CSCI 570 : HW-01 (Spring 2021)

Aditya Jain

January 30, 2021

Question 1

Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if f(n)= O(g(n))

$$2^{logn}, (\sqrt{2})^{logn}, n \cdot log(n)^3, 2^{\sqrt{2logn}}, 2^{2^n}, n \cdot logn, 2^{n^2}$$

Answer

Lets consider all given functions as

$$f_1 = 2^{logn}, f_2 = (\sqrt{2})^{logn}, f_3 = n \cdot log(n)^3, f_4 = 2^{\sqrt{2logn}}, f_5 = 2^{2^n}, f_6 = n \cdot logn, f_7 = 2^{n^2}$$

Now, lets take logarithm and simplify above functions

$log(f_1) = log(2^{logn}) = logn \cdot log2 = logn$

$log(f_2) = log((\sqrt{2})^{logn}) = logn \cdot log(\sqrt{2}) = \frac{logn}{2}$

$log(f_3) = log(n(logn)^3) = logn + 3loglogn$

$log(f_4) = log(2^{\sqrt{2logn}}) = \sqrt{2logn} \cdot log2 = \sqrt{2logn}$

$log(f_5) = log(2^{2^n}) = 2^n \cdot log2 = 2^n$

$log(f_6) = log(n \cdot logn) = logn + loglogn$

$log(f_7) = log(2^{n^2}) = n^2 \cdot log(2) = n^2$

Using asymptotic analysis, growth rate is in order ,

$$log(f_4) < log(f_2) < log(f_1) < log(f_6) < log(f_3) < log(f_7) < log(f_5)$$

Since log is monotonically increasing, This also implies , $f_4 < f_2 < f_1 < f_6 < f_3 < f_7 < f_5$

Thus,

$$2^{\sqrt{2logn}} < (\sqrt{2})^{logn} < 2^{logn} < nlogn < n(logn)^3 < 2^{n^2} < 2^{2^n}$$

Question 2

Give a linear time algorithm based on BFS to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. It should not output all cycles in the graph, just one of them. You are NOT allowed to modify BFS, but rather use it as a black box. Explain why your algorithm has a linear time runtime complexity.

Answer

Algorithm:

Lets consider our black box BFS returns a BFS tree given a starting node. We will then compare number of edges in our BFS tree with the number of edges in Graph.

If number of edges is same in both BFS tree and Graph then our graph is a tree and do not contain any cycles. Here we wll output 0. Else, number of edges in graph will be more and Graph contains a cycle. Here we will output 1.

Explanation:

A BFS tree is a subset of edges from graph. Any two vertex in a tree (our BFS tree) must have atleast one same ancestor and will be mutually reachable from one another using that ancestor. But if Graph contain one more edge between any two vertices then it is introducing one more connection path and thus which will form a cycle.

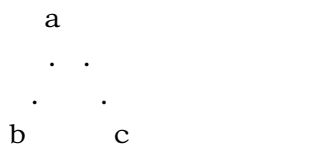

Fig 1: BFS tree
b and c are mutually reachable from one another via node a

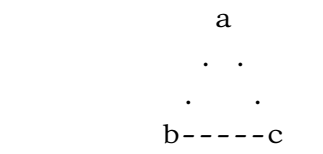

Fig 2: Graph
if there is one more edge (b,c), then b and c will form a cycle via a

Pseudo Code:

```
bfs_tree = BFS(source)
count_edges_tree <- count(bfs_tree) #bfs tree is list of edges in tree
count_edges_graph <- count(graph) #graph is list of edges in graph
if count_edges_tree == count_edges_graph:
    print(0)
else:
    print(1)
```

Runtime complexity of Algorithm:

1. Finding BFS tree using BFS algorithm - O(E+V)
2. Counting edges in graph - O(E)
3. Counting edges in tree- O(E)
4. Comparing edge count - O(1)
5. **Overall Complexity of algorithm : O(E+V)**

Question 3

A binary tree is a rooted tree in which each node has at most two children. Show by induction that in any non empty binary tree the number of nodes with two children is exactly one less than the number of leaves

Answer

Lets consider a binary tree with number of leaves be n. Let P(n) denote number of nodes with 2 children given number of leaves n.

$$Predicate\ is\ ,\ P(n) = n - 1 \ \ \forall \ \ n > 0$$

We will prove this by Induction

For base case, Tree with one leave, $P(1) = 1 - 1 = 0$, this is true since there is no node with 2 children if there is only 1 leave.

