UCSD CSE 101 Section B00, Winter 2015 MIDTERM (SOLUTIONS)

February 5, 2015

Question	Points	Score
1	10	
2	10	
3	10	
4	10	
5	10	
TOTAL	50	

INSTRUCTIONS. Be clear and concise. Write your answers in the space provided. Use the backs of pages, and/or the scratch page at the end, for your scratchwork. All graphs are assumed to be simple (no loops or multiple edges). Good luck!

You may freely use or cite the following subroutines from class:

- dfs(G). This returns three arrays of size $|V|$: *pre*, *post*, and *cc*. If the graph has k connected components, then the *cc* array assigns each node a number in the range 1 to k
- bfs(G, s), dijkstra(G, l, s), bellman-ford(G, l, s). Each of these returns two arrays of size $|V|$: *dist* and *prev*.

QUESTION 1. True/False, Short Answer.

Briefly justify your answers.

(a) ***(2 points)*** **True or False: Depth-first search on a graph $G = (V, E)$ can be implemented to run in time $O((|V| + |E|)^3)$.**

True. DFS can be implemented to run in $O(|V| + |E|)$ time, which is $O(|V| + |E|)^3$.

(b) ***(4 points)*** **Give the big-O solution to the recurrence relation $T(n) = 27T(\frac{n}{3}) + O(log\ n)$.**

The answer is $O(n^3)$

Bottom-heavy recurrence => $O(27^{log_3 n)}$ subproblems at last level, each requiring $O(1)$ work => $O(27^{log_3 n)}) = O(n^{log_3 27)}) = O(n^3)$.

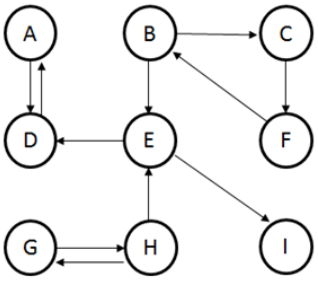
Alternate: $T(n)$ is "between" $A(n) = 27A(n/3) + 1$ (a=27, b=3, d=0) and $B(n) = 27B(n/3) + n$ (a=27, b=3, d=1). By M.T., $A(n) = O(n^3) \leq T(n) \leq B(n) = O(n^3)$.

(c) ***(4 points)*** **Give a recurrence relation for the running time $T(n)$ of the following pseudocode:**

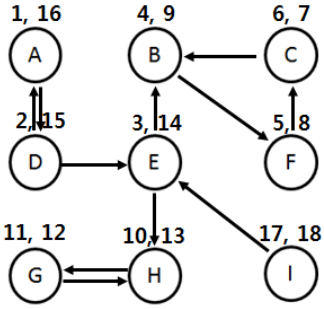
```
   define dqContains([a_1, ..., a_n], k)
1  if n == 1
2      return a_1 == k
3  return dqContains([a_1, ..., a_⌊n/2⌋], k) || dqContains([a_⌊n/2⌋+1, ..., a_n], k)
```

$$T(n) = 2T(n/2) + O(1)$$

QUESTION 2. DFS, SCCs.



Graph G



Graph G^R

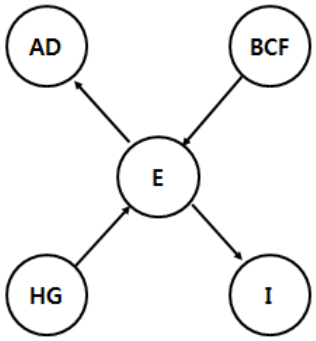
Refer to the graph G shown above.

(a) *(2 points)* In the right side of the figure, draw all edges of the reverse graph G^R, then execute depth-first search on G^R, starting at vertex A, writing *pre* and *post* labels next to each vertex. Break all ties lexicographically (i.e, according to alphabetical order).

(b) *(2 points)* List the strongly connected components (SCCs) of G, in the order that they are found using the algorithm given in class.

$$\{I\}, \{A, D\}, \{E\}, \{G, H\}, \{B, C, F\}$$

(c) *(2 points)* Draw the metagraph of strongly connected components (SCCs) for the above graph G.



(d) *(2 points)* List the source and sink SCCs of the graph G.

Source SCCs: $\{B, C, F\}, \{H, G\}$

Sink SCCs: $\{A, D\}, \{I\}$

(e) *(2 points)* The original graph G can be made strongly connected by adding edges. Determine a minimum-cardinality set of directed edges that, if added to the original graph G, would make the graph strongly connected. If there are multiple solutions, any will do.

