FNU ANIRUDH

Assignment-7

(21.1.3) $2|E|$ times FIND-SET operations, $|V|-K$ times UNION operation

(21.2.1)
```
MAKE-SET(x):
    Initialize a new linked list
    Insert node x

FIND-SET(x):
    return rep[x]

UNION(x, y):
    Let smallist, biglist be the list with less and larger
list according to size[x], size[y]
    put smallist to the tail in biglist
```

(21.2.4) $T = O(n) + O(n) = O(n)$

Example of best case because in each UNION, we only move one element.

(21.3.3) First, we need to find a sequence of m operations on n elements that takes $\Omega(m \lg n)$ time. Start with n MAKE-SETS to create singleton sets $\{x_1\}$, $\{x_2\}$, ---- $\{x_n\}$. Next perform the $n-1$ UNION operations shown below to create a single set whose tree has depth $\lg n$.

UNION (x_1, x_2) — $n/2$ of these
UNION (x_3, x_4)
UNION (x_5, x_6)
⋮
UNION (x_{n-1}, x_n)

UNION (x_2, x_4) — $n/4$ of these
UNION (x_6, x_8)
UNION (x_{10}, x_{12})
⋮
UNION (x_{n-2}, x_n)

UNION (x_4, x_8) — $n/8$ of these
UNION (x_{12}, x_{16})
UNION (x_{20}, x_{24})
⋮
UNION (x_{n-4}, x_n)

⋮

UNION $(x_{n/2}, x_n)$ — 1 of these

Finally, perform $m - 2n + 1$ FIND-SET operations on the deepest element in the tree. Each of these FIND-SET operations takes $\Omega(\lg n)$ time. Letting $m \geq 3n$, we have more than $m/3$ FIND-SET operations, so that the total cost is $\Omega(m \lg n)$.

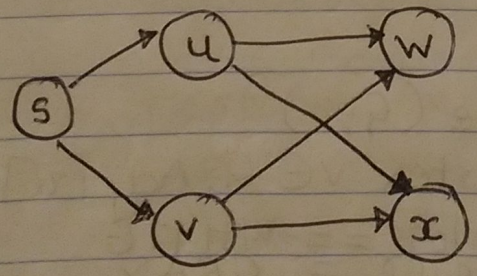
(21.4.4) each MAKE-SET and LINK operation takes $O(1)$ time. Because the rank of a node is an upper bound on its height, each find path has length $O(\lg n)$ which in turn implies that each FIND-SET operations on n elements takes $O(m \lg n)$ time. It is easy to prove an analogue of Lemma to show that if we convert a sequence of m' MAKE-SET, UNION, and FIND-SET operations into a sequence of m MAKE-SET, LINK and FIND-SET operations that take $O(m \lg n)$ time, then the sequence of m' MAKE-SET, UNION and FIND-SET operations takes $O(m' \lg n)$ time.

(22.2.3)
```
BGS(G, s)
    for each vertex u ∈ G.V - {s}
            u.color = WHITE
            u.d = ∞
            u.π = NIL
(5) s.color = GRAY
    s.d = 0
    s.π = NIL
    Q = ∅
    Enqueue(G, s)
    while Q ≠ ∅
        u = Dequeue(G, s)
        for each vertex v ∈ G.Adj[u]
            if v.color == WHITE
(14)            v.color = GRAY
                v.d = u.d + 1
                v.π = u
                Enqueue(G, v)
        u.color = BLACK
```

During the BFS procedure, color of each vertex is initialized as WHITE, changed to GRAY once the node is visited and enqueued, and finally, set to BLACK when it's dequeued and every neighbor vertex is ~~limited~~ visited. Color of node stays GRAY during a period, which lasts from the point when it is enqueued to the point when it is dequeued, followed by the step to visit every neighboring vertex. Therefore the meaning of GRAY color is roughly equivalent to existence in queue. Line 5 and 14 which set vertex color to GRAY, should be removed, in order to use a single bit to store each vertex color.

Or simply the entire BFS procedure does not have any line that distinguishes BLACK from GRAY. It only checks either it is WHITE or non-WHITE, which shows that it is meaningless to have two colors to represent 'visited' vertices.

