Final Exam Solutions

Problem 1. What is Your Name? [2 points] (2 parts)

(a) [1 point] Flip back to the cover page. Write your name there.

(b) [1 point] Flip back to the cover page. Circle your recitation section.

Problem 2. Storing Partial Maxima [30 points] (1 part)

6.006 student, Mike Velli, wants to build a website where the user can input a time interval in history, and the website will return the most exciting sports event that occurred during this interval. Formally, suppose that Mike has a chronologically sorted list of n sports events with associated integer "excitement factors" $e_1, \ldots, e_n$. You can assume for simplicity that n is a power of 2. A user's query will consist of a pair (i, j) with $1 \leq i < j \leq n$, and the site is supposed to return $\max(e_i, e_{i+1}, \ldots, e_j)$.

Mike wishes to minimize the amount of computation per query, since there will be a lot of traffic to the website. If he precomputes and stores $\max(e_i, \ldots, e_j)$ for every possible input (i, j), he can respond to user queries quickly, but he needs storage $\Omega(n^2)$ which is too much.

In order to reduce storage requirements, Mike is willing to allow a small amount of computation per query. He wants to store a cleverer selection of precomputed values than just $\max(e_i, \ldots, e_j)$ for every (i, j), so that for any user query, the server can retrieve two precomputed values and take the maximum of the two to return the final answer. Show that now only $O(n \log n)$ values need to be precomputed.

Solution: We are given the list $e_1, \ldots, e_n$. For each $1 \leq i \leq n/2$, store $\max(e_i, e_{i+1}, \ldots, e_{n/2})$, and for each $n/2 < j \leq n$, store $\max(e_{n/2+1}, \ldots, e_j)$. Recurse separately on the two lists $e_1, \ldots, e_{n/2}$ and $e_{n/2+1}, \ldots, e_n$. Stop the recursion when the list size becomes 1.

If the user's query is (i, j) with $i \leq n/2$ and $j > n/2$, then we can return $\max(\max(e_i, e_{i+1}, \ldots, e_{n/2}), \max(e_{n/2+1}, \ldots, e_j))$. If both $i, j \leq n/2$ or $i, j > n/2$, then the answer is found recursively.

Let $S(n)$ be the number of values stored for a list of length n. By construction, $S(n) = O(n) + 2S(n/2)$, and therefore, $S(n) = O(n \log n)$.

Problem 3. Longest Simple Cycle [30 points] (2 parts)

Given an unweighted, directed graph $G = (V, E)$, a path $\langle v_1, v_2, ..., v_n \rangle$ is a set of vertices such that for all $0 < i < n$, there is an edge from v_i to v_{i+1}. A cycle is a path such that there is also an edge from v_n to v_1. A *simple path* is a path with no repeated vertices and, similarly, a *simple cycle* is a cycle with no repeated vertices. In this question we consider two problems:

- LONGESTSIMPLEPATH: Given a graph $G = (V, E)$ and two vertices $u, v \in V$, find a simple path of maximum length from u to v or output NONE if no path exists.
- LONGESTSIMPLECYCLE: Given a graph $G = (V, E)$, find a simple cycle of maximum length in G.

(a) [20 points] Reduce the problem of finding the longest simple path to the problem of finding the longest simple cycle. Prove the correctness of your reduction and show that it runs in polynomial time in $|V|$ and $|E|$.

Solution: Create a new graph G'. Copy all vertices and edges from G to G'. Then add $|V|$ more vertices to G', $w_1, ..., w_{|V|}$. For $0 < i < |V|$, create a directed edge from w_i to w_{i+1}. Also create a directed edge from v to w_1 and a directed edge from $w_{|V|}$ to u. Run LONGESTSIMPLECYCLE on G'. If the longest simple cycle involves vertices u and v, return the path from u to v in the cycle. Otherwise, output NONE.

Running Time: Creating the new graph takes $O(|V| + |E|)$. We run LONGESTSIMPLECYCLE once. Therefore, the running time of the algorithm is $O(|V| + |E| + L)$ where L is the running time of LONGESTSIMPLECYCLE.

