21S-COMSCI180-1 Final

JOE PINTO, SR

TOTAL POINTS

99 / 100

QUESTION 1

1 Q1 20 / 20

✓ **- 0 pts Correct**

- **- 4 pts** No runtime analysis.
- **- 20 pts** Wrong Algo.
- **- 12 pts** Do not satisfy the runtime constraint.
- **- 3 pts** Tiny mistakes.
- **- 2 pts** Do not return the edges should be removed.

QUESTION 2

Q2 25 pts

2.1 (a) 10 / 10

✓ **- 0 pts Correct**

- **- 2 pts** Algorithm slightly off
- **- 2 pts** Missing complexity, proof of correctness
- **- 5 pts** Click here to replace this description.
- **- 10 pts** Missing tag to question, please check with TA

2.2 (b) 15 / 15

✓ **- 0 pts Correct**

- **- 5 pts** Suboptimal / incorrect algorithm
- **- 6 pts** Incorrect complexity with incorrect algorithm
- **- 7 pts** Click here to replace this description.
- **- 10 pts** Click here to replace this description.

QUESTION 3

3 Q3 25 / 25

✓ **- 0 pts Correct**

- **- 10 pts** Not using 2D dynamic programming.
- **- 25 pts** No answer
- **- 5 pts** Minor mistake with dynamic programming.

QUESTION 4

Q4 30 pts

4.1 (a) 7 / 7

✓ **- 0 pts Correct**

- **- 7 pts** Missing

4.2 (b) 6 / 7

- **- 0 pts** Correct
- **- 0.5 pts** very minor error (see comment)

✓ **- 1 pts special case missing: negation operator**

- **- 1 pts** Value assignment is wrong: +1 is true and -1 is false.
- **- 2 pts** Clauses won't contain AND operator
- **- 3 pts** Incomplete answer: after reading your answer, I still don't know how exactly to create such a tree.
- **- 3 pts** A clause is a three-tuple connected with *or* operators, not "and".
- **- 5 pts** Mostly wrong / only answered "Yes" with no correct explanation.
- **- 7 pts** Empty / completely wrong.
- **- 7 pts** Handwriting extremely hard to read (regrade request is welcomed).

4.3 (c) 16 / 16

✓ **- 0 pts Correct**

- **- 8 pts** Reduction is not polynomial. The ForestVerify problem must be polynomial size. A single tree needs 3x more subtrees per layer (one per +1), giving exponential size. Also not allowed to just do pointers, as the actual ForestVerify problem is still the full exponential tree. (Otherwise you can solve 3SAT in polytime by following a +1 node backwards)
- **- 12 pts** Calling ForestVerify on each tree (clause) only tells you that each tree has some assignment that makes that tree true. Trees representing a single

clause are always satisfiable trivially. This has no guarantee that the SAME assignment makes ALL trees true. "Fixing this" takes exponential time, as it would be a solver for 3SAT, since ForestVerify is trivial on clause trees.

- 12 pts Not allowed to modify the evaluation function or node values. Nodes must be +1/-1. Evaluation is always "True if sum >= 0". You can only check if it's in ForestVerify, aka if some assignment makes sum >= 0 or not. Changing any of this makes it not a valid ForestVerify problem. This was stated in the FAQ. (For comparison, you can't for example change what 3SAT being satisfied means, so you can't change what ForestVerify being True means)

- 12 pts Did not account for ForestVerify accepting if half of the trees can be satisfied, while 3SAT requires all to be satisfied. "sum of [+1 or -1] > 0" only needs half to be +1, the other half can be -1. E.g. just stated to use part (b) for every clause, and no other construction.

Close to correct (trying to add extra trees that are always false), but minor issue

- 4 pts Tried to add "always false" clauses then turn them into trees, which can't be done as seen in HW4 Approx. Close enough to the right answer, you can go straight to making "always -1" trees.

- 4 pts Can't have "always false" literals, since the decision problem considers all possible assignments (similar to HW4). Close enough to the right answer, as it's trying to make "always -1" trees.