Inductive Hypothesis Lets assume predicate holds for k leaves, P(k) = k-1

Inductive Step For k+1 leaves, we will add new node as a second child, if we add new node as first child, the number of leaves will remain same, as shown in fig 3.1.

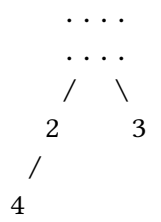

Fig 3.1
Adding node 4 will not increase number of leaves

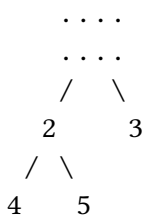

Fig 3.2
Adding node 5 will increase number of leaves as well as number nodes with 2 children

If we add new node as second child, the number of nodes with 2 childern will increase by one, as shown in fig 3.2.

$$P(n + 1) = P(n) + 1 = n - 1 + 1 = n$$

By this inductive step we proved that P(n) = n-1.

Q.E.D

Question 4

Prove by contradiction that a complete graph K_5 is not a planar graph. You can use facts regarding planar graphs proven in the textbook

Answer

According to the theorem, a connected planar graph with atleast 3 vertices must satify below inequality,

$E \leq 3V - 6$

where,
V = number of vertices
E = number of edges

Suppose, K_5 is a complete planar graph, So here V = 5, E = $\binom{5}{2}$ = 10.

$3V - 6 = 9 \Rightarrow E > 3V - 6$

This is a contradiction.

Hence, K_5 is not a planar graph.

Question 5

Suppose we perform a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation

Answer

According to algorithm described in problem:

Cost of i^{th} operation, $Cost(i) = \begin{cases} i, & \text{if } log_2(i) \text{ is an integer} \\ 1, & \text{otherwise} \end{cases}$

So, for n operations

$$\underline{\text{Cost of operations where i is not exact power of 2}} = n - log_2(n)$$

$$\begin{aligned} \underline{\text{Cost of operation where i is exact power of 2}} \\ &= 2^0 + 2^1 + ... + 2^{log_2(n)} \\ &= \frac{2.(2^{log_2(n)} - 1)}{2 - 1} \quad (\textit{using sum of Geometric Progression}) \\ &= 2.(n - 1) \end{aligned}$$

$$\underline{\text{Total Cost}} = n - log_2(n) + 2n - 2 = 3n - log_2(n) - 2$$

$$\begin{aligned} \underline{\text{Amortized Cost}} \\ &= \lim_{n \to \infty} \frac{3n - log_2(n) - 2}{n} \\ &= \lim_{n \to \infty} 3 - \lim_{n \to \infty} \frac{log_2(n) - 2}{n} \\ &= 3 \end{aligned}$$

Thus, amortized cost per operation is 3.

Question 6

We have argued in the lecture that if the table size is doubled when it's full, then the amortized cost per insert is acceptable. Fred Hacker claims that this consumes too much space. He wants to try to increase the size with every insert by just two over the previous size. What is the amortized cost per insertion in Fred's table?

Answer

Using Fred Hackers Method we will increase the size of table by 2 with every insert. Lets visualize cost of insertions/copy for Fred's table for first few insertions.

