CS 180 Final

Hanna Co

June 7, 2021

1. In this problem, we are essentially building a maximum spanning tree, but rather than returning the tree, we want to return the set of discarded edges.

Algorithm:

```
Initialize the set of edges that cause a cycle: $S = \emptyset$
Sort the edges from greatest to smallest weight
For each edge in the graph:
    Take the largest edge, $e$
    If adding $e$ creates a cycle, add it to $S$
    Otherwise, add it to our tree
Return $S$
```

Proof of Time Complexity:
Since this is essentially Kruskal's Algorithm, the time complexity is $O(mlogn)$, as proven in lecture. The time complextiy does not change with the modifications because sorting in ascending order has the same time complexity as sorting in descending order, and adding edges to S can be done in constant time.

Proof of Correctness:
Again, as this is essentially Kruskal's Algorithm, we know that it will build the maximum tree, since we used decreasing rather than increasing edge weights. Thus, we will always discard the smallest edges that form a cycle. We can show this by a simple proof by contradiction. Say we have a graph with n nodes, m edges, and there is a single cycle formed by edges x, y, z, w. Let's say our algorithm throws out w, but $w > y$. We know that this isn't possible in our algorithm, since we sort the edges from greatest to least, so we would've added y before w, thus returning $S = \{w\}$. Thus, our algorithm will always return the set of edges with the smallest weight.

2.

(a) This algorithm uses array indexing that begins at 0.

Algorithm:

```
mid = $\frac{n}{2}$
if (mid = 0 or A[mid - 1] < A[mid]) and (mid = n - 1 or A[mid + 1] < A[mid]):
    return mid
else if mid > 0 and A[mid - 1] > A[mid]
    return Hill(A[$0 \sim mid - 1$])
else
    return Hill(A[$mid + 1 \sim n - 1$])
```

Proof of Time Complexity:

At every step, we halve our array, so the time complexity is $O(logn)$.

Proof of Correctness:

This algorithm checks to see if the middle element is the hill. If it is, then we can simply return it. Otherwise, we check and see if it's smaller that its left element - if it is, the hill will be on the left, so we run it on that half. If not, then the hill will be on the right, so we run the algorithm on the right half. Checking whether the middle element is a hill is intuitive, so we skip past that, and do a simple proof that shows that our algorithm will always find a hill. Let's say we narrowed our problem down to the left subarray. The hill is guaranteed to be on this side, because if A[mid - 1] is not the hill, then the element on it's left (A[mid - 2]) is greater. If this element (A[mid - 2]) is not the hill, then the element on its left is greater, and so on, until we reach the leftmost element (A[0]), in which case that must be the hill. We can use the same argument for the right side of the array, thus showing that if $A[mid - 1] > A[mid]$, then the hill must be on the left, and if $A[mid + 1] > A[mid]$, then the hill must be on the right.

(b)

Algorithm:

```
mid = $\frac{n}{2}$
find the maximum element, $max$, in the middle three columns
if $max$ is in the middle column, return index of $max$
```

```
else if $max$ is in the left-middle column:
    return Hill(A[$0 \sim mid$][$0 \sim n-1$])
else if $max$ is in the right-middle column:
    return Hill(A[$mid \sim n-1$][$0 \sim n-1$])
```

Proof of Time Complexity:

Finding the maximum element in the middle three columns takes $3n$ iterations, simplified to $O(n)$. If we don't find the hill on the first iteration, the recursive call takes $O(\frac{n}{2})$ time, halving with each following call to Hill(). Thus we have the recurrence relation $T(n) = T(\frac{n}{2})$, giving us a time complexity of $O(n)$.

Proof of Correctness:

The proof for this is very similar to the 1D case. If our hill isn't in the center, then the hill must be on the side with the larger element. The reason we are able to simply use the maximum element from each column is also similar to the 1D case, where if the max isn't the hill, then the hill must be on either the left or right side, which shrinks our problem until we either find the hill, or the we reach the end of the array (or in this case, matrix).

3.
This is a dynamic progamming problem, except when we increase the number of cities or fire stations, the previous solution may not hold. We can solve this by having subproblems that have anywhere from 1 to $n-1$ cities, and $K-1$ fire stations. For each subproblem, we take the one with minimum distance from a city to the closes fire station.

