Analysis of Algorithms

Homework 4

Thomas Schollenberger (tss2344)

October 28, 2022

Question: 4

The expected case time complexity for the select algorithm is characterized by the following recurrence
$T(1) = 2$

$$T(n) = (n+1) + \frac{1}{n}\sum_{q=1}^{n-1} T(q)$$

Use techniques, including iteration, to solve the recurrence exactly.

Proof of 4: We know that:

$$T(n)| = (n+1) + \frac{1}{n}\sum_{q=1}^{n-1} T(q)$$

$$T(n+1)| = (n+2) + \frac{1}{n+1}\sum_{q=1}^{n} T(q)$$

From that definition, we can assert that:

$$n\,(T(n) - (n+1)) = \sum_{q=1}^{n-1} T(q)$$

Thus:

$$\begin{aligned}
T(n+1) - T(n)| &= (n+2) - (n+1) + \left(\frac{1}{n+1}\sum_{q=1}^{n} T(q) - \frac{1}{n}\sum_{q=1}^{n-1} T(q)\right) \\
| &= (n+2) - (n+1) + \left(\frac{1}{n+1}\sum_{q=1}^{n-1} T(q) - \frac{1}{n}\sum_{q=1}^{n-1} T(q)\right) + \frac{1}{n+1}T(n) \\
| &= 1 + \left(\frac{1}{n+1}\sum_{q=1}^{n-1} T(q) - \frac{1}{n}\sum_{q=1}^{n-1} T(q)\right) + \frac{1}{n+1}T(n) \\
| &= 1 + \left(\frac{1}{n+1} - \frac{1}{n}\right)\sum_{q=1}^{n-1} T(q) + \frac{1}{n+1}T(n) \\
| &= 1 - \frac{1}{n\,(n+1)}\sum_{q=1}^{n-1} T(q) + \frac{1}{n+1}T(n)
\end{aligned}$$

We can now substitute from the definition:

$$\begin{aligned}
T(n+1) - T(n)| &= 1 - \frac{1}{n\,(n+1)}\,(n(T(n) - (n+1)) + \frac{1}{n+1}T(n) \\
| &= 1 - \frac{T(n) - (n+1)}{(n+1)} + \frac{T(n)}{n+1} \\
| &= 1 - \left(\frac{T(n)}{n+1} - 1\right) + \frac{T(n)}{n+1} \\
| &= 1 + \frac{-T(n)}{n+1} + 1 + \frac{T(n)}{n+1} \\
| &= 2
\end{aligned}$$

Which means we end up with, after subtracting $-T(n)$:

$$\begin{aligned}
T(n+1)| &= T(n) + 2 \\
| &= 2n
\end{aligned}$$

Proving that our select algorithm runs in $O(n)$ time

□

Question: 5

a. CLRS 20.1-1
b. CLRS 20.1-8

Answer to 5a: Out-degree: $\theta(V + E)$, In-degree: $\theta(VE)$ □

Answer to 5b: The expected look up time to see if an edge is in the graph is $O(\frac{n}{m})$, where n is the number of edges in the graph and m is the size of the hash table. This is because, due to the uniform hashing algorithm used in the table, the edges would all hash to the same value and then be chained to their respective index. This would make the hash table function almost identically to a linked list.
A better solution would be a matrix. If we grouped our matrix by adjacent edges, this would have $O(1)$ look up time. The matrix would take up more space, but the speed-space tradeoff would be worth it. □

Question: 6

a. CLRS 20.2-2
b. Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result if line 18 (of the code in the book or line 19 of the code in the slides) were removed.
c. CLRS 20.2-5

Answer to 6a: d answers:
$d_s = 0$
$d_{r/v/u} = 1$
$d_{t/y/w} = 2$ $d_{x/z} = 3$
π answers:
$\pi_{r/u/v} = u$
$\pi_t = r$
$\pi_w = r$
$\pi_v = y$
$\pi_x = w$
$\pi_z = w$
$\pi_s = nil$ □

Answer to 6b: The purpose of the colors is to delineate what state the vertex is in. There is not real purpose, other than visual, as the colors just state whether or not the node is in the queue. This means that we could encode this as a bit, either one or zero, which would represent in or out of the queue □

Answer to 6c: Using the graph below, we can see that regardless of what order we come from ($s \rightarrow u \rightarrow t$ or $s \rightarrow r \rightarrow t$), the shortest distance to t is 1. This proves that the value $d[u]$ assigned to a vertex u is independent of the order in which the vertices appear in each adjacency list.
Using the graph below, we look at π and see that order matters. If we go $s \rightarrow u \rightarrow t$ then the π is u. However, if we go $s \rightarrow r \rightarrow t$ then the π is r. This proves that π is order dependent. □

Question: 7

CLRS 20.3-6

Answer to 7: Algorithm implemented below □

Algorithm 1: DFS Stack Implementation

```
   Data: G, u
 1 foreach v ∈ G.V do
 2     v.color ← WHITE;
 3     v.π ← NIL;
 4 time ← 0;
 5 foreach v ∈ G.V do
 6     if v.color = WHITE then
 7         S ← stack();
 8         push(S, u);
 9         while |S| ≠ 0 do Loop while the stack is not empty
10             x ← peak(S);
11             x.color ← GRAY;
12             found ← FALSE;
13             foreach v ∈ G.Adj[x] do
14                 if v.color = WHITE then If there is still a vertex not visited, push it and stop looping
15                     v.π ← x;
16                     v.color ← GRAY;
17                     time ← time + 1;
18                     push(S, v);
19                     found ← TRUE;
20                     break;
21             if found = FALSE then If all vertexes have been visited, this vertex itself is done
22                 time ← time + 1;
23                 x.f ← time;
24                 x.color ← BLACK;
25                 pop(S);
```