- 4 pts Can't have trees that are true when all clauses true and false when any clause is false, as that just solves 3SAT.

- 12 pts Defined "half-3SAT", reduction from 3SAT to half-3SAT is missing/wrong. This is not trivial. You can't just repeat Approx from Hw as that adds 7 T and 1F and can't reach 1/2, and you can't add "always false" clauses as seen with the HW.

- 0 pts Other issue (see comment)

- 16 pts Reduction in the wrong direction: Showed Forest < 3SAT, problem asked for 3SAT < Forest

- 16 pts No answer, no reduction/construction provided from multi-clause 3SAT, or extremely incorrect

CS180 Final Exam

Due: 11:29 am PDT, June 9
Please submit on gradescope

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) 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.

This problem is equivilant to finding the "Maximum spanning tree" of the graph. This is defined as the spanning tree of a graph that has the maximum possible weight. We can adapt Prims Algorithm (which normally finds the minimum spanning tree) to find the maximum spanning tree as follows:

- negate all edge weights in G ($O(n)$)
- Run Prim's Algorithm on the modified G, $O((m+n)\log n)$
- Renegate the nodes in the Maximum spanning tree ($O(n)$)
- return the set V − S, where S is the set of nodes in the spanning tree.

This implementation essentially computes a maximum spanning tree using a modified prim's algorithm, thereby indirectly computing the minimum cost set of edges needed to make the graph acyclic, as required. The overall complexity is $O(n) + O(n) + O((m+n)\log n)$

$= O((m+n)\log n)$

1 Q1 **20 / 20**

✓ **- 0 pts** Correct

- 4 pts No runtime analysis.

- 20 pts Wrong Algo.

- 12 pts Do not satisfy the runtime constraint.

- 3 pts Tiny mistakes.

- 2 pts Do not return the edges should be removed.

2. (25 pt) 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.

10	4	6	5	0	3	2

10	4	6	5
2	8	4	1
12	0	7	3
13	14	15	16

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.

a) Algorithm: initialize start=0, end=n-1, array = a

```
FindHill(a, start, end)
if start == end:
    return a[start]
else:
    mid = ⌊(start+end)/2⌋
    if(a[mid] > a[mid+1])
        return findHill(a, start, mid)
    else
        return findHill(a, mid+1, end)
```

This algorithm is essentially a binary search that operates in $O(\log n)$, since we half the search size in each recursive call.

2.1 (a) **10 / 10**

- ✓ **- 0 pts** Correct
- **- 2 pts** Algorithm slightly off
- **- 2 pts** Missing complexity, proof of correctness
- **- 5 pts** Click here to replace this description.
- **- 10 pts** Missing tag to question, please check with TA

2 b) Algorithm:

compute mid $= \lfloor \frac{n}{2} \rfloor$
max $= 0$

for each element $e \in$ mid column: $O(n)$
  if $e >$ max:
    max $= e$
for each element $e \in$ mid row: $O(n)$
  if $e >$ max:
    max $= e$
if max is in the mid row and column: $O(1)$
  return max
else if max is from the mid column: $O(1)$
  if max bigger than the left cell and right cell (if they exist):
    return max
  else
    recurse on the subarray that has the bigger neighbour of max (left or right)
else if max is from the mid row:
  if max bigger than cell above and cell below (if they exist):
    return max
  else
    recurse on the subarray that has the bigger neighbour of max (left or right)

Time complexity:

each iteration has $2 \times O(n)$ loops and will also recursively call itself with size $\frac{n}{4} \times \frac{n}{4}$ arrays. This means the total time complexity is

$$T(n) = T\left(\frac{n}{4}\right) + 2n$$

By master theorem, since $\frac{a}{b^k} = \frac{1}{4} < 1$, this is a $O(n)$ algorithm.

2.2 (b) **15 / 15**

✓ **- 0 pts** Correct

- 5 pts Suboptimal / incorrect algorithm

- 6 pts Incorrect complexity with incorrect algorithm

- 7 pts Click here to replace this description.

