad (\forall d \,.\, d \longrightarrow a \;\vee\; d \longrightarrow b \;\Rightarrow\; c \leq d) \end{array}$$

$\square$

This Property says that:

- $\Rightarrow$ given two vertices which occur within a single depth-first tree (taken from the forest), then there is a predecessor of both (with respect to $\longrightarrow$) which occurs earlier in `vs` than any other predecessor of either. (If this were not the case, then a and b would end up in different trees).
- $\Leftarrow$ if the earliest predecessor of either a or b is a predecessor of them both, then they will end up in the same tree (rooted by this predecessor).

These three properties are certainly true of DFS spanning forests, but we make no claim about their completeness. There are other useful properties not derivable from these.

4 Depth-first search algorithms

Algorithm 1. Depth-first search numbering

Having specified DFS (at least partly) we turn to consider how it may be used. The first algorithm is straightforward. We wish to assign to each vertex a number which indicates

[2] We use the notation `ts++us` to indicate any division of the list of trees in the forest, such that the order of the trees is preserved. Note that either `ts` or `us` could be empty. Also, we use $\in$ to indicate list membership and not purely for set membership.

[3] We further overload the $\in$ notation, to mean that both a and b occur within the tree t.

where that vertex came in the search. A number of other algorithms make use of this *depth-first search number*, including the biconnected components algorithm that appears later, for example.

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[4]:

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

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

Now obtaining a list of vertices in depth-first order is easy:

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

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

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

which 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.

We can package these up into a function as follows:

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

(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.)

Algorithm 2. Topological sorting

The dual to preorder is postorder, and unsurprisingly this turns out to be useful in its own right. Postorder places descendants before ancestors and left subtrees before right subtrees:

```
postorder :: Tree a -> [a]
postorder (Node a ts) = postorderF ts ++ [a]


postorderF :: Forest a -> [a]
postorderF ts = concat (map postorder ts)
```

So, like with preorder, we define,

```
postOrd :: Graph -> [Vertex]
postOrd g = postorderF (dff g)
```

[4] The use of repeated appends (`++`) caused by `concat` introduces an extra logarithmic factor here, but this is easily removed using standard transformations.

The lack of left-right cross edges in DFS forests leads to a pleasant property when a DFS forest is flattened in postorder. To express this we need a definition.

Definition
A linear ordering $\leq$ on vertices is a *post-ordering* with respect to a graph `g` exactly when,

$$v \leq w \;\wedge\; v \longrightarrow w \;\Rightarrow\; \exists u \,.\, v \longleftrightarrow u \;\wedge\; w \leq u$$

□

(where $v \longleftrightarrow u$ means $v \longrightarrow u$ and $u \longrightarrow v$). In words, this definition states that, if from some vertex v there is a path to a vertex later in the ordering, then there is also a vertex u which occurs no earlier than w and which, like w is also reachable by a path from v. In addition, however, there is also a path from u to v.

This property is so-named because post order flattening of depth first forests have this property.

Theorem 4
If `vs=postOrd g`, then the order in which the vertices appear in `vs` is a post-ordering with respect to `g`.

Proof
If v comes before w in a post order flattening of a forest, then either w is an ancestor of v, or w is to the right of v in the forest. In the first case, take w as u. For the second, note that as $v \longrightarrow w$, by Property 1, v and w cannot be in different trees of the forest. Then by Property 2, the lowest common ancestor of v and w will do. □

We can apply all this to topological sorting. A topological sort is an arrangement of the vertices of a directed acyclic graph into a linear sequence $v_1, \ldots, v_n$ such that there are no edges from v_j to v_i where $i < j$. This problem arises quite frequently, where a set of tasks need to be scheduled, such that every task can only be performed after the tasks it depends on are performed.

We define,

```
topSort :: Graph -> [Vertex]
topSort g = reverse (postOrd g)
```

Why is this correct? If w comes before v in the result of `topSort g`, then v comes before w in the result of `postOrd g`. Thus, by Theorem 4, there exists a vertex u no earlier than w which is in a cycle with v. But, by assumption, the graph is acyclic, so no such edge $v \longrightarrow$ exists.