(22.2.5) The edges in E_π are shaded in the following graph:



To see that E_π cannot be a breadth-first tree, let's suppose that Adj[s] contains u before v. BFS adds edges (s, u) and (s, v) to the breadth-first tree. Since u is enqueued

before v, BFS then adds edges (u, w) and (u, x). (The order of w and x in $Adj[u]$ doesn't matter) Symmetrically, if $Adj[s]$ contains v before u, then BFS adds edges (s, v) and (s, u) to the breadth-first tree, v is enqueued before u and BFS adds edges (v, w) and (v, x). (Again the order of w and x in $Adj[v]$ doesn't matter). BFS will never put both edges (u, w) and (v, x) into the breadth-first tree. In fact, it will never put both edges (u, x) and (v, w) into breadth-first tree.

22.2.7 Create a graph G where each vertex represents a wrestler and each edge represents a rivalry. The graph contains n vertices and r edges.

Perform as many BFS's as needed to visit all vertices. Assign all wrestlers whose distance is even to be babyfaces and all wrestlers whose distance is odd to be heels. Then check each edge to verify that it goes between a babyface and a heel. This solution would take $O(n+r)$ time for the BFS, $O(n)$ time to designate each wrestler as a babyface or heel, and $O(r)$ time to check edges, which is $O(n+r)$ time overall.

22.3.7 Assume that the stack has following operation:

PUSH (S, v) - pushes v into the stack;

POP (S) - returns the top of the stack and removes it;

TOP (S) - returns the top of the stack without removing it

Denote an empty stack by EMPTY.

Then the pseudocode of this algorithm becomes as follows:

```
DFS_STACK(G,s)
 for each vertex u ∈ V(G) - {s}
   {
       u.color = WHITE
       u.pred = NIL
   }
  time = 0
   S = EMPTY

  time = time + 1
  s.td = time
 s.color = GRAY
 PUSH(S,s)
 While S != EMPTY
  {
    t = TOP(S)
if ∃v ∈ V(G). Adj[t] s.t. v.color == WHITE //
        v's adjacency list hasn't been fully
 examined
    {
       v.pred = t
       time = time + 1
       v.td = time
       v.color = GRAY
       PUSH(S,v)
    }
```

```
else
  {
    t = POP(S)
    time = time + 1
    v.tf = time
    v.color = BLACK
  }
}
```

22.3.12 The following pseudocode modifies the DFS and DFS-VISIT procedures to assign values to the cc attributes of vertices

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
counter = 0
for each vertex u ∈ G.V
        if u.color == WHITE
            counter = counter + 1
            DFS-VISIT (G, u, counter)
DFS-VISIT (G, u, counter)
u.cc = counter        // label the vertex
time = time+1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
   if v.color == WHITE
            v.π = u
```

```
DFS-VISIT(G, v, counter)
u.color = BLACK
time = time + 1
u.f = time
```

This DFS increments a counter each time DFS-VISIT is called to grow a new tree in the DFS forest. Every vertex (and added to the tree) by DFS-VISIT is labeled with the same counter value. Thus u.cc = v.cc if and only if u and v are visited in the same call to DFS-VISIT from DFS, and the final value of the counter is the number of calls that were made to DFS-VISIT by DFS. Since every vertex is visited eventually every vertex is labeled.

Thus all we need to show is that the vertices visited by each call to DFS-VISIT from DFS are exactly the vertices in one connected component of G.

All vertices in a connected component are visited by one call to DFS-VISIT from DFS:

Let u be the first vertex in component C visited by DFS-VISIT. Since a vertex becomes non-white only when it is visited, all vertices in C are white when DFS-VISIT is called for u. Thus, by the white-path theorem, all vertices in C become descendants of u in the forest, which means that all vertices in C are visited (by recursive calls to DFS-VISIT from DFS are in

the same connected component:

If two vertices are visited in the same call to DFS-VISIT from DFS, they are in the same connected component, because vertices are visited only by following paths in G (by following edges found in adjacency lists, starting from some vertex).

22.4.5 First, for each vertex, we need to count how many incoming edges it has. We store these counts in a dictionary keyed by the vertices. This takes $\Theta(V+E)$ time.

For each vertex with 0 incoming edges, store it in a special dictionary. Call that dictionary zero.

We repeatedly do the following: Take a vertex out of zero. For every edge coming out of that vertex, decrement the count of that edge's target vertex. If you happen to decrement a count to 0, add that vertex to zero.

This touches every vertex once and every edge once, so it also takes $\Theta(V+E)$ time.

If there is a cycle, at some time when we try to take a vertex out of zero, we won't find any