Correctness: If a path from u to v exists then $c = \langle u, ..., v, w_1, ..., w_{|V|} \rangle$ is a cycle in G'. The cycle c contains at least $|V|+2$ vertices and, by construction, is the only cycle containing the w vertices. Therefore, all other cycles must have length at most $|V|$. Hence LONGESTSIMPLECYCLE will return this cycle if it exists and our algorithm will return $p = \langle u, ..., v \rangle$. Since the cycle is simple, p must contain only vertices in V and is thus the longest path from u to v in G. If the cycle does not exist, there is no path from u to v and the algorithm correctly outputs NONE.

Note: This is only one possible reduction. People came up with others and that's fine so long as they are correct and polynomial.

Grading:

- 20/20 For any polynomial time reduction. The reduction did not have to be linear to receive full credit.
- 19.5/20 if you added a weighted edge from v to u rather than a number of vertices (the graph was unweighted)
- 18/20 If you gave the reduction of adding an edge from v to u, finding the connected cluster containing v and u and running LSC on that. The problem is that

the connected cluster containing v and u might have a simple cycle that does not involve v and u longer than the one that does involve v and u. The whole connected cluster is a cycle, but not a simple one.

- 17/20 For removing back edges
- 17/20 For removing all nodes u can't reach
- 14/20 If you tried to remove edges one at a time, which does not work, but the algorithm was still polynomial.
- 10/20 If you returned the longest path in the graph, but not necessarily from u to v.
- 10/20 If you did the reduction correctly in the wrong direction
- From the above grades points were also subtracted for:
  - -1 for no runtime analysis
  - -2 for no correctness proof
  - -1 if you forgot to add an edge from v to u and your reduction required one.

(b) [10 points] As we discussed in class, finding a longest simple path is NP-Hard. Therefore, there is no known algorithm that, on input u, v, and G returns a longest simple path from u to v in polynomial time. Using this fact (which you do not need to prove) and Part (a), show that there is no known polynomial time algorithm that can find a longest simple cycle in a graph.

Note: If you were unable to solve Part (a), you may assume an algorithm SIMPLEPATHFROMCYCLE for finding a longest simple path from u to v that runs in time polynomial in L, $|V|$, and $|E|$ where L is the running time of a black-box algorithm for solving LONGESTSIMPLECYCLE.

Solution: If L is polynomial than the algorithm outlined in Part (a) gives a polynomial time algorithm for finding the longest simple path in a graph. Contradiction.

Grading:

- 10/10 For anything like above proof. It was also fine to say that the reduction means LSC must be at least as hard as LSP since we stressed that formulation in class.
- 9/10 If you said you can use LSC to solve LSP and therefore by definition of NP-Hard, LSC is also NP-Hard. The definition of NP-Hard is that all NP optimization problems can be reduced to some problem in NP-Hard. So you need an extra step just tying all the ends together, but it's very close.

Problem 4. Closest pair [28 points] (2 parts)

We are interested in finding the closest pair of points in the plane, closest in the sense of the rectilinear distance (also called the Manhattan or L_1 distance). The rectilinear distance between two points p_1 and p_2 on the plane is defined as $d(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$, where the x's and y's are the first and second coordinates, respectively. In the first part we consider a one-dimensional version of the problem as a warm-up. In both cases, coordinates of the points are real numbers with k significant digits beyond the decimal point, k constant.

(a) [8 points] Warm up - Provide an efficient way to find a closest pair among n points in the interval $[0, 1]$ on the line. Full credit will be given to the most efficient algorithm with a correct analysis.

Solution: Use radix sort on the n numbers (k digits representation). Then go through the sorted list and calculate the distance between each point and the next one, keeping a running minimum along the way. Radix sort will take $O(kn)$, and distance calculations $n - 1$ time $O(k)$, so an overall linear time $O(n)$, since k is fixed. (Half points only given to an $O(n \log n)$ answer with otherwise correct analysis.)

(b) [20 points] Case of the plane - A divide and conquer approach for n points in the square $[0, 1] \times [0, 1]$. Here is a possible strategy. Divide the set of points into two sets of about half size: those on the right of the x-coordinate median, and those on the left. Recursively, find the closest pair of points on the right, and the closest pair of points on the left and let δ_r and δ_l be the corresponding distances. The overall closest pair is either the minimum of these two options, or corresponds to a pair where one point is on the right of the median and the other is on the left. Given δ_r and δ_l, there should be an efficient way to find the latter. Explain how; then write the full recurrence for the running time of this approach; and conclude with its overall running time.