- 10 pts Click here to replace this description.

3. (25 pt) 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.

Use Dynamic Programming.

Define a $(n+1) \times (k+1)$ sized matrix M where each entry $M[i,m]$ (for $1 \le i \le n$ and $1 \le m \le k$) corresponds to the optimal fire station arrangement for placing m fire stations among the first $x_1, \ldots, x_i$ cities, with the additional requirement that a fire station must be at x_i. This added constraint is to avoid the situation where adding a fire station at x_{i+1} alters the previously computed distances of the nearest post offices for cities $x_1, \ldots x_i$, which then allows us to break the problem into subproblems and apply dynamic programming. Note that $M[i,m]$ is then calculated as

$$M[i,m] = \min \left\{ M[i-1, m-1], M[i-2, m-1] + cost(i-1), M[i-3, m-1] + cost(i-2, i-1) \ldots \right\}$$

$$= \min_{1 \le z \le i-1} \left\{ M[z, m-1] + cost(z+1, \ldots i-1) \right\}$$

where $cost(z+1, \ldots i-1)$ denotes the total cost for each city $x_{z+1}, \ldots x_{i-1}$ given that the two closest fire stations are at x_z and x_i

We effectively are computing the best firestation location (x_z) such that the resulting sum of distances of cities $x_1, \dots x_i$ is minimized.

The final solution $M[n, K]$ is then

$$M[n, K] = \min_j \left\{ M[j, K] + \sum \text{Costs}(j+1, \dots N) \right\}$$

Given the costs are computed in $O(n)$ (by simple sums) each entry is computed in $O(n^2)$ time, giving an overall complexity for nk entries of $O(nkn^2)$ $= O(n^3k)$ time. This can be lowered by computing the costs progressively based on the previous entries costs, making the calculation $O(1)$, so overall we compute each iteration in $O(n)$ time for a total runtime of $O(n^2k)$.

3 Q3 **25 / 25**

- ✓ **- 0 pts** Correct
- **- 10 pts** Not using 2D dynamic programming.
- **- 25 pts** No answer
- **- 5 pts** Minor mistake with dynamic programming.

4. (30 pt) Decision tree is an important model for binary classification. Given an input binary string $\boldsymbol{x} = x_1 x_2 \ldots x_d$, 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.

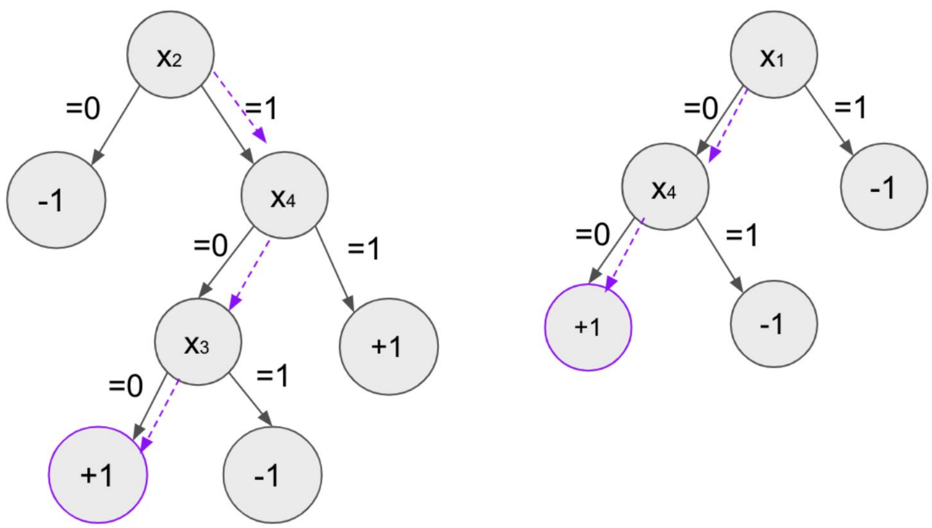


Figure 2: A decision forest. For input $x_1 x_2 x_3 x_4 = 0100$ it wil traverse the trees based on the dashed arrow, so the first tree outputs $+1$, the second tree output $+1$, and the final output is True. For the same decision forest, the input $x_1 x_2 x_3 x_4 = 0011$ will produce -2, thus False.

