CS 170 Efficient Algorithms and Intractable Problems
Fall 2016 Christos Papadimitriou and Luca Trevisan

Midterm 1

Name: Joe Solutions

SID: 12345678

GSI and section time:

Write down the names of the students on your left and right as they appear on their SID.

Name of student on your left:

Name of student on your right:

Name of student behind you:

Name of student in front of you:

Instructions:

Write your name and SID on each sheet in the spaces provided.

You may consult one handwritten, double-sided sheet of notes. You may not consult other notes, textbooks, etc. Cell phones and other electronic devices are not permitted.

There are 6 questions. The last page is page 11, plus 3 blank pages for scratch.

Answer all questions. On questions asking for an algorithm, make sure to respond in the format we request. Write in the space provided. Good luck!

Note: If you finish in the last 15 minutes, please remain seated and do not leave early, to avoid distracting your fellow classmates.

Do not turn this page until an instructor tells you to do so.

1. (12 pts.) Fun with Recurrences and FFT

Write your answer in the box. You don't need to justify your answer.

(a) What is the solution of the recurrence $T(n) = 6T(n/3) + n^2$?
Using Master theorem with $a = 6$, $b = 3$, and $d = 2$, we see that $\log_3 6 < 2$, so the runtime is $\Theta(n^2)$

(b) Find a so that the solution of the recurrence $T(n) = aT(n/3) + n^2$ is $\Theta(n^3)$.
Using Master Theorem, we need $a = 27$ so that $\log_3 27 = 3 > 2$.

(c) What is the output of the FFT when the input is [0,-1,0,0]?
$A(1) = -1$; $A(i) = -i$; $A(-1) = 1$; $A(-i) = i$

(d) In a flash of insight, you realize the FFT algorithm would work just fine splitting each polynomial into three! Fill in the missing pieces of the modified algorithm, which should have the same asymptotic runtime as the FFT we've seen.

Algorithm 1 TRIFFT(polynomial $A(x)$ of degree $n-1$, with n a power of 3)

```
1: Set $\omega := e^{2\pi i/n}$
2: **if** $\omega = 1$ **then**
3:     **return** $A(1)$
4: Find polynomials $A_0$, $A_1$, and $A_2$ such that $A(x) :=$ ____________________
5: Call TRIFFT(($A_0, \omega^3$)), TRIFFT(($A_1, \omega^3$)), and TRIFFT(($A_2, \omega^3$)) to obtain $n/3$ evaluations of each
   of these polynomials.
6: For $j = 0$ to $n-1$:
7:     ____________________
8: **return** $A(\omega^0), A(\omega^1), \ldots, A(\omega^{n-1})$.
```

$n :=$ the minimum value of $3^i > d$
$A(x) := A_0(x^3) + xA_1(x^3) + x^2A_2(x^3)$
$A(\omega^j) = A_0(\omega^{3j}) + \omega^j A_1(\omega^{3j}) + \omega^{2j} A_2(\omega^{3j})$
Common Mistakes:

- Putting $A_0(x) + xA_1(x) + x^2A_2(x)$ in the first blank.
- Putting $A_0 + xA_1 + x^2A_2$ in the first blank. You received full credit for this answer only if you got the second blank correct.
- Putting $A(\omega^j) = A_0(\omega^j) + \omega^j A_1(\omega^j) + \omega^{2j} A_2(\omega^j)$ in the second blank.

2. (17 pts.) Divide and Conquer with Roots

Let's find a divide-and-conquer solution to the following problem.

Input: a list $[a_1, \ldots, a_n]$ of n numbers, representing the roots of a polynomial $P(x)$. Thus, $P(x) = (x - a_1) \cdot (x - a_2) \cdots (x - a_n)$.

Output: The list $[c_0, \ldots, c_{n-1}]$ of coefficients of the same polynomial: $P(x) = c_0 + c_1 x + \ldots + c_{n-1} x^{n-1}$

For example, if $n = 2$ and the input is $[1, -3]$ then the output would be $[-3, 2, 1]$, because $(x - 1) \cdot (x + 3) = -3 + 2x + x^2$.