Algorithm 3. Connected components

Two vertices in an undirected graph are *connected* if there is a path from the one to the other. In a directed graph, two vertices are connected if they would be connected in the graph made by viewing each edge as undirected. Finally, with an undirected graph, each tree in the depth-first spanning forest will contain exactly those vertices which constitute a single component.

We can translate this directly into a program. The function `components` takes a graph and produces a forest, where each tree represents a connected component.

```
components :: Graph -> Forest Vertex
components g = dff (undirected g)
```

where a graph is made undirected by:

```
undirected :: Graph -> Graph
undirected g = buildG (bounds g)
                      (edges g ++ reverseE g)
```

The undirected graph we actually search may have duplicate edges, but this has no effect on the structure of the components.

Algorithm 4. Strongly connected components

Two vertices in a directed graph are said to be *strongly connected* if each is reachable from the other. A strongly connected component is a maximal subgraph, where all the vertices are strongly connected with each other. This problem is well known to compiler writers as the dependency analysis problem—separating procedures/functions into mutually recursive groups. We implement the double depth-first search algorithm of Kosaraju (unpublished), and Sharir (1981).

```
scc :: Graph -> Forest Vertex
scc g = dfs (transposeG g) (reverse (postOrd g))
```

The vertices of a graph are ordered using `postOrd`. The reverse of this ordering is used as the initial vertex order for a depth-first traversal on the transpose of the graph. The result is a forest, where each tree constitutes a single strongly connected component.

The algorithm is simply stated, but its correctness is not at all obvious. However, it may be proved as follows.

Theorem 5
Let a and b be any two vertices of `g`. Then

$$(\exists t \in \texttt{scc g} \,.\, a \in t \wedge b \in t) \;\Leftrightarrow\; a \longleftrightarrow b$$

Proof
The proof proceeds by calculation. We write $\texttt{g}^{\mathrm{T}}$ for the transpose of `g`. Edges $\longrightarrow$ in `g` will be edges $\longleftarrow$ in $\texttt{g}^{\mathrm{T}}$. Further, let $\leq$ be the post-ordering defined by `postOrd g`. Then its reversal induces the ordering $\geq$. Now,

$$\begin{array}{ll}
 & \exists t \in \texttt{scc g} \,.\, a \in t \wedge b \in t \\
\Leftrightarrow & \{ \text{ Definition of } \texttt{scc} \} \\
 & \exists t \in \texttt{dfs } \texttt{g}^{\mathrm{T}} \texttt{ (reverse (postOrd g))} \,.\, a, b \in t \\
\Leftrightarrow & \{ \text{ By Property 3 } \} \\
 & \exists c \,.\, c \longleftarrow a \;\wedge\; c \longleftarrow b \;\wedge \\
 & \quad (\forall d \,.\, d \longleftarrow a \;\vee\; d \longleftarrow b \;\Rightarrow\; c \geq d) \\
\Leftrightarrow & \exists c \,.\, a \longrightarrow c \;\wedge\; b \longrightarrow c \;\wedge \\
 & \quad (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c)
\end{array}$$

From here on we construct a loop of implications.

$$\begin{array}{ll}
 & \exists c \,.\, a \longrightarrow c \;\wedge\; b \longrightarrow c \;\wedge \\
 & \quad (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c) \\
\Rightarrow & \{ \text{ Consider } d = a \text{ and } d = b \;\} \\
 & \exists c \,.\, a \longrightarrow c \;\wedge\; a \leq c \;\wedge\; b \longrightarrow c \;\wedge\; b \leq c \;\wedge \\
 & \quad (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c) \\
\Rightarrow & \{ \leq \text{ is a post-ordering } \} \\
 & \exists c \,.\, (\exists e \,.\, a \longleftrightarrow e \;\wedge\; c \leq e) \;\wedge \\
 & \quad (\exists f \,.\, b \longleftrightarrow f \;\wedge\; c \leq f) \;\wedge \\
 & \quad (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c) \\
\Rightarrow & \{ e = c \text{ and } f = c \text{ using } (\forall d \ldots) \} \\
 & \exists c \,.\, a \longleftrightarrow c \;\wedge\; b \longleftrightarrow c \\
\Rightarrow & \{ \text{ Transitivity } \} \\
 & a \longleftrightarrow b
\end{array}$$

