UCSD CSE 101, Winter 2018 MIDTERM **VERSION: A**
Solutions

February 1, 2018

Question	Points
1	10
2	10
3	10
4	10
TOTAL	40

INSTRUCTIONS. Be clear and concise. Write your answers in the space provided. **Do not write your answers on the back of pages, since we will scan into Gradescope**. Use the scratch pages at the end and/or ask for extra sheets, for your scratchwork. Make sure to check your work. 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 G is undirected and has k connected components, then the *cc* array assigns each node a number in the range 1 to k. dfs(G) has runtime complexity $\mathcal{O}(|V| + |E|)$.
- bfs(G, s), dijkstra(G, l, s). Each of these returns two arrays of size $|V|$: *dist* and *prev*.
  bfs(G, s) has runtime complexity $\mathcal{O}(|V| + |E|)$.
  dijkstra(G, l, s) has runtime complexity $\mathcal{O}((|V| + |E|)log(|V|))$ assuming a binary heap PQ implementation.
  $\mathcal{O}(|V||E|)$.

Clarifications During EXAM:

DURING EXAM: CLARIFICATIONS

- Cover page: Disregard the extraneous O(|V||E|) at the bottom of the page (was for Bellman-Ford).
- **As a courtesy to your classmates, please do not leave during the last 10 minutes. Thank you. (Please keep your notes pages.)**
- Q1a: **big-O** (as with M.T. and class slides) is fine.
- Q1d: "are incident to the source vertex" → "are incident to, **and directed away from**, the source vertex" [as in the example]
- Q2a, 2b: Write the six vertex labels to show the order in which the vertices are **first** visited (DFS) or enqueued (BFS). **Only six letters should be written: A-_-_-_-_-_ .**
- [Q2d: "minimum-cardinality" = having smallest cardinality, i.e., number of elements.]
  - Q2d: This is a directed graph ...
- Q3: You just have to return ANY local min (or, max, depending on version).
  - If you don't know the peak-finding problem from before, don't worry. Your algorithm has to be different. (Q3 is not asking about peak-finding!)
- Q3: return the **index** of the local minimum (or, maximum) element that you find
- Q4a: your algorithm must run in linear time.
- **Q4b: Your modification of Dijkstra will obtain the required minimum gas tank size (needed to travel from s) for each vertex.**
- **Remember: backs of pages aren't scanned; use scratch**

QUESTION 1. Short Answer.

(a) ***(2 points)*** **Write down a recurrence relation for the running time $T(n)$ of the procedure $f(n)$. Then solve the recurrence using the Master Theorem. Briefly justify your answer.**

```
procedure f(n):
    if n == 1:
        print("end")
        return
    f(n/2)
    f(n/2)
    for (i = 1; i < n; i = i * 2):
        print("CSE 101")
```

Clarification During EXAM:

- Q1a: **big-O** (as with M.T. and class slides) is fine.
- Q1d: "are incident to the source vertex" → "are incident to, **and directed away from**, the source vertex" [as in the example]

Solution: The recurrence relation is $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(\log(n))$. We know that if $f(n) = \mathcal{O}(\log(n))$, then we also have $f(n) = \mathcal{O}(n)$. Following the approach used in HW2 review Problem 1, Example 4 (see HW2 review slides), upper-bounding as $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$ allows application of the Master Theorem with $a = 2, b = 2, d = 1, a = b^d$, giving $T(n) = \mathcal{O}(n^d \log(n)) = \mathcal{O}(n \log(n))$.
The $\log(n)$ function can also be upper-bounded by n^ϵ where $0 < \epsilon < 1$. This would yield $T(n) = \mathcal{O}(n)$ using the third case of the M.T. (Unrolling-based analyses can also lead to this result.) The alternate version of this question had three recursive calls to instances of size $n/3$, and the same bounds applied.

(b) ***(2 points)*** **Give a tight big-O bound for the runtime of the procedure $g(n)$. Briefly justify your answer.**

```
procedure g(n):
    for (i = 1; i < n; i = i * 2):
        for (j = 1; j < n; j = j + 2):
            for (k = n; k > 1; k = k / 3):
                print("CSE 101")
```

Let s_1 be the number of steps in the first loop (the outer one). Then, $i = 2^{s_1 - 1} < n$, $s_1 < \log_2(n) + 1$, $s_1 = \mathcal{O}(\log(n))$

Let s_2 be the number of steps in the second loop (the middle one). Then, $j = 2s_2 - 1 < n$, $s_2 < \frac{n+1}{2}$, $s_2 = \mathcal{O}(n)$

Let s_3 be the number of steps in the third loop (the inner one). Then, $k = \frac{n}{3^{s_3 - 1}} > 1$, $3^{s_3 - 1} < n$, $s_3 < \log_3(n) + 1$, $s_3 = \mathcal{O}(\log(n))$

Therefore, the desired bound is $\mathcal{O}(\log(n)) \cdot \mathcal{O}(n) \cdot \mathcal{O}(\log(n)) = \mathcal{O}(n \log^2(n))$.