(a) Write pseudocode to solve this problem. You do not need to write a main idea, or prove correctness.

Algorithm 2 ROOT2COEF($A[1..n]$)

```
1: if $n = 1$ then
2:     return $[-A[1], 1]$
3: $C_L :=$ ROOT2COEF($A[1..\lfloor n/2 \rfloor]$)
4: $C_R :=$ ROOT2COEF($A[\lfloor n/2 \rfloor + 1, n]$)
5: Use the FFT to multiply the polynomials with coefficients $C_L$ and $C_R$. Set $C :=$ the result's coefficients.
6: return $C$.
```

(b) Write a recurrence for your algorithm. *You don't need to justify your answer.*

$$T(n) = 2T(n/2) + \Theta(n \log n)$$

(c) Solve the recurrence: find the runtime of your algorithm.
Using Wikipedia's Master Theorem, the runtime is $\Theta(n \log^2 n)$

(d) **(1pt extra credit)**
$(x - a) \cdot (x - b) \cdots (x - z) =$
(*Hint:* Short answer expected.) 0, because of the $(x - x)$ term. Trollolloll!

3. (28 pts.) True/False

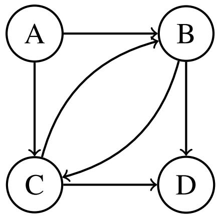
Write T or F in the box. Then, if the statement is true, justify briefly. If the statement is false, give a counterexample.

(a) $\boxed{}$ Every DAG has a source vertex and a sink vertex.

True; for example, the first vertex in linearized order is a source, and the last is a sink.

(b) $\boxed{}$ Every graph that has a source vertex and a sink vertex is a DAG.

False; consider the following graph:



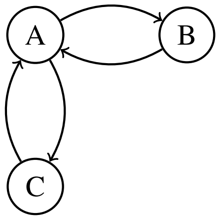
(c) $\boxed{}$ A directed graph is a DAG if and only if it is linearizable.

True; if a graph is a DAG, we know there must be a source vertex (see above), and we can find a linearized order by popping successive source vertices and their outgoing edges.

If a directed graph is not a DAG (has a cycle), we cannot linearize it because whichever vertex in the cycle goes first will have an edge to it from a future vertex.

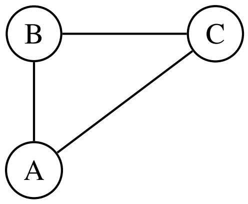
(d) If all cycles in a directed graph are of length 2 (they consist of two edges), then its SCCs have sizes of at most 2 vertices. (By *cycle* we mean a closed path that does not pass through a vertex twice).

False; consider the following graph:



(e) In an undirected, weighted graph, if Dijkstra and Prim remove vertices from the priority queue in the same order, then the shortest path tree and the MST are the same.

False; consider the following graph, where $d(A,B) = 3$, $d(B,C) = 4$, and $d(A,C) = 5$. Then, the shortest path tree will be (A,B) and (A,C), while MST will be (A,B) and (B,C).



(f) For any vertex in a weighted, undirected graph, the edge incident to that vertex that has smallest weight must be part of some MST of the graph.

True; this is a minimum edge across the cut between this vertex and the rest of the graph.

You can also justify this by starting Prim's at that vertex - we know Prim's is guaranteed to give us a valid MST, and Prim's will always pick the smallest edge out of the vertex. It is also possible to justify it with Kruskal's, but it is not as simple - you must argue that the first edge that Kruskal's processes on that vertex must be the smallest, and that it must be added to the graph (it cannot create cycles because it is the first edge with that vertex considered so far).

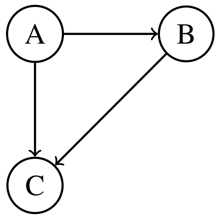
Many students attempted an exchange argument, claiming that by adding the lowest-weight edge to the graph, we would end with a lower-weight spanning tree. This is valid, but you can't just remove any edge from the vertex - you have to remove the (at least as large) edge on the vertex that is part of the cycle, or else you may not end up with a tree.

