Midterm 1 Solutions

Note: These solutions are not necessarily model answers. Rather, they are designed to be tutorial in nature, and sometimes contain a little more explanation than an ideal solution (especially in Q4). Also, bear in mind that there may be more than one correct solution. **You should read through these solutions carefully, even for problems that you answered correctly!** *The maximum total number of points available is 65.*

1. True or False?

(i) **False.** One possible counterexample is the following: *3pts*

$$f(n) = n^2; \qquad g(n) = \begin{cases} n & \text{if } n \text{ is odd;} \\ n^3 & \text{if } n \text{ is even.} \end{cases}$$

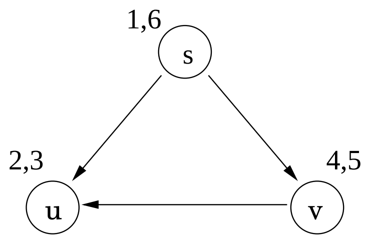
Then we have $\frac{f(n)}{g(n)} = n$ for all odd n, and so when n is large enough we cannot have $f(n) \leq Cg(n)$ for any constant C. Thus $f(n) \neq O(g(n)$. Similarly, considering even n we see that $g(n) \neq O(f(n))$. The key point here is that the definition of big-O requires that $f(n) \leq Cg(n)$ for *all* large enough n; by making the function $g(n)$ oscillate between values that are much bigger and much smaller than $f(n)$, we can ensure that neither $f(n) \leq Cg(n)$ nor $g(n) \leq Cf(n)$ holds as $n \to \infty$. Of course, to get this weird behavior we have to cook up a slightly strange function $g(n)$; for most functions that we encounter in the analysis of algorithms, this doesn't happen.

Not many people got this part right. The main bogus answer was to say "True, because either $f(n)$ has to grow at least as fast as $g(n)$ or vice versa." This and other incorrect answers did not work with the exact definition of big-O. Some people got close to a valid counterexample by trying to define two oscillating functions (e.g., $\sin(n)$ and $\cos(n)$); many of these attempts were correct in spirit, but sometimes fell down on details such as $f(n)$ and $g(n)$ not tending to infinity.

(ii) **True.** If $\mathrm{pre}(u) < \mathrm{post}(u) < \mathrm{pre}(v) < \mathrm{post}(v)$ then the DFS backtracks from u before vertex v is discovered. But if there were an edge between u and v then the search would have to explore this edge (and hence discover v) before backtracking from u. *3pts*

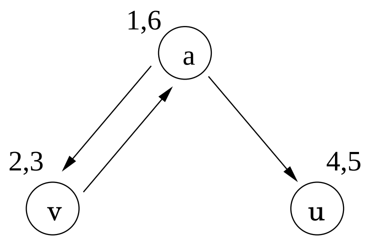
Most people got this right.

(iii) **False.** If $\mathrm{pre}(u) < \mathrm{post}(u) < \mathrm{pre}(v) < \mathrm{post}(v)$ then there could still be a directed edge from v to u; this would be a cross edge for the DFS. Here is a small example, showing the pre- and post-visit labels (the search starts at s): *3pts*



Most people got this right.

(iv) **False.** Let v be the vertex with lowest post-visit label. There could be a back edge from v to one of its ancestors a, but it is not necessary that v is reachable from every vertex that is reachable from a. Here is a simple counterexample: *3pts*

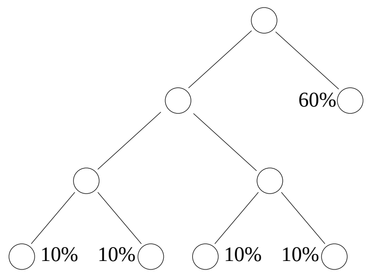


Many people got this wrong. The most common mistake was to claim "True" because the only vertices reachable from v are its ancestors, and each of them certainly has a path to v. This argument fails to realize that those ancestors may themselves be able to reach other vertices that are not ancestors of v, and indeed from which v is not reachable at all, such as u in the above example.

(v) **False.** The same example as in part (iv) works here: a is the vertex with highest post-visit label, and all vertices are reachable from a, but not all vertices are in the same SCC. *3pts*

Most people got this right.

(vi) **False.** Consider the tree below, which specifies a prefix-free code for an alphabet of five letters. This tree can arise (e.g.) from the following frequencies: 60%, 10%, 10%, 10%, 10%. *3pts*



The most common mistake here was to say "True because otherwise the Huffman tree would not be a full tree." But as the example above demonstrates, the tree can be full without any leaves at level 2.