Solution: Searching for closest pair of points that cross the median x-coordinate can be limited to the vertical strip around the median line of width +/- δ where $\delta = \min\{\delta_r, \delta_l\}$. By having the points belonging to this strip sorted according to their y-coordinates, one can go through of all them bottom up, and for each check distances to points above them within a vertical bound of δ (one can show that only the 7 points above need to be checked for each point). In the worst case, all n points are in this strip, so we end up with a recurrence for the running time of $T(n) = 2T(n/2) + O(n)$ and so $T(n) = O(n \log n)$.

Problem 5. APSP Algorithm for Sparse Graphs [30 points] (3 parts)

Let $G = (V, E)$ be a weighted, directed graph that can have some of the weights negative. Let $n = |V|$ and $m = |E|$, and assume that G is strongly connected, i.e., for any vertex u and v there is a path from u to v in G.

We want to solve all-pairs-shortest-path problem (APSP) in G, i.e., we want to either find all the vertex-to-vertex distances $\{\delta(v, u)\}_{v,u \in V}$, or report existence of a negative-length cycle. We will design an algorithm for this task that runs in $O(mn + n^2 \log n)$ time. (Note that when G is not dense, i.e., when m is $o(n^2)$, the running time of this algorithm is asymptotically better than the one of the Floyd-Warshall algorithm.)

(a) [5 points] Fix some vertex $t \in V$ and consider the vertex potential $\lambda_t(u) = \delta(u, t)$ where $\delta(u, t)$ is the shortest path from $u \in V$ to t. Give an algorithm for calculating $\lambda_t(u)$ for all $u \in V$ and analyze the running time.

Solution: Bellman-Ford in $O(|V||E|)$ time.

(b) [5 points] Show $\lambda_t(u)$, the potential from Part (a), is a feasible potential even if some of the original weights are negative.

Proof of the feasibility of λ:

Consider $w^*(u, v)$ for any $u, v \in V$. By definition

$$w^*(u, v) := w(u, v) - \lambda(u) + \lambda(v) = w(u, v) - \delta(u, t) + \delta(v, t).$$

Since $\delta(u, v) \leq w(u, v)$, we have that

$$w^*(u, v) = w(u, v) - \delta(u, t) + \delta(v, t) \geq \delta(u, v) - \delta(u, t) + \delta(v, t).$$

But, by triangle inequality we have that $\delta(u, t) \leq \delta(u, v) + \delta(v, t)$, thus

$$w^*(u, v) \geq \delta(u, v) - \delta(u, t) + \delta(v, t) \geq 0,$$

as desired.

(c) [20 points] Show how you use the vertex potential from Parts (a) and (b) to solve APSP in G in $O(mn + n^2 \log n)$ time, including the time it takes to calculate the vertex potential.

Solution: Choose any vertex t in G and run Bellman-Ford algorithm from it on the transposed graph G to either compute all the distances $\delta(u, t)$ for every $u \in V$, or detect a negative-length cycle. If such a cycle is detected in the transposed graph then it also exists in the original graph, so just report its existence and terminate.

Otherwise, define the vertex potential $\lambda(u) := \delta(u, t)$ and look at the resulting reduced weight $w^*(u, v) := w(u, v) - \lambda(u) + \lambda(v)$ in G. Since G is strongly connected (and we now know it does not have negative-length cycles), all distances $\delta(u, t)$ are finite and thus all $\lambda(u)$'s are finite as well. From the recitations (and pset 5B) we know that this implies that λ is a feasible potential, i.e., for every $u, v \in V$ $w^*(u, v)$ is non-negative (for the sake of completeness we include the proof of feasibility below).

Since the reduced weight w^* is non-negative, one can run Dijkstra's algorithm in G from every node $u \in V$ to compute all the vertex-to-vertex distances $\{\delta^*(u, v)\}_{u,v \in V}$ with respect to w^*. Next, extract the true vertex-to-vertex distances in G by computing $\delta(u, v) = \delta^*(u, v) - \lambda(v) + \lambda(u)$ for all $u, v \in V$, and output them.

The running time of the above algorithm is dominated by one execution of Bellman-Ford algorithm – which is $O(mn)$ time – and n executions of Dijkstra's algorithm – which is $O(n(m + n \log n)) = O(mn + n^2 \log n)$ time using the implementation of Dijkstra with Fibonacci heaps. Thus the total running time is $O(mn + n^2 \log n)$, as desired.

