Undirected graph cut vertex detection in haskell

Kurian Jacob
Saurav Shrivastava

November 22, 2017

Contents

1 Introduction

Our objective is to study Data.graph library present in haskell.this package provide various graph algorithms such as DFS,finding strongly connected or Bi-connected component,Topological sort and reachability of vertex etc. In this repert first ,we will first explore the representation of graph using Data.graph package then we modify existing DFS algorithm to detect articulation point (cut vertex) in Undirected graph.

2 Representation of graph

There are many ways to represent (Undirected) 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 Undirected 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.

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

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

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')]
```

3 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 Vertex information,where forest vertex information consist of tuple of status,depth,low-point and visited.this is defined as a new type as follows:

```
type VertexInfo = (Bool, Int, Int, Int, Int)
```

3.1 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.Definition of modified dfs is given as follows:

```
newdfs::Graph -> [Vertex] -> Forest VertexInfo
newdfs graph v = prune (bounds graph) (map (generate graph) v)
```

3.1.1 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.

3.1.2 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.
we define prune as follows:

```
prune::Bounds -> Forest Vertex -> Forest VertexInfo
prune bounds ts = run bounds (chop ts 0 False 0 [])
```

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::Forest Vertex -> Int -> Bool ->
                       Int -> Forest VertexInfo -> ArtSetM s (Forest VertexInfo)
chop [] p pexists d children = if not pexists then
return children
else
do
info <- retrieve p
return [(Node info children)]

chop (Node v ts : us) p pexists d children = do
(visited, depth, low, parent, id) <- retrieve v
if visited then
if pexists && (p /= parent) then
do
(pvisited, pdepth, plow, pparent, pid) <- retrieve p
update p (pvisited, pdepth, (min plow depth), pparent, pid)
chop us p True d children
else
chop us p False d children
else do
update v (True, d, d, p, v)
as <- chop ts v True (d + 1) []
if pexists then do
(pvisited, pdepth, plow, pparent, pid) <- retrieve p
(_, _, nlow, _, _) <- retrieve v
update p (pvisited, pdepth, (min plow nlow), pparent, pid)
bs <- chop us p pexists d (as++children)
return bs
else do
```

```
bs <- chop us p pexists d (as++children)
return 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.

4 Articulation point

A vertex in an Undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.

For a disconnected Undirected graph, an articulation point is a vertex removing which increases number of connected components.

The algorithm utilizes the properties of the DFS tree. In a DFS tree, a vertex u is parent of another vertex v if v is discovered by u. A vertex u is articulation point iff one of the following two conditions is true:

- u is the root of the DFS tree and has at least two children
- u is not the root and no vertex in the subtree rooted at one of the children of u has a back edge to an ancestor of u.

Let `disc_time[u]` be the time at which a vertex u was discovered/explored during the dfs traversal. Let low[u] be the earliest discovery time of any vertex in the subtree rooted at u or connected to a vertex in that subtree by a back edge. Then

- If some child x of u has low[x] $\geq$ `disc_time[u]`, then u is an articulation point.
- low[u] = min(low[v] | v is a child of u $\cup$ `disc_time[u]` | (u, x) is a back edge from u)

This gives rise to the following algorithm for finding all articulation points:

```
articulation graph = collectForest (newdfs graph [(head (vertices graph))])

maxLow::Forest VertexInfo -> Int
maxLow l = maximum (map f l) where
f (Node (_, _, low, _, _) _) = low

collectVertex::Tree VertexInfo -> [Vertex]
collectVertex (Node info []) = []
collectVertex (Node (_, d, _, _, id) children) = (if (maxLow children) >= d then [id] else
collectDescend::Forest VertexInfo -> [Vertex]
collectDescend [] = []
collectDescend (x:xs) = collectVertex x ++ collectDescend xs
collectForest::Forest VertexInfo -> [Vertex]
collectForest [(Node (_, _, _, _, id) children)] = if length children > 1 then id:rest else
rest = collectDescend children
```

The time and space complexity of the algorithm is same as than of DFS i.e., O(V + E).

5 Conclusion

Data.graph are a powerful tool in the toolbox of a functional programmer.It helps to implement various graph algorithm in unweighted graph,however weighted graph algorithm are absent in this package.