(c) ***(3 points)*** **An arbitrary DAG $G = (V, E)$ is given as an adjacency list. Which of the following can be completed in linear time ($O(|V| + |E|)$)? You will get credit ONLY for marking all valid choices (and no other choices). CIRCLE EXACTLY ONE of the choices (yes | no) in each row.**

A. Reverse all edges of G

B. Identify all SCCs in G

C. Find all source and sink nodes in G

D. Topologically sort the vertices of G

A	**yes**	**no**
B	**yes**	**no**
C	**yes**	**no**
D	**yes**	**no**

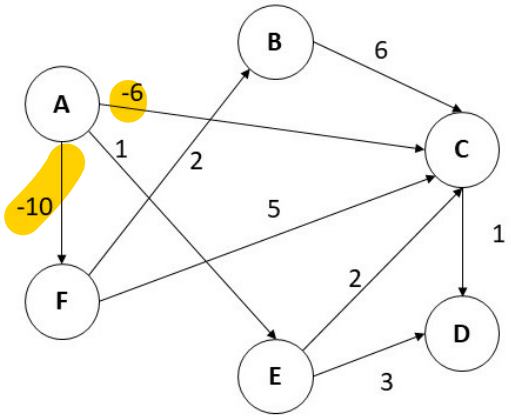
A - yes. Create a new adjacency list of size $|V|$. For each node v in G iterate through the nodes u in its adjacency list and instead of edge (v, u) add edges of type (u, v) to the new list.

B - yes. Construct a reversed graph G^R in linear time (see above). Rearrange the nodes in a decreasing order of their post numbers with the help of DFS in linear time. Run the algorithm that finds connected components in undirected graphs iterating through nodes in a decreasing order of their post numbers. (In a DAG, this will end up discovering that every node is its own SCC.)

C - yes. Have two arrays outdegree and indgree of size $|V|$. Iterate through all edges (u, v) in the adjacency list and increase values outdegree[u] and indegree[v] by one. Return all nodes with outdegree = 0 (sinks) and return all nodes with indegree = 0 (sources).

D - yes. Create an empty list. While you have an unvisited node v, start DFS from it. Add some node u to the list when DFS exits from that node (u gets popped from the stack). Reverse the list. It will be the topological order of the graph G.

(d) *(3 points)* **Explain why Dijkstra's algorithm still returns correct shortest paths to all reachable vertices if there are exactly two negative-weight edges in the graph, but both of these edges are incident to the source vertex. The figure below gives an example of such a graph where the source is A.**

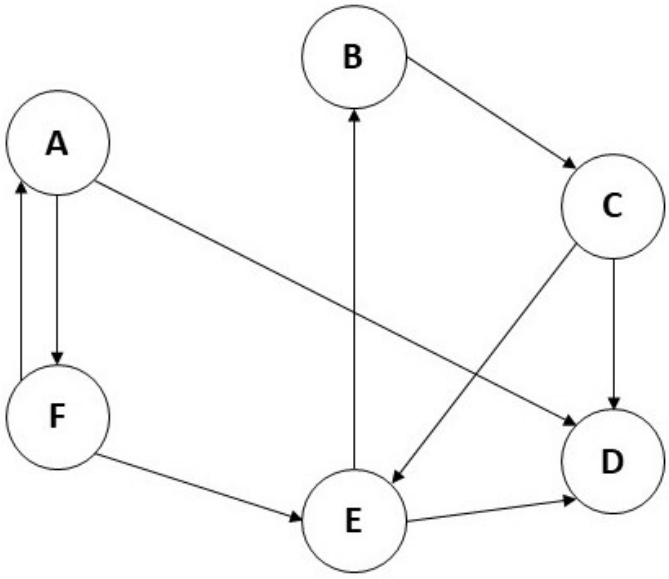


Consider the proof of Dijkstra's algorithm. The proof depended on the fact that if we know the shortest paths for a subset $S \subset V$ of vertices, and if (u, v) is an edge going out of S such that v has the minimum estimate of distance from s among the vertices in $V \setminus S$, then the shortest path to v consists of the (known) path to u and the edge (u, v). We can argue that this still holds even if the edges going out of the vertex s are allowed to be negative. Let (u, v) be the edge out of S as described above. For the sake of contradiction, assume that the path claimed above is not the shortest path to v.

