4005-800 Algorithms

Homework 3

Christopher Wood

April 11, 2012

PROBLEM 1. *CLRS 22.1-1*

Solution.

Given an adjacency list representation of a *directed* graph, the only way to determine the adjacent vertices of each vertex $v \in V$ is to traverse the entire adjacency list of v. Using this fact, we can easily determine the time complexity of computing the out-degree and in-degree of every vertex as follows.

1. To compute the out-degree for a single vertex $u \in V$, we must count the total amount of vertices contained within the adjacency list of u. To do this, we must traverse the the adjacency list for u, which amounts to traversing all outgoing edges starting from u as well. Therefore, in order to compute the out-degree of every vertex in a directed graph, we must repeat this procedure for every vertex, which means that we will traverse over every vertex and every edge in the graph. Thus, the time complexity is to compute the out-degree of every vertex is $\Theta(V + E)$.

2. To compute the in-degree for a single vertex $u \in V$, we must inspect all adjacency lists for every vertex $v \in V$ to determine if u is adjacent to v. Only after a complete traversal of the entire adjacency list representation can we be certain that we have examined all possible edges leading to u, and thus can compute the in-degree. A naive approach to extend this to all vertices would be to repeat this search procedure V times, amounting in a time complexity of $O(V(V + E))$. However, if we use an auxiliary data structure to keep track of the in-degree of every vertex $u \in V$, we need only perform this adjacency list traversal once, incrementing the in-degree of each vertex v that is visited in the traversal. Therefore, just as with the out-degree calculation, the time complexity is simply $\Theta(V + E)$.

PROBLEM 2-a. *CLRS 22.2-2*

Solution.

After running the breadth-first search on the undirected graph shown in Figure 22.3 in the textbook (using vertex u as the source) we arrive at the following values for d and π.

Vertex	d	π
r	4	s
s	3	w
t	1	u
u	0	NIL
v	5	r
w	2	t
x	1	u
y	1	u

Note that the steps for the BFS procedure were omitted because the textbook question did not explicitly ask for them; it simply asked for the d and π values upon termination.

PROBLEM 2-b. *CLRS 22.2-3*

Solution.

The purpose of the black color is to indicate that a vertex has been completely traversed and all of its neighbors have been discovered. By removing line 18 in the BFS algorithm, we limit the vertex colors to only white and gray, which means only a single bit is necessary to store this information. We now show that removing the black color does not change the results of the BFS procedure.

The initialization part of the BFS procedure (lines 1-9) does not change by removing the black color. Therefore, we must examine the main loop of the procedure, which is entirely dependent on how the queue Q is used, in order to determine if this modified version produces the same results as the original.

The results of the BFS procedure depend on the order in which vertices are visited in the breadth-first traversal, which ultimately depends on the order in which they are enqueued into Q. Based on the algorithm we know that vertices are only enqueued into Q when they are found to be colored white (i.e. not gray or black). By removing the black color we are not changing the result of this condition, since there is no distinction between black and gray vertices when they are enqueued (i.e. both gray and black are non-white). Therefore, the removal of the black color does not change the order in which vertices are enqueued into Q. Furthermore, since vertices are still colored gray when they are discovered, and all vertices are discovered in the same order, we can see that the removal of the black color does not change the breadth-first traversal of the BFS routine.

As an additional argument, one can see that the algorithm does not explicitly depend on a vertex being colored black (i.e. there is no conditional statement in the algorithm that hinges on whether a vertex is gray or black). Therefore, by removing the color black, we are not changing the control flow of the algorithm in any way. Thus, we can conclude that removing the color black does not change Q and in effect change the breadth-first traversal of a graph, nor does it change the control flow of the algorithm, so it must therefore produce the same result.

PROBLEM 2-c. *CLRS 22.2-5*

Solution.

In the correctness proof provided in the textbook, it is shown that $u.d = \delta(s, u)$ upon termination of the BFS procedure, meaning that $u.d$ is always equal to the length of the shortest path between s and u after running the BFS procedure. Since there are no assumptions made about the order of a vertex's adjacency list in the BFS algorithm, this means that the BFS procedure will always produce the shortest path lengths between a start vertex s and all other vertices $u \in V$ for any graph G, despite the order of its vertex adjacency lists. This is intuitively true since the order of vertices in the adjacency list representation of a graph G does not have any effect on the topolgy of G (i.e. the actual edges that exist in the graph), and therefore does not have any effect on the path lengths in G. Therefore, we can conclude that the value $u.d$ assigned to a vertex u is independent of the order in which the vertices appear in each adjacency list for G.