which gives us one direction. But to complete the loop:

$$\begin{array}{ll}
 & a \longleftrightarrow b \\
\Rightarrow & \{ \text{ There is a latest vertex reachable from } a \text{ or } b \} \\
 & a \longleftrightarrow b \;\wedge\; \exists c \,.\, (a \longrightarrow c \;\vee\; b \longrightarrow c) \;\wedge \\
 & (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c) \\
\Rightarrow & \{ \text{ Transitivity of } \longrightarrow \} \\
 & \exists c \,.\, a \longrightarrow c \;\wedge\; b \longrightarrow c \;\wedge \\
 & (\forall d \,.\, a \longrightarrow d \;\vee\; b \longrightarrow d \;\Rightarrow\; d \leq c)
\end{array}$$

as required, and so the theorem is proved. □

To the best of our knowledge, this is the first calculational proof of this algorithm. Traditional proofs (see Corman *et al.* (1990), for example) typically take many pages of wordy argument. In contrast, because we are reusing an earlier algorithm, we are able to reuse its properties also, and so obtain a compact proof. Similarly, we believe that it is because we are using the DFS forest as the basis of our program that our proofs are simplified as they are proofs about values rather than about processes.

A minor variation on this algorithm is to reverse the roles of the original and transposed graphs:

```
scc' :: Graph -> Forest Vertex
scc' g = dfs g (reverse (postOrd (transposeG g)))
```

The advantage now is that not only does the result express the strongly connected components, but it is also a valid depth-first forest for the original graph (rather than for the transposed graph). This alternative works as the strongly connected components in a graph are the same as the strongly connected components in the transpose of the graph.

5 Implementing depth-first search

In order to translate a graph into a depth-first spanning tree we make use of a technique common in lazy functional programming: generate then prune. Given a graph and a list of vertices (a root set), we first generate a (potentially infinite) forest consisting of all the vertices and edges in the graph, and then prune this forest in order to remove repeats. The choice of pruning pattern determines whether the forest ends up being depth-first (traverse in a left-most, top-most fashion) or breadth-first (top-most, left-most), or perhaps some combination of the two.

5.1 Generating

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`.

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

Unless `g` happens to be a tree anyway, the generated tree will contain repeated subtrees. Further, if `g` is cyclic, the generated tree will be infinite (though rational).

Of course, as the tree is generated on demand, only a finite portion will be generated. The parts that `prune` discards will never be constructed.

5.2 Pruning

The goal of pruning the (infinite) forest is to discard subtrees whose roots have occurred previously. Thus we need to maintain a set of vertices (traditionally called "marks") of those vertices to be discarded. The set-operations we require are initialisation (the empty set), membership test, and addition of a singleton. While we are prepared to spend linear time in generating the empty set (as it is only done once), it is essential that the other operations may be performed in constant time.

The easiest way to achieve this is to make use of *state transformers*, and mimic the imperative technique of maintaining an array of booleans, indexed by the set elements. This is what we do. We provide an explanation of state-transformers in the Appendix, but as they have already been described in a number of papers (Moggi 1989, Wadler 1990, Peyton Jones and Wadler 1993, Launchbury 1993, Launchbury and Peyton Jones 1994), and already been implemented in more than one Haskell variant, we avoid cluttering the main text.

The implementation of vertex sets is easy:

```
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
```

Using these, we define `prune` as follows.

```
prune :: Bounds -> Forest Vertex -> Forest Vertex
prune bnds ts
  = runST (mkEmpty bnds `thenST` \m ->
           chop m 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 of `prune` is the value generated by `chop`, the final state being discarded.