---

2. Divide-and-Conquer

(a) Recurrence: $T(n) = 5T(n/2) + \Theta(n^2)$. *4pts*
Solution: From the Master Theorem with $a = 5$, $b = 2$, $d = 2$, we get that $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 5})$.

(b) Recurrence: $T(n) = 3T(n/2) + \Theta(n^2)$. *4pts*
Solution: From the Master Theorem with $a = 3$, $b = 2$, $d = 2$, we get that $T(n) = \Theta(n^d) = \Theta(n^2)$.

(c) Recurrence: $T(n) = \sqrt{n}T(\sqrt{n}) + \Theta(n)$. *4pts*

Solution: Unwinding the recurrence, and replacing the $\Theta(n)$ term by Cn for a constant C,[1] we get

$$\begin{aligned}
T(n) &= n^{1/2}T(n^{1/2}) + Cn \\
&= n^{1/2}(n^{1/4}T(n^{1/4}) + Cn^{1/2}) + Cn \\
&= n^{3/4}T(n^{1/4}) + Cn + Cn \\
&= n^{3/4}(n^{1/8}T(n^{1/8}) + Cn^{1/4}) + Cn + Cn \\
&= n^{7/8}T(n^{1/8}) + Cn + Cn + Cn \\
&\vdots \\
&= n^{1-2^{-k}}T(n^{2^{-k}}) + kCn. \qquad (1)
\end{aligned}$$

To get the base case we need to take $k = \log\log n$, for then $n^{2^{-k}}$ becomes $n^{1/\log n}$, which is a constant. With this value of k, the first term in (1) becomes a constant times $n^{1-2^{-k}}$, which is less than n, and this is dominated by the second term which is $Cn\log\log n$. Thus the solution is $T(n) = \Theta(n\log\log n)$.

Most people got into a mess with this one. Some tried to apply the Master Theorem, even though the recurrence is not of the correct form for that theorem. Others started to unwind the recurrence but failed to notice the correct form of the series as a function of k, the number of levels of unwinding (as in (1) above). Many of these same people also didn't correctly compute the number of levels, which is $\log\log n$. Note that in fact this recurrence does work $\Theta(n)$ at every level; since there are $\log\log n$ levels the solution is $\Theta(n\log\log n)$.

---

3. Dijkstra's algorithm

(a) Here is the table produced by Dijkstra's algorithm: *6pts*

	Iteration					
Node	0	1	2	3	4	5
A	0	0	0	0	0	0
B	∞	1	1	1	1	1
C	∞	∞	3	3	3	3
D	∞	4	4	4	4	4
E	∞	8	7	7	7	6
F	∞	∞	7	5	5	5

Most people got this right. The most common mistake was to set $\text{dist}(E) = 6$ *in the fourth interation.*

6pts

(b) Our goal is to figure out a path starting from vertex s at time t_0, to a destination z, in the shortest possible time. The problem can be regarded as a shortest path problem if the travel time for each edge is treated as the "length" of that edge. Now the major difference here with the standard setting of the shortest path problem is that the "length" of each edge (u, v) is *dynamic* rather than a constant, depending on the time at which we leave vertex u. However, because of the physicality constraint, we may always assume that in any optimal path we always leave a vertex as soon as we arrive there (since waiting at a vertex cannot decrease the total travel time). This means that we can compute an effective length for each edge: namely, the length of edge (u, v) should be taken as $\ell_{(u,v)}(\text{time}(u))$, where $\text{time}(u)$ is the earliest arrival time at u. This earliest arrival time will be updated as the algorithm proceeds, just as with the dist labels in the usual shortest paths problem. Hence only the update portion of Dikjkstra's algorithm needs to modified: when we examine the edge (u, v), we update the label $\text{time}(v)$ based on $\text{time}(u) + \ell_{(u,v)}(\text{time}(u))$. Here is the algorithm:

[1] Of course, $\Theta(n)$ is not quite the same as a constant times n; what we really mean here is that we could carry out the following calculations with $\Theta(n)$ replaced by the upper bound $C_1 n$ and the lower bound $C_2 n$ and get the same form of answer (with only the constants changed). Hence our calculations will find the correct Θ-expression for $T(n)$.

```
procedure ModifiedDijkstra($G, l, s, t_0$):
  for all $u \in V$:
    $\mathrm{time}(u) = \infty$
    $\mathrm{prev}(u) = nil$
  $\mathrm{time}(s) = t_0$

  $H = makeheap(V)$ (using time values as keys)
  while $H$ is not empty:
    $u = deletemin(H)$
    for all edges $(u, v) \in E$:
      if $\mathrm{time}(v) > \mathrm{time}(u) + \ell_{(u,v)}(\mathrm{time}(u))$ :
        $\mathrm{time}(v) = \mathrm{time}(u) + \ell_{(u,v)}(\mathrm{time}(u))$
        $\mathrm{prev}(v) = u$
          $decreasekey(H, v)$
```

After running this algorithm, the fastest route from s to v can be obtained easily using by backtracking using the prev labels.