In Figure 22.3 from the textbook, we see that t must precede x in the adjacency list for w because the edge (t, u) exists in the resulting breadth-first tree (t and x share the common neighbor of u, but u was visited through t because t was enqueued into Q and visited before x). However, if we swap the position of t and x in the adjacency list for w, a BFS traversal will yield the edge (x, u), rather than (t, u), which will result in a different different breadth-first tree. This difference occurs because the vertices adjacent to x will be enqueued in Q before the vertices adjacent to t because x is visited first from the adjacency list of w. Therefore, the predecessor (π) of u will be x, not t, and we conclude that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

PROBLEM 3. *CLRS 22.3-7*

Solution.

The code for the DFS algorithm that uses a stack for its depth-first traversal is shown below. It uses three stacks to keep track of the recursive, depth-first traversal of vertices that are discovered (SD), the path of vertices that need to be finished by backtracking (SF), and a temporary stack used to push adjacent vertices onto the main discovery stack in the correct order (TS).

ALGORITHM 1: DFS(G)

1: **for** each vertex $u \in G.V$ **do** ▷ Initialization
2: $u.color = WHITE$
3: $u.\pi = NIL$
4: **end for**
5: $time = 0$
6: $SD = makeStack()$ ▷ Maintains depth-first traversal of the graph
7: $SF = makeStack()$ ▷ Maintains backtracking of graph (for finish times)

```
8: TS = makeStack()                          ▷ Temporary stack to reverse adjacency list order
9: for each vertex u ∈ G.V do
10:     if u.color == WHITE then
11:         Push(SD, u)                      ▷ Ensure that all vertices are visited
12:         while not isEmpty(SD) do
13:             v = Top(SD)
14:             Pop(SD)
15:             if v.color == WHITE then
16:                 time = time + 1
17:                 v.d = time
18:                 v.color = GRAY
19:                 vCount = 0               ▷ The number of new vertices that are discovered
20:                 for each vertex w ∈ G.Adj[v] do      ▷ Reverse the vertex order
21:                     Push(TS, w)
22:                 end for
23:                 while not isEmpty(TS) do
24:                     w = Top(TS)
25:                     Pop(TS)
26:                     if w.color == WHITE then         ▷ Only push undiscovered vertices
27:                         Push(SD, w)
28:                         w.π = v
29:                         vCount = vCount + 1
30:                     end if
31:                 end while
32:                 if vCount == 0 then      ▷ Backtrack using SF if we reached a dead-end
33:                     v1 = v
34:                     done = Size(SF) == 0
35:                     while not done do
36:                         v2 = Top(SF)
37:                         if v1 ≠ v2 then
38:                             time = time + 1
39:                             v1.f = time
40:                             v1.color = BLACK
41:                             v1 = v2
42:                             Pop(SF)
43:                             if Size(SF) == 0 then
44:                                 done = True
45:                             else
```

```
                        v_2 = Top(SF)
                    end if
                else
                    done = True
                end if
            end while
        else                                  ▷ Append multiple copies of v for backtracking
            for i = 1 → vCount do
                Push(SF, v)
            end for
        end if
    end if
    time = time + 1
    u.f = time
    u.color = BLACK
  end if
end for
```

PROBLEM 4-a. $T(1) = 1, T(n) = aT(n-1) + bn$

Solution.

To solve this recurrence relation using the iteration method, we first expand the recursive calls in order to identify a pattern, as shown below:

$$\begin{aligned}
T(n) &= aT(n-1) + bn \\
&= a(aT(n-2) + b(n-1)) + bn \\
&= a^2T(n-2) + ab(n-1) + bn \\
&= a^2(aT(n-3) + b(n-2)) + ab(n-1) + bn \\
&= a^3T(n-3) + a^2b(n-2) + ab(n-1) + bn \\
&= ... \\
&= a^kT(n-k) + a^{k-1}b(n-(k-1)) + ... + ab(n-1) + bn
\end{aligned}$$

Now, if we let $k = (n-1)$, we will reach the end of these recursive calls and end up with the following result:

$$
\begin{aligned}
T(n) &= a^{n-1}T(n-(n-1)) + a^{n-2}b(n-(n-2)) + ... + ab(n-1) + bn \\
&= a^{n-1} + b\sum_{i=0}^{n-2} a^i(n-i) \\
&= a^{n-1} + bn\sum_{i=0}^{n-2} a^i - b\sum_{i=0}^{n-2} ia^i \\
&= a^{n-1} + bn\left(\frac{a^{n-1}-1}{a-1}\right) - b\left(\frac{(n-2)a^n - (n-1)a^{n-1} + a}{(a-1)^2}\right) \\
&= a^{n-1} + \frac{bna^{n-1} - bn}{a-1} - \frac{b(n-2)a^n - b(n-1)a^{n-1} + ba}{(a-1)^2}
\end{aligned}
$$

Now, by discarding all lower order constant terms and simplifying, we can see that $T(n) = O(na^n)$.