Problem 6. Airplane Scheduling [30 points] (3 parts)

Consider the runway reservation system from PS2. Recall that we kept track of requested landing times from airplanes by storing them in a balanced binary search tree. In that problem set, we required, for safety, that no landing time be within three minutes of any other landing time in the tree. We showed we could insert into the tree, delete from the tree, and check the validity of a landing time in $O(\log n)$ time.

Sometimes severe weather hits, and the 3-minute window between flights just isn't safe, so a new window size is determined. In these cases, an extra runway might be opened up at a nearby airport to take flights which don't fit within the new window. For all parts, you may use data structure augmentations provided that you explain the augmentation. Its maintenance may not increase the asymptotic running time of other operations, but you are not required to prove this.

(a) [5 points] Provide a very fast (constant-time) algorithm to determine if there are any flights which are not valid with the new window so that the extra runway can start opening immediately.

(b) [15 points] Also provide a slower algorithm which locates which specific flights are not valid within the new window so they can be rescheduled. For example, assuming a window of 3 minutes with flights at times 28, 31, 35, 40, 43, 48, 53, 57, 60, if the window size were expanded to 4 minutes, the algorithm should return that 28, 31, 40, 43, 57, 60 are invalid.

Solution: Like question 2d on the problem set, but instead of tracking for the presence of a particular window size, track the minimum window of the subtree. Constant-time lookup at the root to see if any window is smaller than the new window value. Traversing the tree is both $O(n)$ and $O(k \log n)$ to find all newly invalid times. With bounds on the window size, we could keep a side-structure with all flights belonging to a given window.

(c) [10 points] Can the running time of this slower algorithm be improved if we assume bounds on the maximum size the window could become?

Problem 7. Traveling on a Budget [30 points] (2 parts)

Arthur Dent has \$500 and 1000 hours to go from Cambridge, MA, to Berkeley, CA. He has a map of the States represented as a directed graph $G = (V, E)$. The vertices of the graph represent towns, and there is a directed edge $e = (A, B)$ from town A to town B if there is some means of public transportation connecting the two towns. Moreover, the edge is labeled with a pair (m_e, t_e), representing the cost $m_e \in \{0, 1, \ldots\}$ in dollars of transportation from A to B and the time $t_e \in \{0, 1, \ldots\}$ in hours that it takes to go from A to B.

Arthur is interested in finding a path from Cambridge to Berkeley that does not cost more than \$500 and does not take more than 1000 hours, while also minimizing the objective $5M^2 + 2T^2$, where M is the cost of the trip in dollars and T is the duration of the trip in hours. He was looking for an algorithm that runs in time polynomial in $|V|$ and $|E|$ ...

(a) [10 points] Due to his lack of knowledge in algorithms, he gave up on the idea of respecting the budget and time constraints. At least he thought he could efficiently find the path minimizing the objective $5M^2 + 2T^2$. He tried modifying Dijkstra's algorithm as follows: If an edge e was labeled (m_e, t_e), he assigned it a weight $w_e = 5m_e^2 + 2t_e^2$, and ran Dijkstra's algorithm on the resulting weighted directed graph. Show that Arthur's algorithm may return incorrect results, i.e. return a path that does not minimize the objective $5M^2 + 2T^2$.

(b) [20 points] Now provide an algorithm that solves Arthur's original problem in time polynomial in $|E|$ and $|V|$. Your algorithm should find the path that minimizes the objective $5M^2 + 2T^2$, while at the same time respecting the constraints $M \leq 500$ and $T \leq 1000$. Please describe your algorithm precisely, and justify its correctness and running time. More credit will be given to faster algorithms, provided that the analysis of the algorithm is correct.

[*Hint 1: Use dynamic programming.*]

[*Hint 2: For each town A, integer values $m \leq 500$ and $t \leq 1000$, either there is a path from Cambridge to A that requires cost m and time t, or there is not.*]

Solution sketch: We construct a 501-by-15 sized matrix M_v at each vertex v so that the (i, j)'th entry of M_v is 1 if there's a path from the node for Cambridge to v of total cost i and total duration j and is 0 otherwise. $M_{\text{Cambridge}}$ is set to be 1 at entry $(0, 0)$ and 0 everywhere else.