CS180 Final Exam Solutions

For all the algorithms you design, in addition to describe your algorithm clearly, please also (a) briefly justify the correctness of the algorithm; (b) present the time complexity of the algorithm and briefly justify the reason. Partial credits will be given if your algorithm has complexity slightly worse than the solution for all the problems.

1. (20 pt) (Yuanhao) Given an undirected connected graph where each edge is associated with a positive weight, we want to find a set of edges such that removing those edges will make the graph acyclic. Design an algorithm to find such edge set with the smallest total weight. The algorithm should run in $O((m+n)\log n)$ time.

   Solutions: This is equivalent to finding the "max" cost spanning tree, which can be done by negating all the edge weight and then find a minimum spanning tree. In detail, first negate all edge weights and the graph is denoted by G'. Then use Prim's or Kruskal's algorithm to find the minimum spanning tree T' for G'. Edges in T' are kept and the rest edges in the original graph are removed with the smallest total weight.

2. (25 pt) (Lucas) In this problem, our goal is to design sublinear time algorithms for finding a “hill” in a given 1D or 2D array. We say an element in a 1D or 2D array is a “hill” if and only if its value is larger than all its neighbors. In 1D array the neighbors for $A[i]$ are $A[i-1]$ and $A[i+1]$ and in 2D the neighbors for $A[i, j]$ are $A[i-1, j]$, $A[i+1, j]$, $A[i, j-1]$, $A[i, j+1]$. Elements on the boundary of arrays will have less neighbors, for instance $A[0]$ only has one neighbor $A[1]$; $A[0, 0]$ only has two neighbors $A[0, 1]$, $A[1, 0]$. An array could have multiple hills, and we only need to find one of them. Figure 1 illustrates two examples, one in 1D and another in 2D.

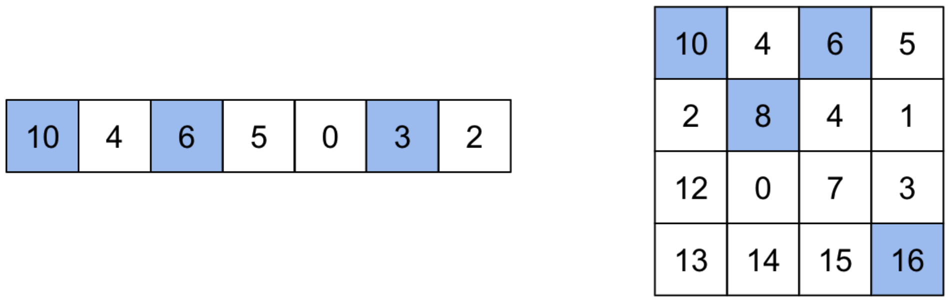

Figure 1: The right panel illustrates a 1D example and the left panel illustrates a 2D example, where the blue cells are hills. There could be multiple hills and our goal is to find one of them.

(a) (10 pt) Given a 1D integer array of size n and assume the values are distinct. Design an algorithm to find a hill in $O(\log n)$ time.

(b) (15 pt) Now we extend the algorithm to find a hill in a 2D array of size $n \times n$. Design an algorithm to return the position of one of the hills in $O(n)$ time. Partial credits will be given to algorithms with slightly higher complexity, for instance, a solution with time complexity $O(n \log n)$ will get 10 points.

Solution: Divide and conquer:

(a) Check the middle element and its neighbors to decide which half we want to go. If $A[mid] < A[mid+1]$, then discard the left half; otherwise discard the right half. This is like binary search so $O(\log n)$ time.

(b) Check all the elements of the middle row and column. If the intersection of middle row and column is the max element, then it’s a hill. Otherwise WLOG assume the max is on the middle row, call this element x. (You can just treat the row coordinates as columns and columns to be rows if the max happens to be on a column.) Check the elements above and below it. If they are both smaller than x then x is the hill. Otherwise one of them is larger than x, we can then know there’s a hill in the corresponding quadrant, so the problem is reduced to a subproblem with size $n/2 \times n/2$. Time complexity is $O(n) + O(n/2) + O(n/4) + \cdots = O(n)$.