PROBLEM 4-b. $T(1) = 1, T(n) = aT(n-1) + bn\ log(n)$

Solution.

To solve this recurrence relation using the iteration method, we first expand the recursive calls in order to identify a pattern, as shown below:

$$
\begin{aligned}
T(n) &= aT(n-1) + bn\ \log(n) \\
&= a(aT(n-2) + b(n-1)\ \log(n-1)) + bn\ \log(n) \\
&= a^2T(n-2) + ab(n-1)\ \log(n-1) + bn\ \log(n) \\
&= a^2(aT(n-3) + b(n-2)\ \log(n-2)) + ab(n-1)\ \log(n-1) + bn\ \log(n) \\
&= a^3T(n-3) + a^2b(n-2)\ \log(n-2) + ab(n-1)\ \log(n-1) + bn\ \log(n) \\
&= ... \\
&= a^kT(n-k) + a^{k-1}b(n-(k-1))\ \log(n-(k-1)) + ... + ab(n-1)\ \log(n-1) + bn\ \log(n)
\end{aligned}
$$

Now, if we let $k = (n-1)$, we will reach the end of these recursive calls and end up with the following result:

$$
\begin{aligned}
T(n) &= a^{n-1}T(n-(n-1)) + a^{n-2}b(n-(n-2))\ \log(n-(n-2)) + ... + ab(n-1)\ \log(n-1) + bn\ \log(n) \\
&= a^{n-1}T(1) + a^{n-2}b(n-(n-2))\ \log(n-(n-2)) + ... + ab(n-1)\ \log(n-1) + bn\ \log(n) \\
&= a^{n-1} + b\sum_{i=0}^{n-2} a^i(n-i)\ \log(n-i)
\end{aligned}
$$

Now we make the observation that $a^n(\frac{1}{a})^{n-i} = a^i$, so we can re-write the summation above as $a^n \sum_{i=0}^{n-2}(\frac{1}{a})^{n-i}(n-i)\log(n-i)$, which is less than $a^n \sum_{i=0}^{n-2}(n-i)\log(n-i)$. Furthermore, we make the observation that $\sum_{i=0}^{n-2} \log(n-i) < \sum_{i=0}^{n-2} \log(n)$, which means we that $a^n \sum_{i=0}^{n-2}(n-i)\log(n-i) < a^n \log(n)\sum_{i=0}^{n-2}(n-i)$. We now have the following:

$$\begin{aligned} T(n) < a^{n-1} + a^n b \log(n) \sum_{i=0}^{n-2}(n-i) &= a^{n-1} + a^n b \log(n)\Big(\frac{1}{2}(n-1)(n+2)\Big) \\ &= a^{n-1} + \frac{a^n b \log(n)}{2}(n^2+n-2) \end{aligned}$$

Therefore, by discarding constant and lower order terms in this expression for $T(n)$, we can conclude that $T(n) = O(a^n \log(n) n^2)$.

PROBLEM 4-c. $T(1) = 1, T(n) = aT(n-1) + bn^c$

Solution.

To solve this recurrence relation using the iteration method, we first expand the recursive calls in order to identify a pattern, as shown below:

$$\begin{aligned} T(n) &= aT(n-1) + bn^c \\ &= a(aT(n-2) + b(n-1)^c) + bn^c \\ &= a^2T(n-2) + ab(n-1)^c + bn^c \\ &= a^2(aT(n-3) + b(n-2)^c) + ab(n-1)^c + bn^c \\ &= a^3T(n-3) + a^2b(n-2)^c + ab(n-1)^c + bn^c \\ &= ... \\ &= a^kT(n-k) + a^{k-1}b(n-(k-1))^c + ... + ab(n-1)^c + bn^c \end{aligned}$$

Now, if we let $k = (n-1)$, we will reach the end of these recursive calls and end up with the following result:

$$\begin{aligned} T(n) &= a^{n-1}T(n-(n-1)) + a^{n-2}b(n-(n-2))^c + ... + ab(n-1)^c + bn^c \\ &= a^{n-1}T(1) + a^{n-2}b(n-(n-2))^c + ... + ab(n-1)^c + bn^c \\ &= a^{n-1} + b\sum_{i=0}^{n-2} a^i(n-i)^c \end{aligned}$$

Now we make the observation that $a^n(\frac{1}{a})^{n-i} = a^i$, so we can re-write the summation above as