Insert	Old Size	New Size	Copy
1	1	3	0
2	3	5	1
3	5	7	2
4	7	9	3
5	9	11	4
6	11	13	5
7	13	15	6
8	15	17	7
9	17	19	8

Lets generalize the pattern, assume we start with an array of size 1 and make n inserts.

$$\underline{\text{Cost of n Insertions}} = n$$

$$\begin{aligned}\underline{\text{Cost of copy operations}} & \\ &= 1 + 2 + 3 + 4 + \ldots + n - 1 \\ &= \frac{n.(n+1)}{2} - n\end{aligned}$$

$$\underline{\text{Total Cost}} = n + \frac{n.(n+1)}{2} - n = \frac{n.(n+1)}{2}$$

$$\underline{\text{Amortized Cost}} = \frac{n.(n+1)}{2n} = \frac{n+1}{2}$$

Thus, amortized cost per insertion in Fred's table is $\mathcal{O}(n)$.

Question 7

You are given a weighted graph G, two designated vertices s and t. Your goal is to find a path from s to t in which the minimum edge weight is maximized i.e. if there are two paths with weights 10→1→5 and 2→7→3 then the second path is considered better since the minimum weight. (2) is greater than the minimum weight of the first (1). Describe an efficient algorithm to solve this problem and show its complexity.

Answer

Algorithm Explanation

Here we will form a maximum spanning tree with the help of Disjoint-Set data structure and kruskals algorithm which will give us a tree connecting all vertices using all possible maximum weight edge. Later we can use BFS to find a path between s and t.

The path that we will get from BFS has maximized minimum edge weight. Because the edges which could also help connecting s and t using lesser weight edge has been discarded by Kruskals algorithm.

Steps:

1. Find Maximum Spanning Tree using Disjoint-Set and Kruskals Algorithm
2. Run BFS from start node to reach destination, the path obtained has maximized minimum edge weight.

Pseudo Code

```
graph <- list of edge with their weights
spanning_tree <- initialize empty list

#implementation of disjoint-set to detect cycles for Kruskals algorithm
rank <- initialize map to store rank (initially 0) of every node
leader <- initialize map to store leader (initially -1) of every node

#finds leader of given vertex, uses path compression and rank
function find(x)
    ldr = leader[x]
    while ldr != -1:
        ldr = find(ldr)
        leader[x] = ldr
    end while
    return ldr

#makes union of 2 sets
function union(x,y)
    ldr1 = find(x)
    ldr2 = find(y)
    if ldr1 == ldr2
        do nothing
    else if rank[ldr1] > rank[ldr2]
        leader[ldr2] = ldr1
    else if rank[ldr2] > rank[ldr1]
        leader[ldr1] = ldr2
    else
        leader[ldr1] = ldr2
        rank[ldr2] = 1+rank[ldr2]
    end if
```

```
#graph sorted with weights in descending order
graph <- sorted(graph, reverse = True)

#Kruskals Algorithm
for edge in graph
    start <- edge.source
    end <- edge.destination
    weight <- edge.weight

    if find(start) == find(end)
        do nothing, start and end belong are already connected in spanning tree
    else
        union(start, end)
        add edge to spanning_tree
    end if

source <- given in problem
destination <- given in problem
queue <- initialize a queue with only start source vertex

visited <- initialize a maps that keeps track of visited vertex
visited[source] <- True

path <- maps each node to list of nodes to traverse from source
path[source] <- [source]

while queue not empty
    current <- queue.pop()
    for child in spanning_tree with current as one end of edge
        if child == destination
            #this path will have maximized minimum edge weight
            print( path[current]+child )
        else if not visited[child]:
            queue.add( child )
            visited[child] <- True
            path[child] <- path[current]+child
        end if
end while
```

Run Time Complexity

1. Initializing rank and leader maps for Disjoint-Set : $\mathcal{O}(V)$.
2. Operations in Disjoint-Set using path compression and rank is $\mathcal{O}(\alpha(V))$ where α is Inverse-Ackerman function.
3. Sorting edges by weight in descending order is $\mathcal{O}(ElogE)$.

4. Running Kruskals Algorithm $\mathcal{O}(ElogE + \alpha(V)E)$.
5. Running BFS to find path between source and destination $\mathcal{O}(E + V)$
6. **Overall complexity :** $\mathcal{O}(V + E + V + ElogE + \alpha(V)E)$**. After ignoring non-dominating terms it is** $\mathcal{O}(ElogE) = \mathcal{O}(ElogV)$