CSCE 586 - Design and Analysis of Algorithms
Date: Wednesday 17th October, 2018
Name: Micah Hayden
Homework #2
Documentation:

Chapter 3, Problem #5

Problem Statement:

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

Inductive Proof:

Let n be the number of nodes in a tree, s be the number of leaves, and p be the number of nodes with two children.
Prove: $p = s - 1$ for a binary tree.
Base Case: $n = 1$

If $n = 1$, there is a single node, which is a leaf. It has no parent nodes. Thus, $s = 1$, $p = 0$

$$\begin{aligned} p &= s - 1 \\ 0 &= 1 - 1 \\ 0 &= 0 \end{aligned}$$

Inductive Hypothesis: $p = s - 1$ for $n \geq 1$
Inductive Step: Let us consider the insertion of a new node x. After insertion, x has parent node y. Because a node in a binary tree can have at most 2 children, there are only two cases for y.

1. y has no children
2. y has one child

Let s and p be the number of leaves, and the number of parent nodes with two children, respectively. Similarly, let s' and p' be the number of leaves and parent nodes with two children after the insertion of x.

Case #1: y has no children, y is a leaf node. x is inserted as a new leaf, and y becomes a node with one child. Thus, $p' = p$, and $s' = s$. Thus, anytime a node is inserted to a leaf node,

$$p' = s' - 1 \rightarrow p = s = 1$$

Case #2: y has one child, y's child is a leaf node. x is inserted as a leaf, thus $s' = s + 1$. Because y had one child prior to x, after x is inserted, y has two children. Thus, $p' = p + 1$.

$$\begin{aligned} p' &= s' - 1 \\ (p + 1) &= (s + 1) - 1 \\ p &= s - 1 \end{aligned}$$

The theorem is proved by induction because any time a node is inserted, the statement $p = s - 1$ holds true. Because the statement is true for $n = 1$, it will remain true for any $n' = n + 1$.
Difficulty: 3
Time: 30 Minutes

Chapter 4, Problem # 4

Problem Statement:

Develop an algorithm that takes two sequences of events - S' of length m and S of length n, each possibly containing an event more than once, and decides in time $O(m+n)$ whether S' is a subsequence of S.

Description of Algorithm:

This algorithm will loop through the items in S and determine whether S' is a subsequence of S. S and S' are implemented as two queues, with $\mid S \mid = n$ and $\mid S' \mid = m$.

The algorithm is as follows:

Algorithm 1 Subsequence Determination

Let queues A and B represent S' and S, respectively
while $A \neq \{\}$ and $B \neq \{\}$ **do**:
  $A' = front[A]$
  $B' = dequeue[B]$
  if $A' = B'$ **then**:
    A' has occurred in B
    $dequeue[A]$ ▷ This removes A' from queue A
  else
    A' has not occurred not in B
  end if
end while
if $A = \{\}$ **then**:
  S' is a subsequence of S
else:
  S' is not a subsequence of S.
end if

Asymptotic Analysis:

Worst Case:
The worst case scenario would be when the algorithm exits the while loop after n iterations, discovering either that S' is a subsequence, sharing the final event with S; or that S' is not a subsequence of S.
The while loop terminates in **at most** n loops because $n \geq m$. All other operations occur in constant time. Thus, the total run time of the algorithm is below:

$$O(n$$

Best Case:
The best case would occur when S' represents the first m events of set S. This would have the following run time:

$$\Omega(m)$$

Average Case:
In this particular algorithm, the average case would occur directly between the best case and worst case, it would terminate after $\frac{n}{2}$ iterations. This would indicate that the algorithm discovered that S' was a subsequence of S halfway through S. Its run time would be $O(m+n+\frac{n}{2}) = O(m+n)$, the same as the best/worst case scenarios. The determination of the average case is arbitrary in this example, again reiterating the strictness of the bound $\Theta(m+n)$.