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?

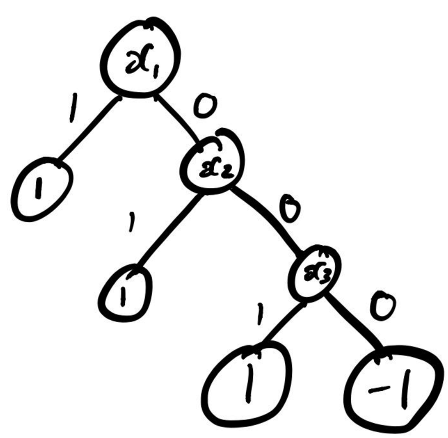
(c) (16 pt) Derive a polynomial time reduction from 3-SAT to Forest-Verify.

a) In order to show Forest-Verify is in the NP class, we can construct the following certifier:

In this case our input string, s would be the decision forest itself. Our "evidence" t would be a binary input string $x = x_1 x_2 \ldots x_n$. Our certifier algorithm $B(s,t)$ would then simply be running t through the decision forest s to verify whether t outputs true. This trivial substitution can be done in polynomial time, so this means that the forest verification problem is in the NP class, since it has an efficient certifier.
Note that this can be done in polynomial time since at most each tree can go through d decisions to evaluate, where d is the depth of the tree, so for some number of trees, this will be some polynomial order of d.

b) You could always construct a decision tree in the following way for a single 3-SAT clause:

consider clause $\{x_1 \vee x_2 \vee x_3\}$. This clause will only evaluate to true if at least 1 of x_1, x_2 or x_3 is of value 1. We can mimic this behavior by constructing the decision tree as follows:



4.1 (a) **7 / 7**

✓ **- 0 pts** Correct

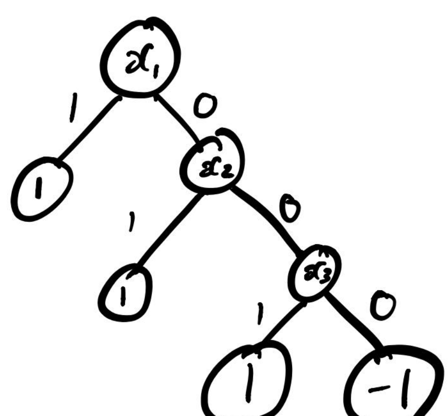
- 7 pts Missing

a) In order to show Forest-Verify is in the NP class, we can construct the following certifier:

In this case our input string, s would be the decision forest itself. Our "evidence" t would be a binary input string $x = x_1 x_2 \dots x_n$. Our certifier algorithm $B(s,t)$ would then simply be running t through the decision forest s to verify whether t outputs true. This trivial substitution can be done in polynomial time, so this means that the forest verification problem is in the NP class, since it has an efficient certifier. Note that this can be done in polynomial time since at most each tree can go through d decisions to evaluate, where d is the depth of the tree, so for some number of trees, this will be some polynomial order of d.

b) You could always construct a decision tree in the following way for a single 3-SAT clause:

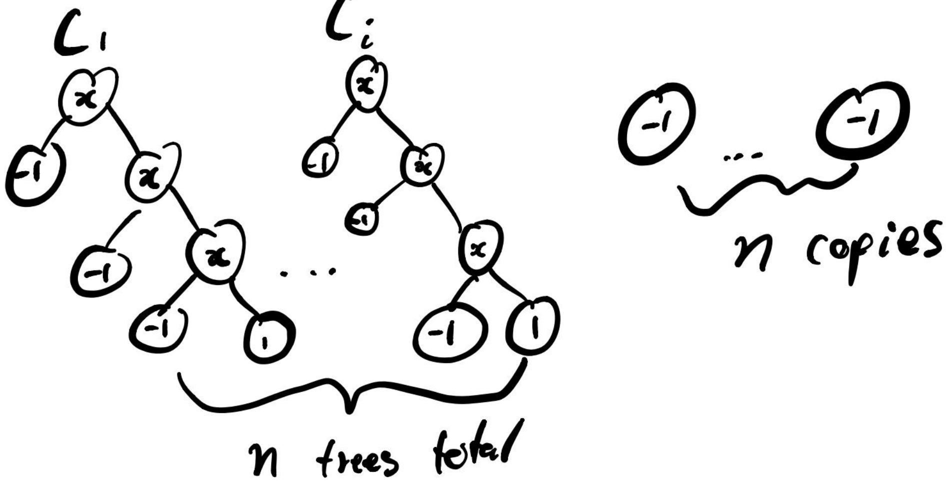
consider clause $\{x_1 \vee x_2 \vee x_3\}$. This clause will only evaluate to true if at least 1 of x_1, x_2 or x_3 is of value 1. We can mimic this behavior by constructing the decision tree as follows:



If we interpret the tree prediction of 1 as true and -1 as false, this tree logic will return true iff and only if at least one of x_1, x_2 or x_3 has value 1, otherwise it will return false, which replicates the behavior of a 3-SAT clause.

c) In order to polynomial time reduce 3-SAT into forest-verify we can consider an arbitrary 3-SAT problem input $C_1 \wedge C_2 \dots \wedge C_n$ where each C_i is an "or" of 3 literals $\in \{x_1, \dots x_m, \overline{x}_1, \dots \overline{x}_m\}$

For each C_i, we create a tree as in part b), which will output -1 if C_i is not satisfied (false) and 1 when C_i is satisfied (true). Additionally, for each C_i, we will add a "negative" tree, which is a tree that will always evaluate to -1. This could be represented as a single leaf node with value -1. Our decision forest would then look as follows (x being the appropriate literal)

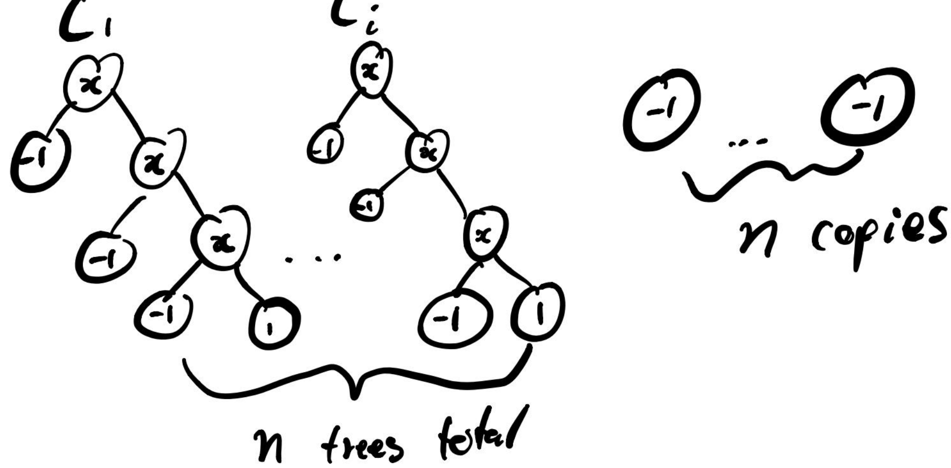


4.2 (b) **6 / 7**

- **- 0 pts** Correct
- **- 0.5 pts** very minor error (see comment)
- ✓ **- 1 pts special case missing: negation operator**
- **- 1 pts** Value assignment is wrong: +1 is true and -1 is false.
- **- 2 pts** Clauses won't contain AND operator
- **- 3 pts** Incomplete answer: after reading your answer, I still don't know how exactly to create such a tree.
- **- 3 pts** A clause is a three-tuple connected with *or* operators, not "and".
- **- 5 pts** Mostly wrong / only answered "Yes" with no correct explanation.
- **- 7 pts** Empty / completely wrong.
- **- 7 pts** Handwriting extremely hard to read (regrade request is welcomed).