```
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         `thenST` \bs ->
      returnST ((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.

The first, namely, `chop m ts`, prunes the forest of descendants of `v`, adding all these to the set of marked vertices. Once this is complete, the pruned subforest is named `as`, and the remainder of the original forest is chopped. The result of this is, in turn, named `bs`, and the resulting forest is constructed from the two.

All this is done lazily, on demand. The state combinators force the computation to follow a predetermined linear sequence, but exactly where in that sequence the computation is, is determined by external demand. Thus if only the top-most left-most vertex were demanded then that is all that would be produced. On the other hand, if only the final tree of the forest is demanded, then because the set of marks is single-threaded, all the previous trees will be produced. However, this is demanded by the very nature of DFS anyway, so it is not as restrictive as it may at first seem.

At this point one may wonder whether any benefit has been gained by using a functional language. After all, the code looks fairly imperative. To some extent such a comment would be justified, but it is important to note that this is the *only* place in the development that destructive operations have to be used to gain efficiency. We have the flexibility to gain the best of both worlds: where destructive update is vital we use it, where it is not vital we use the powerful modularity options provided by lazy functional languages.

5.3 DFS

The components of generate and prune are combined to provide the definition of DFS.

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

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.

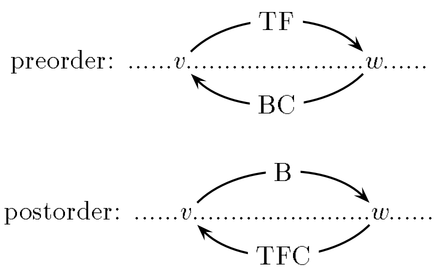
If paying an extra logarithmic factor is acceptable, then it is possible to dispense completely with the imperative features used in `prune`, and to use an implementation of sets based upon balanced trees, for example.

6 More algorithms

Algorithm 5. Classifying edges

We have already seen the value of classifying the graph edges with respect to a given depth-first search. Here we codify the idea by building subgraphs of the original containing all the same vertices, but only a particular kind of edge.

Tree edges are easiest, these are just the edges that appear explicitly in the spanning forest. The other edges may be distinguished by comparing preorder and/or postorder numbers of the vertices of an edge. We can summarise the situation in the following diagram:



The above diagram expresses the relationship between the four types of edge (tree edges (T), forward edges (F), back edges (B), and cross edges (C)) and the preorder and postorder numbers. Only back edges go from lower postorder numbers to higher, whereas only cross edges go from higher to lower in *both* orderings. Forward edges, which are the composition of tree edges, cannot be distinguished from tree edges by this means—both tree edges and forward edges go from lower preorder numbers to higher (and conversely in postorder)—but as we can already determine which are tree edges there is no problem. The implementation of these principles is now immediate and presented in Figure 3.

To classify an edge we generate the depth-first spanning forest, and use this to produce preorder and postorder numbers. We then have all the information required to construct the appropriate subgraph.

Algorithm 6. 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.

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

One application of this algorithm is to test for the existence of a path between two vertices:

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

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 ceases. Thus `dfs` implements a true *search* and not merely a complete *traversal*.

Algorithm 7. Biconnected components

We end by programming a more complex algorithm—finding *biconnected components*. An undirected graph is biconnected if the removal of any vertex leaves the remaining subgraph connected. This has a bearing in the problem of *reliability* in communication networks. For example, if you want to avoid driving through a particular town, is there an alternative route?

