CP312 Assignment

Memet Rusidovski: 130951550

March 19, 2022

1 Breath First Search

Please see final page in PDF file.

2 Depth First Search

Please see final page in PDF file.

3 Determine If G is Valid Tree

Algorithm 1 valid tree algorithm

```
1: visited = {}
2: procedure DFS($i, edges\{\}, wurr$)          ▷ //n is the number of edges
3:     if $i$ in $visited$ then
4:         return $false$
5:     end if
6:     visted[ i ] = 1
7:     for j in edges[i] do
8:         if  not dfs(j, edges, i) then
9:             return $false$
10:        end if
11:    end for
12:    return $true$
13: end procedure
14: procedure isValid(n, edges{})          ▷ // edge are in an adjacency list
15:    if n <= 1  then
16:        return true
17:    end if
18:    return $dfs(0, edges, -1)$
19: end procedure
```

This algorithm is normally $O(V + E)$ but when applied to verify G is a graph it becomes $O(v)$. This is because a valid tree should have one less edge than vertices. this means the DFS Algorithm should only Visit each edge outer vertex once. if it did have more edge then the tree would not be valid and the algorithm would stop before traversing $|E|$ edges.

4 Topological sorting

[u, t, z, v, x, y, w]
[u, t, z, v, x, w, y]
[u, t, v, z, x, y, w]
[u, t, v, z, x, w, y]
[u, t, v, x, z, y, w]
[u, t, v, x, z, w, y]
[u, t, v, x, w, z, y]
[u, z, t, v, x, y, w]
[u, z, t, v, x, w, y]
9 total topological sorts

5 Proof of Zero in degree

Solution was done by hand and added later in the PDF.

6 G to G'

Adjacency list: Since G ' contains all paths of two edges or less then G' will contain all of G since path (u, v) is an edge and a path of length one. same is true with disconnected vertices they will also belong to G'. With this G is copied to G'.

Now we traverse the entire adjacency list and start a BFS search at each vertex. We use BFS to get an exhaustive search. When a path of 3 Distance 2 is found an. edge from this point connecting the starting point is added to G'.

By the diagram at the end of the PDF we see the search finds a path of two edges and then adds this Path (in Blue to G' as an edge.

The search stops once all vertices of distance two are found. Then the search repeats for all Vertices in the list. This gives us G'

Time Complexity would be $O(V + cE)$ where C is a multiple depending on the path length. is this case $O(V + 2E)$ sine all nodes will be traversed and all edge Traversed twice in search.

To do this without BFS we can start by taking the first vertex and looping through each node in it's adjacent list. Taking each node and making a new edge from the original vertex to ever connected to the original adjacent list. For example in the diagram we start with P. We take all nodes adjacent, so R and W. Then we loop through and get every node adjacent to R and W to make a new edge back to P. Thus we add edges, (P, X), (P, S), (P, Y). We repeat for all the next vertices in the adjacency list. Same time complexity.

Algorithm 2 adjacency list

```
G' = {}
G' = G
procedure makePrime($list${})
    c = 0
    for i in list do
        for j in i do
            G'.append(edge(list[c], j)
        end for
        c++
    end for
    return $G'$
end procedure
```

Adjacency matrix: In the adjacency matrix version we loop through the entire 2D array. Rows being the starting vertex and columns being the destination, ie Edge(row, column). when we hit a 1 in the matrix we go the that column vertex in G and loop through to find all vertices in the row with value 1. Then we add these as new edges starting at original row value.

The time complexity is O(V^3) since we need to traverse the entire matrix which is V^2 then again each row make V^3.

Algorithm 3 adjacency matrix

```
procedure makePrime
    G' = [vertices][vertices]
    G' = G
    for x in G do
        for y in x do
            if y == 1 then
                for x in G[y] do
                    if z == 1 then
                        G'.add(edge(x, z))
                    end if
                end for
            end if
        end for
    end for
    return $G'$
end procedure
```

7 Shortest Cycle

By completing a BFS search on the point v where the finishing point is also v we get an algorithm that finds a shortest cycle. The algorithm looks through the array until it hits vertex v again and returns the last point before hitting v. we take this last point and we can backtrack using the predecessors stored in our hash table. Then add a point from the last node to v to complete our cycle. Since this is a breadth first search algorithm the worst case run time is O(V+E). This is because at most we search all edges and vertices to find a cycle.

Algorithm 4 Shortest Cycle

```
visited = {'key' : { 'a' : dist, 'b' : pred} }
s = source node
list = adj list
count = 0
procedure BFS($n, count$)
    for i in list do
        for j in i do
            if finds vertex v then
                return last vertex
            end if
            visited[j]['a'] = count
            visited[j] ['b] = n
        end for
        BFS(i, count + 1)
    end for
    return $G'$
end procedure
```

8 Largest Number of Components

Given an adjacency list we loop through all the vertices. We count all the edges connected to each vertex in the list and keep track off the vertex with the most connected edges. Once we loop trough we take the vertex with the most edges and delete it, returning the number of edges.