$a^n \sum_{i=0}^{n-2} (\frac{1}{a})^{n-i}(n-i)^c$, which is less than $a^n \sum_{i=0}^{n-2}(n-i)^c$. Furthermore, we make the observation that $\sum_{i=0}^{n-2}(n-i)^c < \sum_{i=0}^{n-2} n^c$, which means we that $a^n \sum_{i=0}^{n-2}(n-i)^c < a^n \sum_{i=0}^{n-2} n^c$. We now have the following:

$$\begin{aligned} T(n) < a^{n-1} + a^n b \sum_{i=0}^{n-2}(n)^c &= a^{n-1} + a^n b\Big((n-1)n^c\Big) \\ &= a^{n-1} + a^n b(n^{c+1} - n^c) \end{aligned}$$

Therefore, by discarding constants and lower order terms in this expression for $T(n)$, we can conclude that $T(n) = O(a^n n^{c+1})$.

PROBLEM 4-d. $T(n) = aT(n/2) + bn^c$

Solution.

To solve this recurrence relation using the iteration method, we assume that $n = 2^k$ and then continue to expand the recursive calls in order to identify a pattern, as shown below:

$$\begin{aligned} T(n) &= aT(n/2) + bn^c \\ &= a(aT(n/2^2) + b(n/2)^c) + bn^c \\ &= a^2T(n/2^2) + ab(n/2)^c + bn^c \\ &= a^2(aT(n/2^3) + b(n/2^2)^c) + ab(n/2)^c + bn^c \\ &= a^3T(n/2^3) + a^2b(n/2^2)^c + ab(n/2)^c + bn^c \\ &= ... \\ &= a^iT(n/2^i) + a^{i-1}b(n/2^{i-1})^c + ... + ab(n/2)^c + bn^c \end{aligned}$$

Now, if we let $i = k$, we will reach the end of these recursive calls and end up with the following result:

$$\begin{aligned} T(n) &= a^kT(n/2^k) + a^{k-1}b(n/2^{k-1})^c + ... + ab(n/2)^c + bn^c \\ &= a^kT(1) + a^{k-1}b(n/2^{k-1})^c + ... + ab(n/2)^c + bn^c \\ &= a^k + b\sum_{j=0}^{k-1} a^j(n/2^j)^c \end{aligned}$$

By letting $n = 2^k$, we can translate this result into the following:

$$
\begin{aligned}
T(2^k) &= a^k + b\sum_{j=0}^{k-1} a^j 2^{(k-j)c} \\
&= a^k + b\sum_{j=0}^{k-1} a^j 2^{kc} 2^{-jc} \\
&= a^k + b2^{kc}\sum_{j=0}^{k-1} a^j 2^{-jc} \\
&= a^k + b2^{kc}\sum_{j=0}^{k-1} (\frac{a}{2^c})^j \\
&= a^k + b2^{kc}\left(\frac{(\frac{a}{2^c})^k - 1}{(\frac{a}{2^c}) - 1}\right) \\
&= a^k + \frac{b2^{kc}(\frac{a}{2^c})^k - b2^{kc}}{(\frac{a}{2^c}) - 1}
\end{aligned}
$$

Now, by ignoring the constant denominator from the term above and replacing k with $\lg(n)$, we have the following:

$$
\begin{aligned}
T(n) \approx a^k + b2^{kc}(\frac{a}{2^c})^k - b2^{kc} &= a^{\lg(n)} + bn^c(\frac{a}{2^c})^{\lg(n)} - bn^c \\
&= n^{\lg(a)} + bn^c n^{\lg(\frac{a}{2^c})} - bn^c \\
&= n^{\lg(a)} + bn^c n^{\lg(a) - \lg(2^c)} - bn^c \\
&= n^{\lg(a)} + bn^c n^{\lg(a)-c} - bn^c \\
&= n^{\lg(a)} + bn^{\lg(a)} - bn^c \\
&= (b+1)n^{\lg(a)} - bn^c
\end{aligned}
$$

Now, by discarding constants, we can conclude that $T(n) = O(n^{\lg(a)} - n^c)$, since the dominating term depends on the constants a and c.

PROBLEM 5.

Solution.

The maximum element out of a set of 5 numbers can be found by iteratively applying the following equation.

$$
\max(a,b) = \frac{a+b}{2} + |\frac{a-b}{2}| = \frac{a+b+|a-b|}{2}
$$

The source code for the $max5$ routine that relies on this equation is shown below.

```
1  def max5(x1, x2, x3, x4, x5):
2          max1 = (x1 + x2 + abs(x1 - x2)) / 2
3          max2 = (max1 + x3 + abs(max1 - x3)) / 2
4          max3 = (max2 + x4 + abs(max2 - x4)) / 2
5          max4 = (max3 + x5 + abs(max3 - x5)) / 2
6          return max4
```