This problem has been discussed online (e.g., finding local minimum in a 2D array) but many solutions are wrong. In particular, here are several wrong or suboptimal answers:

- Each time only check a row, finding the max value x, check the elements above and below x to determine whether discarding the top half or bottom half. This is correct, but if you only do row-split, the complexity will be $O(n \log n)$ since each iteration always requires time n and we have $\log n$ iterations. (so this is a solution with suboptimal time complexity; some online posts said this is $O(n)$ which is wrong, we will also deduce some more points if they give this solution but think this has $O(n)$ time complexity).

- The second problematic solution is to argue you can extend the previous suboptimal solution but split rows and columns alternatively. This is actually **wrong**. See the following example:

0	6	5	0	0
0	7	4	20	0
1	9	3	19	0
0	13	0	18	0
0	15	16	16	0

  First, choose middle row, the max is 19 and since $20 > 19$ you're going to focus on the top half. Then you choose the middle column you'll have either 5, 4, 3 or 5, 4. In both cases 5 is max and $6 > 5$ so you're going to go left. Unfortunately, there is no hill on the top-left quadrant.
- To make the previous problematic solution correct, you need to record the "max value" in the history and if the new max value in the branch is smaller than the historical max, you have to go to both branch instead of a single branch. This will then be correct but also has time complexity more than $O(n)$. (Or at least, I don't know a way to show this is $O(n)$)

3. (25 pt) (Noor) There are n cities on a highway with coordinates $x_1, \ldots, x_n$ and we aim to build $K < n$ fire stations to cover these cities. Each fire station has to be built in one of the cities, and we hope to minimize the average distance from each city to the closest fire station. Please give an algorithm to compute the optimal way to place these K fire stations. The algorithm should run in $O(n^2K)$ time. Partial credits will be given to algorithms with slightly higher complexity, for instance, a solution with time complexity $O(n^3K)$ will get 15 points.

Solution: Dynamic programming. Sort in O(n log n). Assume $x_1 < x_2, \cdots < x_n$. Let $L(i,j)$ be the cost for covering first i cities with j fire departments such that there is a fire department at x_i. . We have

$$L(i,j) = \min_{1 \le p \le i-1} \big(L(p, j-1) + \text{cost}(p+1, \ldots, i-1)\big),$$
$$L(i,1) = \sum_{j=1}^{i-1} |x_i - x_j|,$$

where cost$(p+1, \ldots, i-1)$ is the cost of cities $p+1, \ldots, i-1$ when there are fire departments located at p and i. If one uses a naive way to compute cost, the algorithm will require $O(n^3k)$, but if we compute cost progressively the DP will be $O(n^2k)$ time. Finally we need to consider all the cases when the last fire office is placed at $1, \ldots, n$, so the final solution is

$$\min_{i=1}^{n} \big(L(i,K) + \sum_{j=i+1}^{n} |x_i - x_j|\big).$$

Notes:

- **Cho: We can give full credit even if the students don't trace back to print out the optimal placement; as long as they can get the optimal value they'll get full credit.**

4. (30 pt) (Kevin) Decision tree is an important model for binary classification. Given an input binary string $\boldsymbol{x} = x_1x_2 \dots x_d$, each x_i denotes a binary attribute of an input instance (e.g., in practice an input instance could be a document, an image, or a job application). A decision tree tries to map this string to a prediction value based on a tree structure—starting from root node, at each node we decide going left or right by the value of an attribute x_i; and at each leaf node will assign either $+1$ or -1 to the input. A decision forest consists of multiple decision trees, and the final prediction value is the sum of all these predictions. If we use $f_t(\boldsymbol{x})$ to denote the prediction value of the t-th tree and assume there are in total T trees, the final prediction of the decision forest is

$$\begin{cases} True & \text{if } \sum_{t=1}^{T} f_t(\boldsymbol{x}) \geq 0 \\ False & \text{otherwise.} \end{cases}$$

For example, Figure 2 illustrates a decision forest and the prediction values for several input strings.