Then there must be some other path from s to v which is shorter. Since $s \in S$ and $v \notin S$, there must be some edge (i, j) in this path such that $i \in S$ and $j \notin S$. But then, the distance from s to j along this path must be greater than that the estimate of v, since v had the minimum estimate. Also, the edges on the path between j and v must all have non-negative weights since the only negative edges are the ones out of s. Hence, the distance along this path from s to v must be greater than the the estimate of v, which leads to a contradiction.
(Note: a much simpler explanation of why negative weights only on edges outgoing from s don't lead to incorrect permanent labels (shortest-path distances from s) will receive full points. Essentially, the temporary labels of all vertices will correctly account for the negative-weight edges right from the start.)

QUESTION 2. DFS, BFS, SCC-Finding.

Refer to the directed graph G below, to answer the following questions. Remember that all ties are broken according to lexicographic order (alphabetically-lowest first).



Clarification During EXAM:

- Q2a, 2b: Write the six vertex labels to show the order in which the vertices are **first** visited (DFS) or enqueued (BFS). **Only six letters should be written: A-_-_-_-_-_ .**
- [Q2d: "minimum-cardinality" = having smallest cardinality, i.e., number of elements.]
  - Q2d: This is a directed graph ...

Solution:

(a) ***(3 points)*** **Execute depth-first search on the given graph starting at vertex A. List the vertices in the order they are visited.**

A, D, F, E, B, C

(b) ***(3 points)*** **Execute breadth-first search on the given graph starting at vertex A. List the vertices in the order they are visited.**

A, D, F, E, B, C

(c) ***(2 points)*** **How many strongly connected components does G have?**

There are three SCCs. (They are: {A, F}, {B, C, E}, {D}.)

(d) ***(2 points)*** **List a minimum-cardinality set of edges which, if added, will make G strongly connected.**
The set of edges will contain exactly one edge, either $\{(D, A)\}$ or $\{(D, F)\}$.

QUESTION 3. Algorithm Design: D/Q Local Maximum

Given an array of distinct integers $A = a_1, ..., a_n$, an element a_i is a *local maximum* if it is greater than its adjacent (i.e., neighboring) elements. In other words, if a_i has two neighbors and $a_{i-1} < a_i > a_{i+1}$, then a_i is a local maximum. Examples:

- **If $n = 1$, then a_1 is a local maximum.**
- **If the list is in increasing order, a_n is a local maximum.**
- **$A = [1, 3, 5, 7, 8, 10]$ - $a_6 = 10$ is the only local maximum in the array.**
- **$A = [1, 5, 3, 8, 7, 10]$ - each of $a_2 = 5, a_4 = 8$ and $a_6 = 10$ is a local maximum (return any one of them).**

Design an algorithm that, given an array $A = a_1, ..., a_n$, finds a local maximum in A. Your algorithm should have the same $O(\log n)$ runtime as the solution to the "peak finding" question from 2016's midterm, but your algorithm itself will be different. NOTE: A linear-time solution will receive NO CREDIT.
Clarification During EXAM During EXAM:

- Q3: You just have to return ANY local min (or, max, depending on version).
  - If you don't know the peak-finding problem from before, don't worry. Your algorithm has to be different. (Q3 is not asking about peak-finding!)
- Q3: return the **index** of the local minimum (or, maximum) element that you find

Solution:

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

We use a recursive approach. We find $a_{\frac{n}{2}}$ and compare it with its neighbors. We have the following cases which can be put in consecutive if statements. (Since all elements in A are distinct, all possible cases are considered.)

1. $n \leq 3$. In this case, the largest element in the array is the local maximum.
2. $a_{\frac{n}{2}-1} < a_{\frac{n}{2}} > a_{\frac{n}{2}+1}$. In this case, $a_{\frac{n}{2}}$ itself is a local maximum and we can return it.

$a_{\frac{n}{2}} < a_{\frac{n}{2}+1}$. In this case, there must be a local maximum somewhere to the right of $a_{\frac{n}{2}}$, so we recurse on $a_{\frac{n}{2}+1}, ..., a_n$.

4. $a_{\frac{n}{2}} > a_{\frac{n}{2}+1}$. In this case, there must be a local maximum somewhere to the left of $a_{\frac{n}{2}}$, so we recurse on $a_1, ..., a_{\frac{n}{2}-1}$.