Algorithm:
Let OPT(n, K) be the optimal placement when there are n cities and K fire stations, along with total distance
Let dist(i) be the total distance from x_i to cities x_{i+1} to x_n.
We know $OPT(1, K) = dist(1)$.

```
for $i = 2 \sim n$
    for $j = 1 \sim K$
        if $i \leq j$
            OPT(i, j) = 0
        else
            OPT(i, j) = $min_{1 \leq h < n}\{OPT(h, K-1) + dist(i)\}$
```

The optimal solution is stored at OPT(n, K), and the average distance would be $\frac{OPT(n,K)}{n}$.

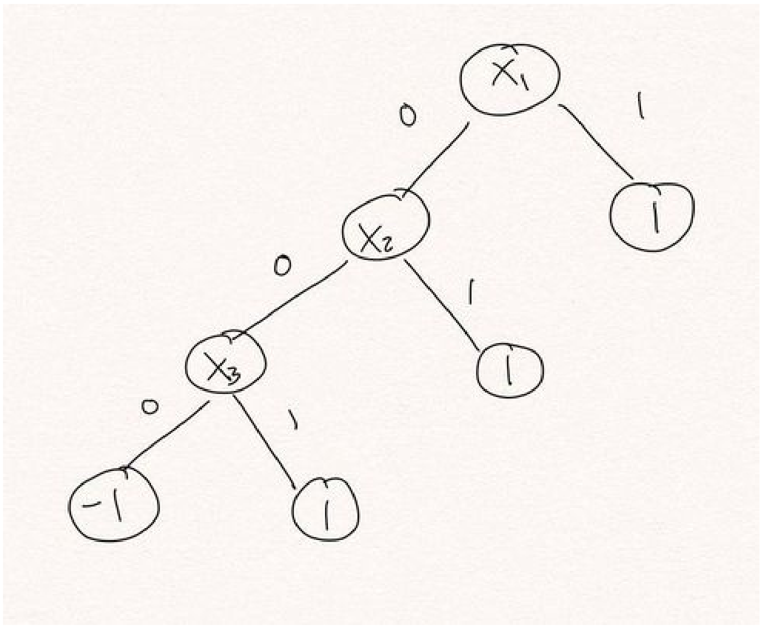
Proof of Time Complexity:
We have three nested loops, going from $2 \sim K, 1 \sim n$, and $1 \sim n$. Simplified, these loops runs n^2K interations in the worst case, thus our time complexity ifs $O(n^2K)$.

Proof of Correctness:
We start with the simple case: one city. Regardless of the value of K, there is only one placement for fire stations. For all other subproblems, we place one fire station at the rightmost city, and find the most optimal placement of $K-1$ fire stations, factoring in how the total distance changes with each fire station placement.

4.

(a) For a problem to belong in NP, it must have a polynomial time certifier. Forest-Verify has a certifier, C, that takes in a string x. It can use that string to traverse the forests in polynomial time (number of trees $*$ max depth of trees), and return whether or not x is a solution. Thus, since Forest-Verify had a polynomial time certifier, it is NP.

(b) Yes, this is possible by turning literals $(x_1, x_2, ..., x_n)$ into nodes. We can assign -1 and 1 in a way where, if the prediction of the decision tree outputs true, it satisifes the clause. Otherwise, it does not. For example, say we have the 3-SAT clause $x_1 \bigcup x_2 \bigcup x_3$, where the proper literal assignment is x_1 = true, x_2, x_3 = false. The following is a possible decision tree:



(c) A polynomial time reduction from 3-SAT to Forest-Verify will essentially solve 3-SAT using Forest-Verify. The main difference between the two is that Forest-Verify returns true if half or more trees are *true*, where 3-SAT returns true if all clauses are *true*. We use the strategy from (b) to expand each clause into a decision tree. Say that there are n trees. We then must add n more trees that always result in false (we can do this simply by adding trees that don't have 1 assigned to any of the inputs, thus it will always return -1). We can examine the two cases to prove that this reduction works.

Case 1: returns true

The sum of all the predictions can never be greater than 0, because that would mean more than half the trees are satisfiable, which isn't possible, given our modification. This means that excactly half of the trees are satisfiable, and the other half isn't. The half that isn't satisfiable is the half that contains the n "always false" trees, thus meaning that the original n trees created from 3-SAT are satisfiable.

Case 2: returns false

In this case, less than half the trees are satisfiable. This means that, in addition to the n trees we added, there are some trees in the original n that aren't satisfiable. Thus, 3-SAT should have also returned false.