If a graph is not biconnected the vertices whose removal disconnects the graph are known as *articulation points*. Locating articulation points allows a graph to be partitioned into biconnected components (actually a partition of the *edges*). In Figure 4 vertices that are articulation points are marked with an asterisk. The naïve, brute force method requires $O(V(V+E))$ time (where the problem graph has V vertices and E edges). A more efficient algorithm is described by Tarjan (1972), where biconnected components are found during the course of a depth-first search in $O(V+E)$ time. Here we apply the same theory as Tarjan, but express it via explicit intermediate values.

```
tree :: Bounds -> Forest Vertex -> Graph
tree bnds ts = buildG bnds (concat (map flat ts))
  where
    flat (Node v ts) = [ (v,w) | Node w us <- ts] ++ concat (map flat ts)

back :: Graph -> Table Int -> Graph
back g post = mapT select g
  where  select v ws = [ w | w <- ws, post!v<post!w ]

cross :: Graph -> Table Int -> Table Int -> Graph
cross g pre post = mapT select g
  where  select v ws = [ w | w <- ws, post!v>post!w, pre!v>pre!w]

forward :: Graph -> Graph -> Table Int -> Graph
forward g tree pre = mapT select g
  where  select v ws = [ w | w <- ws, pre!v<pre!w] \\ tree!v
```

Figure 3: Classification of graph edges

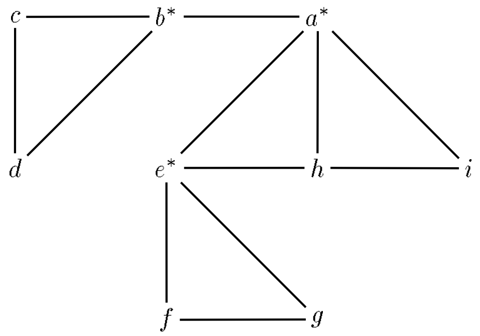


Figure 4: An undirected graph

Tarjan's method is based on the following theorem:

Theorem 6
Given a depth-first spanning forest of a graph, v is an articulation point in the graph if and only if: (i) v is a root with more than one child; or (ii) v is not a root, and for all proper descendants w of v there are no edges to any proper ancestors of v.

We apply this theorem by associating a *low point* number with every vertex. The low point number of v is the smallest DFS numbered vertex that can be reached by following zero or more tree edges, and then along a single graph edge.

We calculate low point numbers by traversing the DFS trees bottom-up, and associating each vertex with its low point number. The function `label` (see Figure 6) annotates a tree with both depth-first numbers and low-point numbers. At any vertex, the low point number is the minimum of:

1. the DFS number of the vertex;

2. the DFS numbers of the vertices reached by a single edge; and

3. the low point numbers of the vertex's descendants in the tree.

For example, the result of running `label` on the DFS spanning tree produced from the graph in Figure 4, gives the annotated tree depicted in Figure 5.

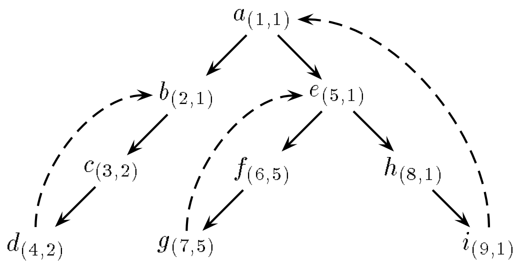


Figure 5: The depth-first forest for the undirected graph

Dashed lines are the important back edges used for calculating low points. Tree nodes are triples, for instance, $e_{(5,1)}$, represents the triple $(e, 5, 1)$, where 5 is the depth-first number and 1 the low point number of vertex e.

Now that we have low points for vertices we can calculate articulation points. By part (ii) of Theorem 6 if the depth-first number of v is less than or equal to the low point of w then v is an articulation point.