Time complexity is Theta(V+E) since in all cases you will need to traverse the entire list, so the bound is appropriate.

Algorithm 5 Largest number of components

```
G = [vertices][edges]
highestVertex = Pair()                          ▷ // key value pair
count = 0
procedure findVert
    for i in G do
        for j in i do
            count++
            if highestVertex.value < count then
                highestVertex = (i, count)
            end if
        end for
        count = 0
    end for
    return $highestVertex.value$
end procedure
```

CP312 Assignment # 5

Memet Rusidovski
130951550

Q1 <u>BFS Algorithm</u>

example: y(∅, 0) node (pred, Distance)

Queue

Step 1 y(∅, 0)

x(y, 1)
2. y(∅, 0)

u(y, 1)
x(y, 1)
3. y(∅, 0)

t(x, 2)
u(y, 1)
x(y, 1)
4. y(∅, 0)

w(x, ~~3~~ 2)
t(x, 2)
u(y, 1)
x(y, 1)
5. y(∅, 0)

s(w, 3)
w(x, 2)
t(x, 2)
u(y, 1)
x(y, 1)
Step 6 y(∅, 0)

r(s, 4)
s(w, 3)
w(x, 2)
t(x, 2)
u(y, 1)
x(y, 1)
7. y(∅, 0)

v(r, 5)
r(s, 4)
s(w, 3)
w(x, 2)
t(x, 2)
u(y, 1)
x(y, 1)
8. y(∅, 0)

<u>Done</u>

Path from y to v

DFS Algorithm   node(Pred , Discovery time , finish)

Q2

Step1 y(∅, 0, ∅)

x(y, 1, ∅)
2. y(∅, 0, ∅)

t(x, 2, ∅)
x(y, 1, ∅)
3. y(∅, 0, ∅)

w(t, 3, ∅)
t(x, 2, ∅)
x(y, 1, ∅)
4. y(∅, 0, ∅)

→ Hit node Alredy visited

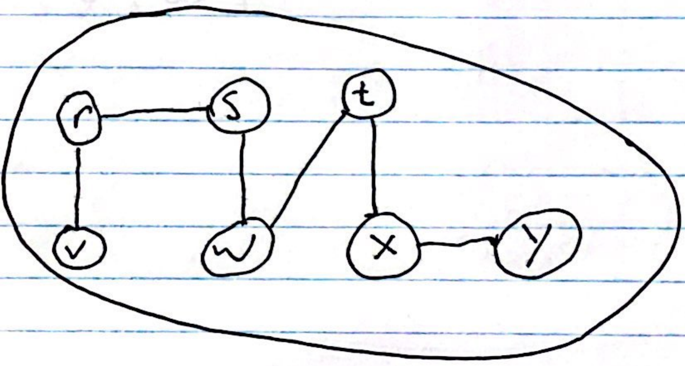
w(t, 3, ∅)
t(x, 2, ∅)
x(y, 1, ~~∅~~ 4)
5. y(∅, 0, ∅)

s(w, 5, ∅)
w(t, 3 ∅)
t(x, 2, ∅)
x(y, 1, 4)
6. y(∅, 0, ∅)

r(s, 6, ∅)
s(w, 5, ∅)
w(t, 3, ∅)
t(x, 2, ∅)
x(y, 1, 4)
7. y(∅, 0, ∅)

v(r, 7, ∅)
r(s, 6, ∅)
s(w, 5, ∅)
w(t, 3, ∅)
t(x, 2, ∅)
x(y, 1, 4)
8. y(∅, 0, ∅)

v(r, 7, 8
r(s, 6, 9
s(w, 5, 10
w(t, 3, 11
t(x, 2, 12
x(y, 1, 4)
9-15 y(∅, 0, ~~∅~~ 13)



Q5. assume For contridiction that all vertices $v \in V$ indegree ~~≠~~ ≥ 1.

~~this means all points are connect~~
~~this means all vertices are are connected~~

$$\begin{tikzcd}
V_1 \arrow[r] & V_2 \arrow[r] & V_3 \arrow[r] & V_4 & \cdots\cdots \arrow[r] & V_n
\end{tikzcd}$$

if all vertices V_1 to V_n ~~a~~ are connected by out degree there must be an edge connecting every vertex by in degree as well

$$\begin{tikzcd}
V_1 & V_2 \arrow[l] & V_3 \arrow[l] & V_4 \arrow[l] & \cdots\cdots & V_{n-1} \arrow[ll, bend right] & V_n \arrow[l]
\end{tikzcd}$$

However the only way to connect all points in these two ways ~~is~~ creates a cycle for a finite graph. Since a DAG must connect all ~~pts~~ vertices with ~~in~~ out degree and connecting all points with ~~out~~ indegree creates a cycle this contradicts the definition of a DAg.

As well ~~8~~ Dag's ~~was~~ must be topologically Searchable and without a starting point Violate the definition of a Dag