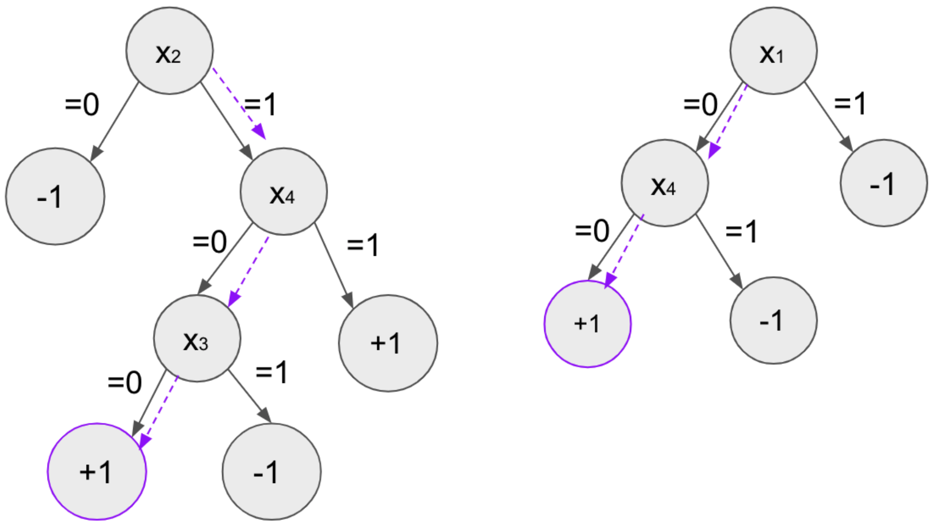

Figure 2: A decision forest. For input $x_1x_2x_3x_4 = 0100$ it will traverse the trees based on the dashed arrow, so the first tree outputs $+1$, the second tree output $+1$, and the final output is True. For the same decision forest, the input $x_1x_2x_3x_4 = 0011$ will produce -2, thus False.

An important property for a machine learning model is that the model can't always produce the same output. Therefore, we want to solve the **Forest-Verify** problem such that given a decision forest, determine whether there exists a d-dimensional input binary string $\boldsymbol{x}$ such that the prediction of this decision forest is $True$. (The same procedure can also detect whether there exists an input to produce $False$).

Show the **Forest-Verify** problem is NP-complete.

(a) (7 pt) Show the Forest-Verify problem belongs to NP.

(b) (7 pt) Let's first assume there's only one Clause in 3-SAT, can you turn this into a single decision tree such that the prediction of Decision tree corresponds to the value of this Clause?

(c) (16 pt) Derive a polynomial time reduction from 3-SAT to Forest-Verify.

Solution:

(a) Trivial. We make a verifier which takes evidence t that's the proposed binary string that makes it true. The verifier feeds the input string into all trees, and evaluates. To evaluate, we make at most 1 decision per node (some nodes we might not even visit). Input includes all nodes, so this work is polynomial in the number of nodes. So this is a polytime verifier, and so this is in NP.

(b) We can build a depth-3 tree according to the clause, such that it produces -1 only when the clause is False and $+1$ otherwise. Specifically, it has a path down to -1 for if all 3 literals are false, and if any one literal is true then it branches off to $+1$.

Literals may be x or !x, but tree nodes are only variables, not their negative. If a literal is x, then the corresponding node has x = true go to +1, and false go to the -1 path. And vice versa, if the literal is !x, then the corresponding decision node has x = false go to +1, and true go to the -1 path.

(c) We can turn each clause into a depth-3 tree as in (b). These trees are constant size, so this is polynomial work (one tree per clause). If there are K clauses, the forest will total to K if all the clauses are satisfied, and $\leq K - 2$ if any clause is violated. Therefore we can construct another $K - 1$ "dummy" trees that always output -1, to make the total 1 if all clauses are satisfied, and ≤ -1 if any clause is violated. These dummy trees can just be single node trees that are a "-1" leaf. This makes it so the decision forest will output True exactly when the 3SAT has a satisfiable assignment.

—

Doesn't work:
- Modifying leaves to other values besides +1 or -1
- Modifying the evaluation function
- Stacking all the trees into one giant tree (No longer polynomial work: 3 leaves are +1, and each of those needs a subtree, for 3^K total subtrees
- Calling ForestVerify on each individual tree. This doesn't guarentee that it's the same input string that makes them all work.

Maybe ok?
- If they have some other solution with multiple calls to ForestVerify. We allowed polynomial of calls.