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

Algorithm 1 Local Maximum

```
1: procedure LocalMax(A = a_1, ..., a_n)
2:     if n ≤ 3 then
3:         return max(A)
4:     end if
5:     mid = n/2
6:     if a_{n/2 - 1} < a_{n/2} > a_{n/2 + 1} then return a_{n/2}
7:     end if
8:     if a_{n/2} < a_{n/2 + 1} then return LocalMax(a_{n/2 + 1}, ..., a_n)
9:     end if
10:    if a_{n/2} > a_{n/2 + 1} then return LocalMax(a_1, ..., a_{n/2 - 1})
11:    end if
12: end procedure
```

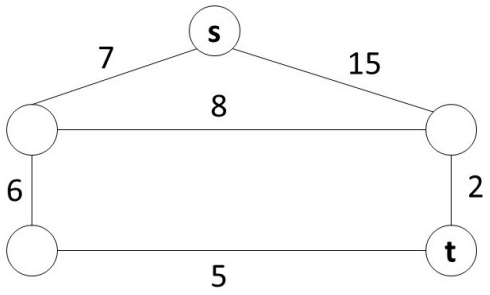
(c) ***(3 points)*** **Analyze the runtime of your algorithm.**

Solution 1: The recurrence for the runtime of LocalMax is $T(n) = T(n/2) + O(1)$. By the Master Theorem, $T(n) \in O(\log n)$.

Solution 2: At each step of the algorithm, we reduce the input size by half. In terms of runtime, this will be the same as binary search, so the runtime is $O(\log n)$.

QUESTION 4. Algorithm Design: Graph Search

You are given a set of cities, along with the pattern of highways between them, in the form of an undirected graph $G = (V, E)$. Each stretch of highway $e \in E$ connects two of the cities, and you know its length in miles, l_e. You want to get from city s to city t. There's one problem: your car can only hold enough gas to cover L miles. There are gas stations in each city, but not between cities. Therefore, you can only take a route if every one of its edges has length $l_e \leq L$. The example graph in the figure below is used to give examples for parts a and b.



Clarification During EXAM:

- Q4a: your algorithm must run in linear time.
- **Q4b: Your modification of Dijkstra will obtain the required minimum gas tank size (needed to travel from s) for each vertex.**

Solution: (a) *(6 points)* Given the limitation on your car's fuel tank capacity, show how to determine in linear time whether there is a feasible route from s to t. If $L = 5$ then the algorithm should return false when run on the example graph. If $L = 7$, the algorithm should return true when run on the example graph.
(i) *(4 points)* Give an English description of your algorithm.

This can be done by performing DFS from s ignoring edges of weight larger than L. For every edge from the current source node to its unvisited neighbour, if the edge weight is greater than L, we do not proceed with DFS further on this neighbour.

In the end, when DFS terminates, we either obtain a path from s to t which satisfies the requires fuel tank capacity, or none at all.

(ii) *(2 points)* Justify why your algorithm runs in linear time.

The algorithm performs DFS in the given graph, starting at s. Therefore, the algorithm runs in linear time.

(b) ***(4 points)*** **You are now planning to buy a new car, and you want to know the minimum fuel tank capacity that is needed to travel from s to t. In the example graph, the answer is 7.**

(i) ***(2 points)*** **An $O((|V|+|E|)\log|V|)$ algorithm to solve this problem is based on Dijkstra's algorithm. The pseudo-code for Dijkstra's algorithm is given below for reference. Please indicate the changes you will make to solve the given problem (indicate the line numbers and the proposed changes). It is possible to modify as few as 2 lines in a correct solution.**

```
1.  procedure dijkstra(G, Adj, s):
2.    input: Graph G = (V, E), directed or undirected;
             with positive edge lengths in Adj, and source vertex s
3.    output: For all vertices u reachable from s, dist(u) is set to the distance from s to u.

4.   for all  u in V:
5.      dist(u) = Inf
6.      prev(u) = nil
7.   dist(s) = 0

8.   H = makequeue(V) (using dist-values as keys)
9.   while H is not empty:
10.     u = deletemin(H)
11.     for all edges (u,v) in E:
12.        if dist(v) > dist(u) + Adj(u, v):
13.           dist(v) = dist(u) + Adj(u, v)
14.           prev(v) = u
15.           decreasekey(H, v)
```

Solution: It is sufficient to change the final loop to:

```
while H is not empty:
   u = deletemin(H)
   for all edges (u, v) in E:
     if dist(v) > max(dist(u), l(u, v))
         dist(v) = max(dist(u), l(u, v))
         prev(v) = u
         decreasekey(H,v)
```

(ii) ***(2 points)*** **Explain why the modification works correctly.**

We redefine the "length" of a path to be the largest edge weight in the path. Then, the "distance" from s to any other vertex t is the minimum possible "length" of any path from s to t.

Compare this with the original definition of distance, i.e. the minimum over all p of the sum of lengths of edges in p. This comparison suggests that by modifying the way distances are updated in Dijkstra we can produce a new version of the algorithm for the modified problem.

In the modified version of Dijkstra, for each vertex t we will find the *smallest* possible "bottleneck" (largest edge weight in a path that reaches t from s). This gives the minimum required gas tank size for reaching t from s.

When implemented with a binary heap, this algorithm achieves the required running time.

SCRATCH PAPER 1 of 2. DO NOT REMOVE.

SCRATCH PAPER 2 of 2. DO NOT REMOVE.