(g) $\square$ If the weights of the edges of a graph are all different, the MST is unique.

True; this was on the HW; there is only one sort of edges by increasing weight, so Kruskal's can only return one MST. Consider some other MST, and consider the first edge which is different between this MST and Kruskal's MST. If it's in this MST and not Kruskal's, contradiction (Kruskal's would have added it if it could). Else, it's in Kruskal's MST and not this one. We can then improve this MST by adding this edge and removing some other edge in the created cycle. Thus this was not an MST, contradiction.

(h) $\square$ If the weights of the edges of a graph are all different, the shortest path tree is unique.

False; consider the following graph, where $d(A,B) = 3$, $d(B,C) = 4$, and $d(A,C) = 7$. Then, the shortest path tree could be either (A,B) and (B,C), or (A,B) and (A,C).



(i) $\square$ If we add the same number to the weights of all edges of a graph, the shortest path tree remains the same.

False; consider the above graph, except $d(A,C) = 8$. The shortest path tree changes when we add 2 to each edge length.

(j) $\square$ If the output of the FFT is [0,0,0,0] in list form, then the input was necessarily also [0,0,0,0].
True; there are several ways to justify this. Running the inverse FFT function on input [0,0,0,0] verifies it. Also, the fact that the FFT is a bijective map means that the zero vector is the only input that would produce an output of zero (claiming that the columns of the FFT matrix are linearly independent is sufficient to show it is invertible). Finally, it also suffices to say just solve the linear system and find that the only answer is 0.

(k) $\square$ The MST is also the tree that minimizes the *maximum* weight of an edge on the tree.
Explanation 0: Consider the heaviest edge e' in some spanning tree. Removing it from the tree yields two connected components; consider the cut between the set of vertices in the first component and the second. If e' is not the lightest edge across this cut, this cannot be an MST (you could replace e' with a lighter edge e across the cut). Otherwise, e' is the lightest edge across this cut and thus some edge of at least this weight must be in any ST. Thus, the MST is choosing the lightest possible heaviest edge in the MST.

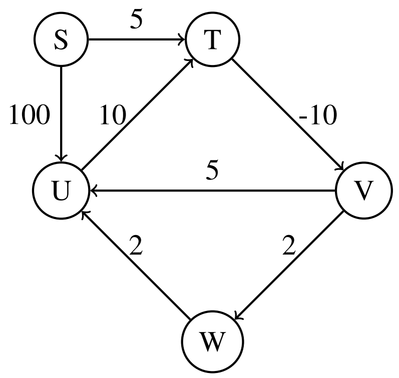
Explanation 1: Consider Kruskal's MST algorithm: it has the property that the ith edge it adds to the MST is the lightest it could possibly be, which is true for the first added edge (the very lightest in the graph) and then inductively (assuming the first $k-1$ added edges were as light as possible, the kth added edge is the lightest possible – the first which doesn't add a cycle). In particular, Kruskal's last added edge is as light as possible: thus Kruksal's minimizes the maximum weight of an edge in the tree.

Now, consider a spanning tree with an edge e' which is heavier than e, the heaviest edge Kruskal's adds. This spanning tree cannot be an MST because, comparing it to Kruskal's, its heaviest edge is heavier than Kruksal's, while all $|V|-2$ other edges are no lighter than Kruksal's other edges, which we showed were as light as possible. Thus this spanning tree cannot be an MST. Therefore, only spanning trees which minimize the maximum weight of an edge in the tree could be MSTs.

Many solutions were on the right track, but were too vague and received partial credit. *Common mistakes*

- ”MST minimizes the weights of the edges, so the largest edge is minimized as well”
- Only the first paragraph of Explanation 1 was not sufficient.
- Subtle: Consider a ST with an edge e of larger weight than the minimum possible max weight edge in the MST, and consider a vertex e connects to. Put this vertex on one side of the cut, now there must be a lighter edge e' across this cut, so we could add e' and take out e, lowering the total weight. Therefore the ST was not an MST, contradiction. *The flaw:* e' could have already been in ST; there's no guarantee that only one edge across a cut is in an MST.
- Consider an MST with a largest edge e larger than the minimum possible max edge. Then we can replace e with some smaller edge in the tree. But then we didn't have an MST in the first place: contradiction. *The flaw:* ”Then we can replace e with some smaller edge in the tree.” is too vague.

