Undirected graph cut vertex detection in haskell

CS613 Project

Kurian Jacob
Saurav Shrivastava

IIT Bombay

November 22, 2017

Table of Contents

Introduction

- Data.graph package is a powerful tool which provide functional programmer to implement graph alogarithms.
- Some inbuilts algorithm in this package are DFS,Strongly connected component,bi-connected component and reachability etc.
- Our aim is to implement Articulation point(cut vertex) detection Algorithm in Undirected graph using this package.

Representation of graph

- Graph is represented as 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.

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

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')]
```

Depth first search

- 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)
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 returned forest Vertex information.

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

- Use of a technique common in lazy functional programming: generate then prune

Depth first search[CONTD.]

- Definition of modified dfs is given as follows:

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

- 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))
```

- Goal of pruning the (infinite) forest is to discard subtrees whose roots have occurred previously.

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

- The final result of prune is the value generated by chop , the final state being discarded.

Depth first search[CONTD.]

- ```
  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)]
  ```


  ```

Depth first search[CONTD.]

- if vertex v is visited then:

```
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
```

Depth first search[CONTD.]

- if vertex v is unvisited then:

```
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
```

Articulation point

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)

Articulation point

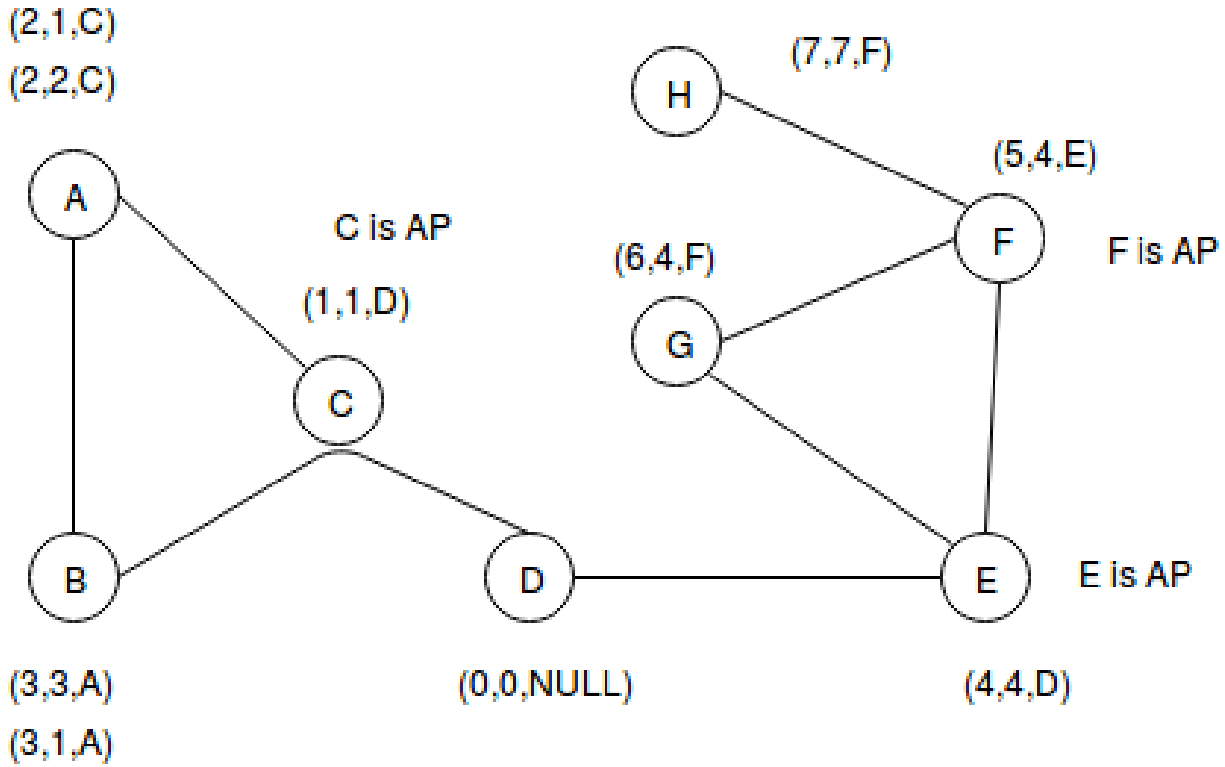

Figure: Articulation point example

Articulation point

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

```
articulation graph = collectForest (newdfs graph [(head (v

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 children
```

Articulation point

```
collectDescend::Forest VertexInfo -> [Vertex]
collectDescend [] = []
collectDescend (x:xs) = collectVertex x ++ collectDescend xs
collectForest::Forest VertexInfo -> [Vertex]
collectForest [(Node (_, _, _, _, id) children)] = if length c
rest = collectDescend children
```

THANK YOU !