Minimum cardinality set has 2 edges.

Either 2 edges from

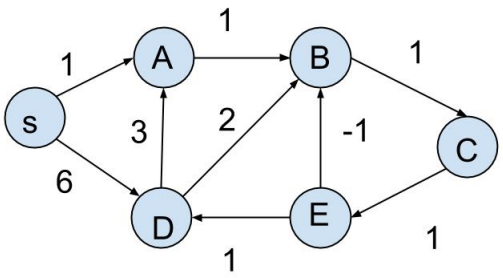
$\{I\} \times \{B, C, F\}$ **(1 edge)**
$\{A, D\} \times \{G, H\}$ **(1 edge)**

Or,

2 edges from
$\{A, D\} \times \{B, C, F\}$ **(1 edge)**
$\{I\} \times \{G, H\}$ **(1 edge)**

QUESTION 3. BELLMAN-FORD.

(a) *(6 points)* Suppose that the Bellman-Ford algorithm is executed on the graph shown below, with S as the source vertex. Fill in the table with the distance values of each vertex, at each iteration of the algorithm.



iteration	s	A	B	C	D	E
0	0	∞	∞	∞	∞	∞
1	0	1	∞	∞	6	∞
2	0	1	2	∞	6	∞
3	0	1	2	3	6	∞
4	0	1	2	3	6	4
5	0	1	2	3	5	4

(b) *(2 points)* If we continue to run Bellman-Ford on the given graph, will the stored *dist* values change between iterations 15 and 16? Answer Yes or No, and give a one-sentence justification.

No. The given graph has no negative cycles, so Bellman-Ford finds all true shortest-path costs after $|V| - 1$ iterations.

(c) *(2 points)* Suppose that an edge from E to A with weight -4 is added to the given graph. If we continue to run Bellman-Ford on the modified graph, will the stored *dist* values change between iterations 15 and 16? Answer Yes or No, and give a one-sentence justification.

Yes. Adding this edge would create a negative-cost cycle E-A-B-C-E.

Midterm NOTES on-screen clarified :
#3,b,c : If you were to continue running the Bellman-Ford iteration...

QUESTION 4. D/Q.

You are given a list of n intervals $[x_i, y_i]$, where x_i and y_i are integers with $x_i \leq y_i$. The interval $[x_i, y_i]$ represents the set of integers between x_i and y_i, inclusive. For instance, the interval $[3, 6]$ represents the set $\{3, 4, 5, 6\}$.

Define the *overlap* of two intervals I, I' to be $I \cap I'$, i.e., the cardinality of their intersection (the number of integers that are included in both intervals).

Devise an $O(n\ log\ n)$ D/Q algorithm that, given n intervals, finds and outputs a pair of intervals that have the maximum overlap. Hint: In HW2 #4, we split a set of intervals by their left endpoints.

(Note: you will not receive credit for the naive, non-D/Q algorithm that tests the overlap of each pair of intervals in $O(n^2)$ time.)

(a) *(3 points)* Describe your algorithm in English. Be very clear in describing your "merge" step.

We will use a D/Q algorithm to solve this problem. Similar to what we did in HW2 #4, we will begin by sorting the intervals by the left endpoint, breaking ties in ascending order of the right endpoint. Then, split the sorted list of intervals in half, and recursively call the subproblem on each half of the list. Recombine by finding the largest right endpoint in the first half list and comparing it with all of the intervals in the second half list.

(b) *(5 points)* Write down pseudocode of your algorithm.

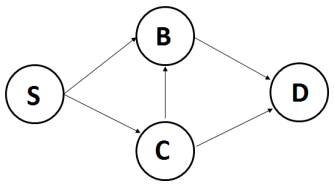
```
Set of intervals [s_1, ..., s_n]
Sort [s_1, ..., s_n] by increasing starting point
return intervals stored as bestIntervals at the end of
maxIntervalOverlap(sorted version of [s_1, ..., s_n])

   define maxIntervalOverlap([s_1, ..., s_n])
1  if n == 1
2   maxEndPtInterval[s_n] = s_n
3   result.maxOverlapIntervals = [] 4   result.maxOverlap = 0 5   return result
6  leftResult = maxIntervalOverlap([s_1, ..., s_{n/2}])
7  rightResult = maxIntervalOverlap([s_{n/2+1}, ..., s_n])
8  if rightResult.maxOverlap > leftResult.maxOverlap
9   bestResult = rightResult
10  else
11   bestResult = leftResult
12  for interval in [s_{n/2+1}, ..., s_n]
13   if overlap(maxEndPointInterval[s_1, ..., s_{n/2}], interval) > bestResult.maxOverlap
14   bestResult.maxOverlap = overlap(maxEndPointInterval[s_1, ..., s_{n/2}], interval)
15   bestResult.maxOverlapIntervals = [maxEndPointInterval[s_1, ..., s_{n/2}], interval]
16  if maxEndPtInterval[s_1, ..., s_{n/2}] > maxEndPtInterval[s_{n/2+1}, ..., s_n]
17   maxEndPtInterval[s_1, ..., s_n] = maxEndPtInterval[s_1, ..., s_{n/2}]
18  else
19   maxEndPtInterval[s_1, ..., s_n] = maxEndPtInterval[s_{n/2+1}, ..., s_n]
20  return bestResult
```