4. (12 pts.) Bellman-Ford Ordering



(a) Consider the directed graph above, with positive edge weights. Find the distances of all the vertices from S.

Vertex v	dist(v)
S	0
T	5
U	-1
V	-5
W	-3

(b) Say we run Bellman-Ford to find these distances, and we can choose the order in which the edges are updated (this order is the same in every iteration of updates). Give an ordering of the edges (formatted as a list like [ST, UT, VW]) for which all distances are correct after only one iteration of updates (after each edge has been updated only once).

You don't need to justify your answer.

Any ordering with ST, TV, VW, WU, and the other edges in arbitrary positions is correct.
One example is to update the edges in the same order as the shortest path: $[ST, TV, VW, WU, SU, UT, VU]$

(c) With the same situation as part (b), give an ordering of the edges for there is at least one distance which is only correct after the last iteration of updates. *You don't need to justify your answer.*

Any ordering with WU, VW, TV, ST and the other edges in arbitrary positions is correct.
One example is to update the edges in the opposite order as the shortest path: $[WU, VW, TV, ST, SU, UT, VU]$

5. (19 pts.) Shortest paths with tie-breaking

Let's modify Dijkstra's algorithm for shortest paths so when there are two or more shortest paths from the start vertex s to some vertex v, the algorithm returns the one that has the *fewest edges*.

For this question, we'll consider directed graphs with nonnegative integer edge lengths.

(a) To accomplish this, besides the `dist` array, we also maintain at each node v an integer `noe(v)` (the number of edges), initialized the same way as `dist`.

Modify the Dijkstra code fragment below, by editing the lines and/or adding new lines as needed, to accomplish this. After the algorithm runs, the `prev` pointers from v should indicate, in reverse, the shortest path from s to v with the fewest edges.

In the code fragment, `length(u,v)` is the length of edge (u,v). Q is the priority queue of vertices, and E is the set of edges.

Briefly explain your modifications.

```
while Q is not empty do:
    u = delete-min(Q)
    for all edges (u,v) in E:
        if dist(v) > dist(u) + length(u,v)
          or (dist(v) = dist(u) + length(u,v) and noe(v) > noe(u) + 1):
            dist(v) = dist(u) + length(u,v)
            noe(v) = noe(u) + 1
            prev(v) = u
```

When the new distance is tied with the old distance, break ties in favor of the path with the shortest number of segments. Whenever we're deciding to go to v via u, the number of edges in this path is just one more than the number of edges in the path to u.

(b) Now suppose the algorithm is on the cloud somewhere, pitifully undocumented, or in C++, and you cannot edit it. There is a way to solve this problem using the original Dijkstra code, by modifying the input to Dijkstra's algorithm.

The given graph (V, E, ℓ) has nonnegative integer edge lengths $\ell(u, v)$ for all edges $(u, v) \in E$. Describe how to generate a new input (V, E, ℓ'), where you keep the same graph but modify the weights, so that the output of Dijkstra's solves the shortest paths with tie-breaking problem on the original graph.

Briefly justify the correctness of your solution.

Keep $V' = V$. Set the edge lengths $(u, v) \in E'$ to $1/|V|$ more than the corresponding edge length $(u, v) \in E$. We know that there are at most $|V| - 1$ edges in the shortest path, so we are adding at most length $(|V| - 1)/|V|)$ to each path, which is not enough to affect relative lengths of paths which in G had different lengths (they had to differ in length by at least 1 given integral edge lengths). On the other hand, in case of ties, these additions make paths with fewer segments shorter.

Run Dikstra's on $G'(V', E')$, and the resulting shortest path is the desired shortest path on G (the vertices and set of edges are the same; only lengths of edges were altered).