The function `collect` coalesces each DFS tree into a *biconnected tree*, that is, a tree where the node elements are biconnected components. At each node the DFS number is compared with the low-point number of all the children. If the child's low-point number is strictly less than the node's DFS number, then the component involving that vertex is not completed. On the other hand, if the node's DFS number is less than or equal to the child's low-point number, then that component is completed once the node is included. The function `bicomps` handles the special case of the root.

```
bcc :: Graph -> Forest [Vertex]
bcc g = (concat . map bicomps . map (label g dnum)) forest
  where  forest = dff g
         dnum = preArr (bounds g) forest

label :: Graph -> Table Int -> Tree Vertex -> Tree (Vertex,Int,Int)
label g dnum (Node v ts) = Node (v,dnum!v,lv) us
   where  us = map (label g dnum) ts
          lv = minimum ([dnum!v]++[ dnum!w | w <- g!v]
                         ++[ lu | Node (u,dw,lu) xs <- us])

bicomps :: Tree (Vertex,Int,Int) -> Forest [Vertex]
bicomps (Node (v,dv,lv) ts)
      = [ Node (v:vs) us | (l, Node vs us) <- map collect ts]

collect :: Tree (Vertex,Int,Int) -> (Int, Tree [Vertex])
collect (Node (v,dv,lv) ts) = (lv, Node (v:vs) cs)
  where  collected = map collect ts
         vs = concat [ ws | (lw, Node ws us) <- collected, lw<dv]
         cs = concat [ if lw<dv then us else [Node (v:ws) us]
                       | (lw, Node ws us) <- collected]
```

Figure 6: Biconnected components algorithm

Finally, `bcc` ties all the other functions together.

Coalescing the tree from Figure 5 will produce the following forest containing two trees.

$$\begin{tikzcd} & {[a,b]} \arrow[dl] & & {[a,e,h,i]} \arrow[dr] & \\ {[b,d,c]} & & & & {[e,f,g]} \end{tikzcd}$$

Figure 7: The biconnected trees

While this algorithm is complex, again it is made up of individual components whose correctness may (potentially at least) be established independently of the other components. This is quite unlike typical imperative presentations where the bones of the recursive DFS procedure are filled out with the other components of the algorithm, resulting in a single monolithic procedure.

7 Analysis of depth-first search

7.1 Complexity

Models for complexity analysis of imperative languages have been established for many years, and verified with respect to reality across many implementations. Using these models it is possible to show that traditional implementations of the various DFS algorithms are linear in the size of the graph (that is, run in $O(V + E)$ time).

Corresponding models for lazy functional languages have not been developed to the same level, and where they have been developed there has not yet been the same extensive verification. Using these models, (see for example Sands (1993)) we believe our implementation of the DFS algorithms to be linear, but because these models have not been fully tested, we also ran empirical tests ourselves.

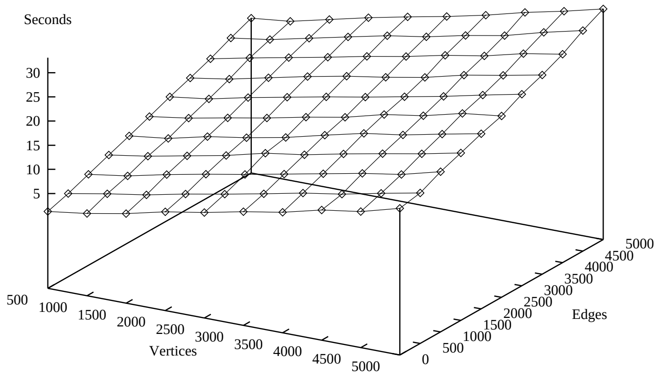


Figure 8: Measurements taken on the strongly connected components algorithm

We took measurements on the strongly connected components algorithm, which uses two depth-first searches. The results of our experiment are in Figure 8. Timings were taken on randomly generated graphs (with differing numbers of vertices and edges) and are accurate to approximately 1%.

The results are quite clear. The plotted points clearly all lie on a plane, indicating the linearity of the algorithm.

We were also curious as to the constant factor that we are paying over an imperative language. We coded up Tarjan's biconnected components algorithm in C, and compared with our Haskell implementation. For the graphs we tested Haskell was between 10 and 20 times slower than C. This was better than we expected as the Haskell implementation is multi-pass whereas the C implementation was the monolithic single-pass algorithm.