If we interpret the tree prediction of 1 as true and -1 as false, this tree logic will return true iff and only if at least one of x_1, x_2 or x_3 has value 1, otherwise it will return false, which replicates the behavior of a 3-SAT clause.

c) In order to polynomial time reduce 3-SAT into forest-verify we can consider an arbitrary 3-SAT problem input $C_1 \wedge C_2 \dots \wedge C_n$ where each C_i is an "or" of 3 literals $\in \{x_1, \dots x_m, \overline{x}_1, \dots \overline{x}_m\}$. For each C_i, we create a tree as in part b), which will output -1 if C_i is not satisfied (false) and 1 when C_i is satisfied (true). Additionally, for each C_i, we will add a "negative" tree, which is a tree that will always evaluate to -1. This could be represented as a single leaf node with value -1. Our decision forest would then look as follows (x being the appropriate literal)



Note that the sum of the C_i trees results will always be n if and only if all clauses are satisfied. The n copies of -1 will then reduce the total forest sum to 0 if and only if all clause trees report 1 (i.e. all 3-SAT clauses are satisfied), causing the

forest verify process to report true since $\sum_{t=0}^{n} f(t) \geq 0$ only when the 3-SAT trees all evaluate to 1. If any 3-SAT tree evaluates to -1, we will get a total forest sum ≤ 0, so forest verify will return false, matching the behaviour of 3-SAT in all cases. Therefore since our modification to the 3-SAT input can be made in polynomial time, we have successfully found a polynomial time reduction from 3-SAT to forest-verify.

4.3 (c) **16 / 16**

✓ **- 0 pts** Correct

- 8 pts Reduction is not polynomial. The ForestVerify problem must be polynomial size. A single tree needs 3x more subtrees per layer (one per +1), giving exponential size. Also not allowed to just do pointers, as the actual ForestVerify problem is still the full exponential tree. (Otherwise you can solve 3SAT in polytime by following a +1 node backwards)

- 12 pts Calling ForestVerify on each tree (clause) only tells you that each tree has some assignment that makes that tree true. Trees representing a single clause are always satisfiable trivially. This has no guarantee that the SAME assignment makes ALL trees true. "Fixing this" takes exponential time, as it would be a solver for 3SAT, since ForestVerify is trivial on clause trees.

- 12 pts Not allowed to modify the evaluation function or node values. Nodes must be +1/-1. Evaluation is always "True if sum >= 0". You can only check if it's in ForestVerify, aka if some assignment makes sum >= 0 or not. Changing any of this makes it not a valid ForestVerify problem. This was stated in the FAQ. (For comparison, you can't for example change what 3SAT being satisfied means, so you can't change what ForestVerify being True means)

- 12 pts Did not account for ForestVerify accepting if half of the trees can be satisfied, while 3SAT requires all to be satisfied. "sum of [+1 or -1] > 0" only needs half to be +1, the other half can be -1. E.g. just stated to use part (b) for every clause, and no other construction.

Close to correct (trying to add extra trees that are always false), but minor issue

- 4 pts Tried to add "always false" clauses then turn them into trees, which can't be done as seen in HW4 Approx. Close enough to the right answer, you can go straight to making "always -1" trees.

- 4 pts Can't have "always false" literals, since the decision problem considers all possible assignments (similar to HW4). Close enough to the right answer, as it's trying to make "always -1" trees.

- 4 pts Can't have trees that are true when all clauses true and false when any clause is false, as that just solves 3SAT.

- 12 pts Defined "half-3SAT", reduction from 3SAT to half-3SAT is missing/wrong. This is not trivial. You can't just repeat Approx from Hw as that adds 7 T and 1F and can't reach 1/2, and you can't add "always false" clauses as seen with the HW.

- 0 pts Other issue (see comment)

- 16 pts Reduction in the wrong direction: Showed Forest < 3SAT, problem asked for 3SAT < Forest

- 16 pts No answer, no reduction/construction provided from multi-clause 3SAT, or extremely incorrect