The most common mistake here was just to say "use the current time stamp to query the black box to get the travel time of the edge." But many people did not indicate how to keep track of that time stamp, and simply used t in the query, rather than $\mathrm{time}(u)$.

2pts

(c) As argued in part (b), the physicality constraint allows us to assume that we leave a vertex as soon as we arrive at it. This is the reason we can use the arrival time $\mathrm{time}(u)$ to query the travel time for edges in the update operation. But if the physicality constraint is removed, we might be able to wait at some vertex u until some time $t' > \mathrm{time}(u)$ such that $t' + \ell_e(t') < \mathrm{time}(u) + \ell_e(\mathrm{time}(u))$. If we could figure out the optimal waiting time t' at u, then we could still use the above algorithm with $\mathrm{time}(u) + \ell_{(u,v)}(\mathrm{time}(u))$ replaced by $t' + \ell_e(t')$. However, in the absence of other information we don't have any way to find the optimal t'.

Many people just said that removing the physicality constraint is the same as introducing negative edges into the graph (without explaining why). Actually this answer is inadequate for at least two reasons: (1) the connection with negative edges doesn't really make sense; (2) even if this were the same as introducing negative edges, it wouldn't show that the algorithm fails because Dijkstra's algorithm does not necessarily always fail with negative edges.

---

4. Yet another MST algorithm

(a) First, since the graph is connected, clearly there is an MST of G. Now there are two reasonable approaches for this problem: *4pts*

Using the Cut Property: Let $X = \emptyset$, $S = \{v\}$, and consider the cut between S and $V - S$. Clearly e_v is the minimum edge crossing this cut, so by the Cut Property e_v is in some MST of G.

Proof by contradiction: Suppose there exists an MST T of G that *doesn't* contain e_v. Add e_v to T. Obviously this creates a cycle. Consider the other edge incident on v, e'_v. If we remove this edge, since $w(e_v) < w(e'_v)$, then we destroy the cycle and create a new tree T' with cost$(T') <$ cost (T). This contradicts the assumption that T is an MST. Thus we can conclude that *all* MSTs of G contain e_v.

Almost everyone got this question right. However, we need to apologize. We should have asked you to prove that e_v is in **all** *MSTs of G. Notice that this follows directly from the proof by contradiction, but not from the Cut Property proof. The reason we should have done this is because this fact greatly simplifies the proof of part (b).*

(b) There are basically three parts to this proof: (1) showing that adding all the e_v to X in one iteration of the while loop is correct; (2) showing that the contraction operation does the right thing; (3) showing that all the iterations taken together lead to the MST of G. *5pts*

(1) By the proof by contradiction in part (a), we know that **all** the e_v selected in the first loop of the algorithm belong to an MST of G, and thus it is OK to add them to X. (This can also be deduced from the weaker version of part (a), with a little bit of extra work.)

(2) First, note that the set of edges e_v that are contracted form a forest (no cycles) in G; this follows from the fact that all edge weights are unique. Contracting the edges e_v causes each connected component (tree) of this forest to collapse into a single vertex, which means that we will never consider any of the other edges internal to the component. But this is correct because we already have a path between all vertices in the component, and any further edges would create a cycle. Furthermore, consider two edges coming out of such a component, both of which point to a node w. Choosing either of these edges will connect the component to w, and hence make every vertex in the component reachable from w. Since all edge weights are unique, we know that one edge is cheaper. Thus, we can simply throw away the more expensive edge. This is exactly what contraction does.

(3) After performing contraction, we now have a new set of vertices (essentially "super-vertices") that themselves need to be connected. We can again appeal to the proof by contradiction in part (a) to choose edges for the MST of this reduced graph G'. This is exactly what happens in the second iteration. We continue in this fashion until the vertex set V consists of a single super-vertex. At this point we have created a single connected component whose edge set X forms a spanning tree of G.

This was a hard question, especially because we didn't give you the best tool to use from part (a). Nonetheless, it was doable, and we were surprised that nobody quite managed full points. The most common mistake was to miss parts of the analysis. When you are analyzing an algorithm, you must consider all the key operations. Many people either discussed the choice of minimum edges, or contraction, but not both. Both are needed in any completely correct answer!