8 Related work

Kashiwagi and Wise (1991) also express their graph algorithms in Haskell. They express a graph problem in terms of a set of recursive equations, and the algorithm is the fixed point of these equations. The graphs are represented by lists, so the algorithms have poor complexity, but are suitable for parallel evaluation. Unfortunately, many of their algorithm implementations are long and unreadable, giving little insight into the structure of the problem. For example, their strongly connected components algorithm is a page of intricate Haskell.

Barth *et al.* (1991) describe *M-structures* in the parallel functional language Id which are well suited for state based computation. For instance, an M-structure array can be used for holding marks to express whether a vertex has been visited before or not during a traversal. The strength of M-structures is that they are designed to support parallel evaluation: their drawback is that referential transparency is lost. With regard to depth-first search, Reif (1985) gives strong evidence that it is inherently sequential; its computational complexity cannot be improved upon by parallel computation. So while M-structures provide a valuable method for general graph searching in parallel, they provide little help for the particular case of depth-first search.

The Graph Exploration Language (GEL) of Erwig (1992) provides explicit extensions to a lazy functional language. These are *exploration operators*, which give a concise way of expressing many graph algorithms. However, not all graph problems can be expressed in terms of a given set of predefined high-level operations, and it seems less than ideal to add new language concepts for every new class of problem that is tackled.

Burton and Yang (1990) experimented with multi-linked structures. They use arrays which are implemented using balanced trees to represent heaps. They give many examples of using multi-linked structures using heaps, one example is an arbitrary depth-first search function. A drawback with their approach is that heaps have to be passed to and returned from each function. Another is that, by using balanced trees a logarithmic factor is incurred, so their depth-first search function is not linear in the size of the graph.

9 Acknowledgements

We would like to thank Kieran Clenaghan and Simon Peyton Jones for the helpful comments they gave on a preliminary draft of this paper. We are also very grateful to Richard Bird for suggesting that DFS can be expressed using the generate/prune paradigm, and to him, Geraint Jones and Theo Norvell for inspiring the correctness proofs. The work was partly supported by the UK Science and Engineering Research Council.

References

Barth, P. S., Nikhil, R. S. and Arvind (1991), M-structures: Extending a parallel, non-strict, functional language with state, *in* J. Hughes, ed., 'Conference on Functional Programming Languages and Computer Architecture', LNCS 523, Springer-Verlag, Cambridge, Massachusetts, pp. 538–568.

Burton, F. W. and Yang, H.-K. (1990), 'Manipulating multilinked data structures in a pure functional language', *Software—Practice and Experience* **20**, 1167–1185.

Corman, T. H., Leiserson, C. E. and Rivest, R. L. (1990), *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts.

Erwig, M. (1992), Graph algorithms = iteration + data structures? The structure of graph algorithms and a style of programming, *in* E. Mayr, ed., 'Graph-Theoretic Concepts in Computer Science', LNCS 657, Springer-Verlag, pp. 277–292.

Harrison, R. (1993), *Abstract data types in Standard ML*, John Wiley and Sons.

Holyer, I. (1991), *Functional programming with Miranda*, Pitman, London.

Hopcroft, J. E. and Tarjan, R. E. (1973), 'Algorithm 447: Efficient algorithms for graph manipulation', *Communications of the ACM* **16**(6), 372–378.

Hudak, P., Peyton Jones, S. L., Wadler, P., Arvind, Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, R., Nikhil, R. S., Partain, W. and Peterson, J. (1992), 'Report on the functional programming language Haskell, Version 1.2', *ACM SIGPLAN Notices* **27**(5).

Kashiwagi, Y. and Wise, D. S. (1991), Graph algorithms in a lazy functional programming language, *in* 'Proceedings of the 4'th International Symposium on Lucid and Intensional Programming', pp. 35–46. Also available as Technical Report Number 330, Computer Science Department, Indiana University.

Launchbury, J. (1993), Lazy imperative programming, *in* 'Workshop on State in Programming Languages', ACM SIGPLAN, Copenhagen, Denmark, pp. 46–56.