(c) *(2 points)* Give an analysis of your algorithm's runtime, $T(n)$. (As applicable, you may invoke the Master Theorem.)
We will break this problem up into two subproblems of half size. Recombining them will require comparing the maximum right endpoint of the first half list with every interval in the second half list. This will take $O(n)$ time. The recurrence relation is $T(n) = 2T(\frac{n}{2}) + O(n)$. By the Master Theorem, this takes $O(n\ log\ n)$ time.

QUESTION 5. PATH-COUNTING IN A DAG.

Given a directed acyclic graph (DAG) $G = (V, E)$ and a source vertex $s \in V$, design an efficient algorithm to obtain the number of distinct paths from s to all other vertices of V. [Hint: recall the discussion of finding longest / shortest paths in a DAG.]

In the example graph below, there are three simple paths from source vertex S to vertex D: $S \to B \to D$, $S \to C \to D$, $S \to C \to B \to D$.



(a) ***(3 points)*** **Give an English description of your algorithm.**

We describe two possible solutions to count all distinct paths in a DAG in $O(|V| + |E|)$: (i) modified BFS, and (ii) modified DAG-shortest-paths.

(i) Solution with modified BFS. We use an array count(u) to save the number of all distinct paths from s to u. We initialize count(s) to 1 and count(u) for all other vertices to 0. We linearize G and add the vertices in linearized order to a queue Q. When we dequeue u, for all edges (v, u), we count the number of distinct paths to a vertex u as count(u) $+=$ count(v) since the number of distinct path from s to u includes the sum of the number of distinct paths from s to its parent v.

(ii) Solution with modified DAG-shortest-paths. We linearize G and then execute DFS. For each vertex in the linearized order, for all edges (v, u), we count the number of distinct paths to a vertex u as count(u) $+=$ count(v). We initialize count(s) to 1 and count(u) for all other vertices to 0.

(b) ***(4 points)*** **Give pseudocode of your algorithm.**

Solution (i)

```
procedure BFS-distinct-paths(G, s)
Input:      DAG G = (V, E);
            vertex s ∈ V
Output:     For all vertices u reachable from s, count(u) is
            the number of all distinct paths from s to u.

Linearize G
for all u ∈ V, in linearized order:
   count(u) = 0
   Q.enqueue(u)
count(s) = 1
while Queue is not empty():
   u = Q.dequeue()
   for all edges (v, u) ∈ E:
     count(u) += count(v)
```

Solution (ii)

```
procedure DAG-distinct-paths(G, s)
Input:      DAG G = (V, E);
Output:     For all vertices u reachable from s, count(u) is
```

```
            the number of all distinct paths from s to u.

for all v ∈ V:
   count(u) = 0
count(s) = 1
Linearize G
for each u ∈ V, in linearized order:
   for all edges (v, u) ∈ E:
     count(u) += count(v)
```

(c) ***(3 points)*** **Analyze the runtime of your algorithm.**

Both (i) and (ii) visit every vertex and edge in $O(|V| + |E|)$.

For (i) and (ii), the running time to linearize G takes $O(|V| + |E|)$ and the two for-loops go through every vertex and connected edge in $O(|V| + |E|)$. Therefore, the running time for the algorithm is $O(|V| + |E|)$.

SCRATCH PAPER. DO NOT REMOVE.

```
procedure dag-shortest-paths(G, l, s)
Input:      DAG G = (V, E);
            edge lengths (l_e : e ∈ E); vertex s ∈ V
Output:     For all vertices u reachable from s, dist(u)
            is set to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
Linearize G
for each u ∈ V, in linearized order:
   for all edges (u, v) ∈ E:
     dist(v) = min{dist(v),dist(u) + l(u, v))}
```