Note: The problem statement requested an algorithm of run-time $O(m+n)$. That asymptotic growth is linear and can be achieved with the algorithm I have presented if one simply changes the input model (ie. chronologically reverse

the items in S' and S and utilize a stack, popping items A' and B' for the comparison, and pushing A' onto stack A if $A' \neq B'$. Reversing S and S' occurs in m and n operations, respectively; and the algorithm will terminate after at most m loops.

$$\begin{aligned} &= O(\,(m\,+\,n)\,+\,m \\ &= O(2 \cdot m\,+\,n) \\ &= \boxed{O(m+n)} \end{aligned}$$

This realization would satisfy the requirements of the problem, yet the method presented is a more efficient realization because it does not require sorting.

Proof:

Let there be two sequences S' and S, of lengths m and n, respectively, such that $m \leq n$. Let there be a queue A representing S', a queue B representing S, and A' representing the first event of A. The events in the queues represent the chronological ordering of events in S' and S, when an event is dequeued from the queue, it is the next chronological step.

If A' does not match the event dequeued from B, A' is not dequeued from A.
Thus, the events in A can only be processed in order, regardless of repetition of events.
Upon termination: if A is empty, all events A matched, in order, corresponding events in B.
If A is not empty, some event A' (or events), were not found in order.

By selecting the queue data structure, this algorithm's function is incredibly clear: there is no way to process the events in A out of order, and the algorithm terminates when either A or B is empty.

Difficulty: 5
Time: 30 Minutes

Chapter 4, Problem # 19

Problem Statement:

Develop an algorithm constructing a spanning tree T in which, for each $u, v \in V$, the bottleneck rate of the $u - v$ path in T is equal to the best achievable bottleneck rate for the pair u, v in G.

The bottleneck rate $b(p)$ is the minimum bandwidth of any edge on P. Thus, the goal is to maximize the bandwidth of the edges (one could think of this as a maximum-spanning tree).

Description of Algorithm:

Because the goal of this algorithm is to maximize the bandwidth, the weight of edge $e \in E = \frac{1}{b_e}$. Thus, the edges of the minimum weight have the maximum bandwidth, and then a Minimum Spanning Tree (MST) algorithm can be used.

I will utilize Kruskal's algorithm. Kruskal's algorithm begins with the set of edges E, and inserts these edges in order of increasing cost. Each edge e is added in this order, as long as it does not create a cycle. If e creates a cycle, it is discarded and the algorithm moves on. Through the definition of cost (weight), increasing cost equals decreasing bandwidth.

This algorithm will utilize the **Union-Find** data structure, which supports the following three applications [1]

1. **MakeUnionFind(**S**)** for a set S will return a **Union-Find** data structure on set S where all elements are in separate sets.
2. **Find(**u**)** will return the name of the set containing u for an element $u \in S$
3. **Union(**A**,** B**)** will change the data structure by merging sets A and B into a single set

Algorithm 2 Maximizing Bandwidth with Kruskal's Algorithm

SORT m edges by weight so that $c_{e_1} \leq c_{e_2} \leq ... \leq c_{e_m}$
$T \leftarrow \emptyset$
for each $v \in V$: **do**
    MAKE-SET(v)
end for
for $i = 1$ **to** m **do**
    $(u, v) \leftarrow e_i$
    if FIND-SET(u) $\neq$ FIND-SET(v) **then**
        $T \leftarrow T \cup e_i$
        UNION(u, v)
    end if
end for
Return T

Asymptotic Analysis:

Proof:

Kruskal's Algorithm produces a minimum spanning tree of G: Consider an edge $e = (v, w)$ added by Kruskal's Algorithm. To be considered, $v \in S$ and $w \notin S$, otherwise e would create a cycle in S. Similarly, edges from S to $V - S$ have been considered yet, otherwise they would have been added without creating a cycle. Thus, e is the cheapest edge with one end in S and another in $V - S$. This concludes that e must belong to every minimum spanning tree.
Let (V, T) be the output of Kruskal's Algorithm. (V, T) contains no cycles because the algorithm explicitly refuses cycles. If (V, T) was not connected, there would exist some set $S' \subset V$ with no edge e' from S' to $V - S'$. This contradicts the behavior of the algorithm because we know that G is connected, and thus there exists at least one edge between S and $V - S$. The algorithm would have added the first such edge that it encountered.
Difficulty: 3
Time: 30 Minutes

References