Launchbury, J. and Peyton Jones, S. L. (1994), Lazy functional state threads, *in* 'Conference on Programming Language Design and Implementation', ACM SIGPLAN, Orlando, Florida.

Manber, U. (1989), *Introduction to Algorithms—A Creative Approach*, Addison-Wesley, Reading, Massachusetts.

Moggi, E. (1989), Computational lambda-calculus and monads, *in* 'Symposium on Logic in Computer Science', IEEE, Asilomar, California.

Paulson, L. C. (1991), *ML for the working programmer*, Cambridge University Press, Cambridge.

Peyton Jones, S. L. and Wadler, P. (1993), Imperative functional programming, *in* '20'th Symposium on Principles of Programming Languages', ACM, Charleston, North Carolina.

Reif, J. H. (1985), 'Depth-first search is inherently sequential', *Information Processing Letters* **20**, 229–234.

Sands, D. (1993), A naïve time analysis and its theory of cost equivalence, TOPPS report D-173, DIKU, University of Copenhagen, Denmark.

Sharir, M. (1981), 'A strong-connectivity algorithm and its applications in data flow analysis', *Computers and mathematics with applications* **7**(1), 67–72.

Tarjan, R. E. (1972), 'Depth-first search and linear graph algorithms', *SIAM Journal of Computing* **1**(2), 146–160.

Wadler, P. (1990), Comprehending monads, *in* 'Conference on Lisp and Functional Programming', ACM, Nice, France, pp. 61–78.

Appendix

Imperative features were initially introduced into the Glasgow Haskell compiler to perform input and output, see Peyton Jones and Wadler (1993). The approach is based on monads (Moggi 1989, Wadler 1990), and can easily be extended to achieve *in-situ* array updates. Launchbury (1993) showed how the original model could be extended to allow the imperative actions to be delayed until their results are required. This is the model we use.

We will use the monad of state-transformers with type constructor `ST` which is defined:

```
type ST s a = a -> (a,s)
```

So elements of type `ST s Int`, say, are functions which, when applied to the state, return a pair of an integer together with a new state. As usual we have the unit `returnST` and the sequencing combinator `thenST`:

```
returnST :: a -> ST s a
returnST a s = (a,s)

thenST :: ST s a -> (a -> ST s b) -> ST s b
(m `thenST` k) s = k a t  where  (a,t) = m s
```

The `ST` monad provides three basic array operations:

```
newArr  ::Ix i=> (i,i) -> a ->ST s (MutArr s i a)
readArr ::Ix i=> MutArr s i a -> i -> ST s a
writeArr::Ix i=> MutArr s i a -> i -> a ->ST s ()
```

The first, `newArr`, takes a pair of index bounds (the type `a` must lie in the index class `Ix`) together with an initial value, and returns a reference to an initialised array. The time this operation takes is linear with respect to the number of elements in the array. The other two provide for reading and writing to an element of the array, and both take constant time.

Finally, the `ST` monad comes equipped with a function `runST`.

```
runST :: (∀s . ST s a) -> a
```

This takes a state-transformer function, applies it to an initial state, extracts the final value and discards the final state. The type of `runST` is not Hindley-Milner because of the nested quantifier, so it must be built-in to Haskell. The universal quantifier ensures that in a state thread variables from other state threads are not referenced. For details of this see Launchbury and Peyton Jones (1994).

So, for example,

```
  runST (newArr (1,8) 0     `thenST` (\nums ->
         writeArr nums 5 42 `thenST` (\_ ->
         readArr nums 5     `thenST` (\v ->
         returnST v))))
```

will return `42`. This can be read as follows: run a new state thread extracting the final value when finished; create a new array indexed from `1` to `8` with components all `0`; then bind this array to `nums`; write to array `nums` at index `5` the value 42; then read the component in `nums` at index `5` and bind this value to `v`; finally return value `v`. Note that the final expression `returnST v` is unnecessary as `readArr` returns a value. The parentheses immediately after `` `thenST` `` are also unnecessary, as Haskell's grammar binds lambda expressions tighter than infix functions.