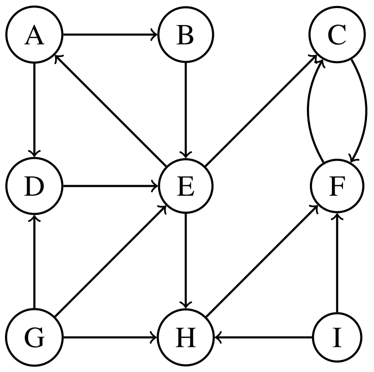
Common Mistakes related to 'adding a constant' (received partial credit) :
(a)Adding 1 to edges . (There were some solutions that did this correctly if you multiply the edges by V or E beforehand)
(b)More generally, adding a constant independent of V or E.
(c)Even more generally, adding a constant greater possibly greater than 1/ V or 1/E
(d)Adding a constant which depended on the 'min edge' (normally in the numerator of your constant). This doesn't work because the min edge can be very large. (e) Adding a constant which depended on the 'max length of a path' instead of E or V . In general, one cannot find the max length of the path (it is an NP-Complete problem, more on this later in the semester.)

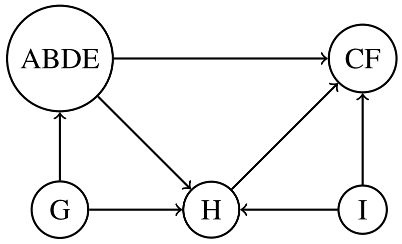
Other common mistakes:

(a) using noe(v) from the previous problem. The point of this question is that we don't have access to the Dijstra code, only the edges, vertices, and nodes.
(b) Trying to Run some kind of BFS or replacing all edges by 1 and running Djistra. In general, the shortest paths ignoring edge weights have very little relation to the shortest paths involving edge weights.
(c) scaling, square-rooting, log'ing the edges.

6. (12 pts.) Linearization and Beyond

(a) Draw the DAG of the strongly connected components of the directed graph below.





(b) How many sink SCCs are there in the above graph? *No justification necessary.*

Just 1, CF.

(c) Consider a directed graph G with multiple SCCs, and a single source SCC: give a short description for the number of edges needed to make the whole graph strongly connected.

Briefly justify your answer.

# edges = # of sink SCCs. Add an edge from an arbitrary vertex in each of them to the source. Alternatively, fix some sink SCC and add an edge from each of the other sink SCCs to this sink and then add an edge from the special sink to the source.

Justification: Look at the DAG of SCCs. It suffices to show that adding these edges makes this DAG strongly connected. Since there is only one source, there is a path from the source SCC to any other SCC. Any SCC has a path to some sink, and has a path to the source using one of the added edges. Therefore, for any SCC A and SCC B, there is a path from A to B by: $A \rightarrow$ a Sink $\rightarrow$ Source $\rightarrow B$, so the graph is strongly connected.

Common Mistakes:

Insufficient Justifications:

- Arguing that the number of edges is minimal because in order for a sink SCC to reach another SCC it must have an outgoing edge. This does show that the number of edges is minimal, provided that adding these edges makes the graph strongly connected, which is not shown.

Weak Justifications:

- Arguing that adding the edges creates multiple cycles within the graph.

- Intuitive/informal arguments.
- Vague arguments stating that there are now cycles in the DAG so the graph is strongly connected.

Incorrect Jusifications:

- Arguing that adding the edges creates one cycle containing all the SCCs. This is false.

Wrong answers:

- # edges = # of SCCs - 1
- # edges = 1
- Assuming that there is only one sink SCC.
- Stating that only one of the added edges needs to go the source, but the rest can go anywhere.
- Anything that mentions SCCs that are "disconnected" from the source. Since there is only one source, all SCCs must be reachable from that source.

Minor errors:

- Taking $\max(1, \#\text{ of sink SCCs})$. The max is unnecessary because there is guaranteed to be at least one sink SCC.

Blank Space

You can use this extra space for scratch work or continuing to answer a question.

Blank Space

You can use this extra space for scratch work or continuing to answer a question.

Blank Space

You can use this extra space for scratch work or continuing to answer a question.