(c) Let V_k be the set of vertices considered in the kth iteration. So $V_1 = V$. In the kth iteration, we choose $|V_k|$ edges to contract, but these edges are not necessarily all distinct. However, any edge can be selected at most twice (e.g., edge (u, v) could be chosen both as e_u and as e_v). Hence we contract at least $|V_k|/2$ edges in this iteration, so we reduce the number of vertices by at least a factor of 2. But clearly we only need to do this at most $\log_2 |V|$ times before $|V|$ reaches 1. *4pts*

[Note that the algorithm might perform much better than this; in fact, it is even possible that we could form the entire MST in one iteration!]

Most students got this. One common mistake was to jump from noticing that there are $|V_k|$ contractions at iteration k, and then say that thus there are $|V_k|/2$ vertices left over after the contractions. This misses the core part of the argument. Another common mistake was to say that in the best case there are $|V_k|/2$ contractions, rather than in the worst case.

(d) Here is a concrete implementation of this algorithm that uses disjoint sets with path compression and union-by-rank to handle contracting nodes. You did not need to provide this algorithm, but it will help anchor the explanation below. *5pts*

```
function MST($G = (V, E)$)
  $X = \emptyset$
  for each $v \in V$:
    MAKESET($v$)
  while $|V| > 1$:
    for each $e = (v_i, v_j) \in E$:
      $p_i$ := FIND($v_i$)
      $p_j$ := FIND($v_j$)
      if $p_i \neq p_j$:
        $e^*_{p_i} :=$ edge that minimizes $\min\{l(e^*_{p_i}), l(e)\}$
        $e^*_{p_j} :=$ edge that minimizes $\min\{l(e^*_{p_i}), l(e)\}$
    for each edge $e^*_v = (u, v)$:
      add $e_v$ to $X$
      UNION$(u, v)$
  return $X$
```

The first for-loop inside the while finds the minimum for each vertex (including super-vertices). The FIND calls avoid considering edges within the same connected component. We handle the $(u, w), (v, w)$ case cited in the problem description because we're taking the min, which always chooses the cheaper edge. Finally, UNION performs the actual contraction.

From part (c), the number of iterations is $O(\log |V|)$. To find the minimum for each vertex, we examine each edge and perform a FIND on the vertices, so this takes $O(|E| \log^* |V|)$ time. Finally, we do $O(|V| \log^* |V|)$ work for contractions. The total time is $O(|E| \log |V| \log^* |V|)$.

No one got full points on this question. Note that you didn't have to go into the detail that we have here. The problem was that most students did not say how they handled multiple edges like (u, w) and (v, w) and did not talk about how to find the minimum after one iteration [Notice that it's enough to say that in the first iteration finding the minimum takes $O(|E|)$ time; however later on you need to use Union/Find to implicitly merge adjacency lists].

(e) The key observation in this part is to notice that we need a way of doing UNIONs and FINDs in $O(1)$ time. *3pts*

First, suppose MAKESET(v_i) actually simply marks in a size-$|V|$ array named parent that parent$(i) = i$. When contracting two vertices v_i, $v_j, i < j$, we let parent(j) = parent(i). This type of UNION only takes $O(1)$ time. Furthermore, suppose we perform contractions in order of vertex labels: namely, we contract the min edge for v_i before v_j if $i < j$. By performing contractions in this order, we are guaranteed that parent(parent(j)) = parent(j).

Finally, we add an edge relabeling phase after all the contractions are done. If an edge is (v_i, v_j) and i is j's parent, then this edge becomes (v_i, v_i). This relabeling ensures that we remove all references to contracted vertices. This takes $O(|E|)$ time.

This new set of edges E' ensures that if $(v_i, v_j) \in E'$, parent$(i) = i$ and parent$(j) = j$ and so FIND calls on the next iteration will take $O(1)$ time.

This algorithm takes $O(\log |V|)$ iterations, but this time computing the minimum takes $O(|E|)$ time and relabeling takes $O(|E|)$ time, and contraction takes $O(|V|)$ time. So the final running time is $O(|E| \log |V|)$.

This was a pretty difficult problem, and no one got it completely right. A small number of people correctly observed that we needed to make Union-Find take $O(1)$ time for each operation. But no one really said how we might do that. Quite a few people said it wasn't possible to do better with Union-Find: this is not true. We have a very structured problem where UNIONs and FINDs are performed in a certain order, and this is what our solution exploits. A number of students said that $\log^* |V|$ is practically a constant, so we can ignore it in the run-time analysis. This is not correct! Big-O notation is about function growth, and $\log^* |V|$ does go to infinity in the limit, even though it does so very slowly.
