Algorithms Assignment 8

Andrew Osborne

April 6, 2019

Problem 15.2-2

The solution to this problem is a simple modification to the already existing `Print-Optimal-Parens(s,i,j)` which I will display below

```
Optimal-Mult(A,s,i,j)
  if i == j
    return A[i]
   else
    // assuming matrix multiplication is already implemented
    return Optimal-Mult(A,s,i,s[i,j])*Optimal-Mult(A,s,s[i,j]+1,j)
```

Problem 22.2-9

Now, let us suppose that `G` is a graph with edges and vertices and that we have access to the `BFS(G,s)` which is written in the text. Firstly, let us denote the edge between nodes `u` and v by `E(u,v)` and let us further suppose that we may store such objects in an array. Then, in the syntax of the book's pseudocode, our algorithm is given by

```
BFS(G,s) // now every node but s has non-null pi value
Assign to each vertex in G a distinct integer .id value
let V be an empty vector of edges

DoublePath(G,V,s)
  for each vertex u in G.adj[s]
    if u.pi == s // hits each node exactly once
      V.append(E(s,u))
      DoublePath(G,V,u)
    else if s.id < u.id // hits at most 2*E times
      V.append(E(s,u))
      V.append(E(u,s))
```

With `BFS` , we are constructing a tree with `s` at the root and our recursion ensures that all paths in said tree are traversed exactly once in each direction, and the other appendation accounts for those paths which are pruned by BFS. Note that we need the `.id` values because we would otherwise traverse each pruned edge twice in each direction, rather than once. `BFS` runs in $O(E+V)$ time and our recursion runs in

$$\Theta(V) + O(E+V) + O(E+V) = O(E+V)$$

time.

22.3-1

In the following tables, let C denote cross edges, T denote tree edges, F denote forward edges, and B denote back edges. Furthermore let "-" indicate that there are no valid edges. Finally $A[i, j]$ represents the valid edges between color i and color j.

For Directed Graphs

-	Black	White	Gray
Black	all	-	B
White	C	all	CB
Gray	FCT	T	BFT

For Undirected graphs

-	Black	White	Gray
Black	TB	-	B
White	-	TB	B
Gray	T	T	TB

22.3-7

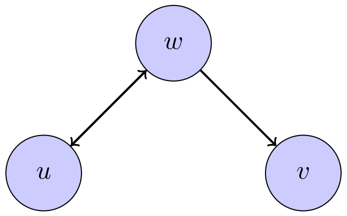
This problem will be answered almost exclusively with pseudocode because of the simplicity of the problem. Let us assume that we possess a stack data structure which has `pop()` which removes and returns the top element of the stack, `peek()` which returns but does not remove the object on the top of the stack and `push(v)` which pushes `v` onto the top of the stack. Finally assume our stack datastructure has a `hasNext()` field which returns true if and only if `pop()!=NIL` .

```
DSF(G)
  for each vertex v in G.V
    v.pi = NIL
    v.color = white
  time = 0
  let S be an empty stack of vertices

  for each vertex v in G.V
    if v.color == white
      time += 1
      v.color = gray
      v.d = time
      S.push(v)
      while S.hasNext()
        // updates peek() going depth first
        // if we pass this loop, we arrived at
        // a node with no white descendants
        for vertex u in G.adj[S.peek()]
          if u.color == white
            time += 1
            u.color = gray
            u.pi = S.peek()
            u.d = time
            S.push(u)
        time = time + 1
        x = S.pop()
        x.color = black
        x.f = time
```

This algorithm is identicle in nature to the one in the text. Note that the peek() function is particularly important here. Also note that in the innermost for loop, when I push an element to S, that element will be returned as S.peek() in future iterations. The innermost for-loop will only exit when a node with no white children is found, at which point we will pop said element off of the stack, color it black, declare it finished, and proceed in this way until S is empty.

22.3-8



Clearly, in the above graph, there is a path from u to v which is identically $u \to w \to v$ but if we begin our DFS on w, discover u, and then discover v, we have

node	d	f
w	1	6
v	4	5
u	2	3

This is sufficient for a counterexample to the proposed conjecture.