CS 170 Efficient Algorithms and Intractable Problems
February 29, 2016 Liam Leahy

HW 6

Due February 29, 11:59pm

Partner: Stephen Qu

1. (15 pts.) Set Cover
In class we saw a greedy algorithm for finding an approximate solution to the set cover problem. In this exercise, you will show that the approximation ratio we found for that algorithm is tight. Prove that for any integer k greater than 1, there is an instance of the set cover problem such that:

i. There are $2^k - 2$ elements in the base set

ii. The optimal algorithm uses just two sets

iii. The greedy algorithm picks at least k sets

Note: "base set" just means the union of all the sets in the instance (sometimes also called the "universe"). Also, you need to prove that for each k greater than 1 there is some instance of the set cover problem that satisfies all three conditions at once.

Solution:

Proof: For the first case, this is trivial as we have two elements ($2^2 - 1$) to make 2 sets which will be grouped into two sets by the optimal and the greedy.

Assume this works for the $2^k - 2$.

For $2^{k+1} - 2$, we have 2^k additional elements. We group these equally among the two sets for the kth iteration solution. This means that the optimal solution will still only be two sets. The greedy solution will choose to group the 2^k new elements first as the max elements that it can choose otherwise would be $2^k - 1$ from the optimal solution from the kth iteration. There are at least k sets remaining from the rest of the graph that the greedy algorithm used, therefore there are now at least $k+1$ sets that the greedy algorithm finds (k from the kth iteration and one from the extra elements this iteration). □

2. (15 pts.) Amortized Analysis

Recall that Queues and Stacks are both data structures that support adding and removing elements, and that in a Queue items are removed in FIFO order (that is, the item removed is the first item to enter the Queue) and in a Stack items are removed in LIFO order (that is, the item removed is the most recent item added to the Stack). As it turns out, it is possible to implement a Queue using two Stacks according to the procedure outlined below (you should take a moment to verify for yourself that this correctly simulates a Queue).

```
INITIALIZATION
1. Initialize two empty Stacks, $S_1$ and $S_2$

ADD(X)
1. Push x onto $S_1$

REMOVE()
1. If $S_1$ is empty and $S_2$ is empty:
2.     Return null
3. If $S_2$ is empty:
4.     While $S_1$ is not empty:
5.         Push $S_1$.pop() onto $S_2$
6. Return $S_2$.pop()
```

In this question you will analyze the amortized running time of the above operations.

(a) Suppose that every time an element x is added to the Queue we put some money beside it. Any time x is pushed onto or popped off of one of the Stacks, we must pay one dollar. How much money do we need to put next to x to ensure that we always have enough money to pay for all pushes and pops of x (including the initial push onto S_1)? Prove your answer correct.

(b) Using part (a), show that the running time of a series of n add and n remove operations, starting from an empty Queue and taking place in any order, takes $O(n)$ time.

Solution:

(a) 4 dollars. The first dolar is necessarily for the initial push onto S_1. The second is for the inevitable pop from S_1. The third is for the immediate push onto S_2. The last is for the eventual pop from S_2.

These must occur in this order exactly this many times because any element is always initially added to S_1. The only time it is removed from S_1 is in order to be put into S_2. The only time that you obtain this item is once it is poped from S_2. This makes sense because S_2 is essentially a container for some subset of the elements in the correct order to be returned. S_1 is a container to initially hold the elements that we add to the Queue but they are in reverse order. In order to get them in the right order we have to add them to S_2, one at a time, backwards (this is why we have S_1 house the elements in reverse). This would normally ruin the order in S_2 if any elements are in S_2. Therefore, we only add elements to S_2 once it is empty. We mind as well dump the entire contents of S_1 in order to increase the efficiency. Therefore, the general process is initially into S_1, then move from S_1 into S_2 once S_2 is empty in order to reveres the order. Finally, we remove from S_2 in order to finally get the element in the order that it should come out.

(b) The total number of dollars that we can use for any element is 4. We add n elements and remove at most n elements. Since the limit on the amount of money possibly used for each element is 4, we know that the maximum number of operations total is $4n + k$ where k is the number of removes that we do when the Queue is empty (since it depends on the order). Therefore, the running time is in $O(n)$.

3. (20 pts.) Updating MSTs

You are given a graph $G = (V, E)$ with positive edge weights, and a minimum spanning tree $T = (V, E')$ with respect to these weights; you may assume G and T are given as adjacency lists. Now suppose the weight of a particular edge $e \in E$ is modified from $w(e)$ to a new value $\hat{w}(e)$. You wish to quickly update the minimum spanning tree T to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each, give a linear time algorithm for updating the tree. For each, use the four part algorithm format. However, for some of the cases the main idea, proof and justification of running time may be quite brief. Also, you may either write pseudocode for each case separately, or write a single algorithm which includes all cases.

(a) $e \notin E'$ and $\hat{w}(e) > w(e)$

(b) $e \notin E'$ and $\hat{w}(e) < w(e)$

(c) $e \in E'$ and $\hat{w}(e) < w(e)$

(d) $e \in E'$ and $\hat{w}(e) > w(e)$

Solution:

Lemma 1 *Given some undirected MST T and two vertices $(u, v) \in T$, it is the case that there is only oone path through T from u to v.*

Proof: By the definition of an MST, there must be a path from u to v, call this path p. If there is another path p', there is some point, δ, that the other path diverges from p. There must also be a point, m, at which the paths merge since they end at the same point v. This will necessarily create a cycle in T as you could go from δ to m since T is undirected. □

(a) **Main Idea:** In this case, it is not necessary to update anything as the MST cannot change.

Pseudocode:

Procedure 1 Given G, the original graph, T, the original MST $\hat{e}$ the edge that is updated, and $\hat{w}$ the new weight for $\hat{e}$.

```
1: function CASE_A($G, T, \hat{e}, \hat{w}$)
2:     return
```

Proof: If T is already the MST of G before, increasing the weight of an edge in G does not change the MST at all. An MST is defined to be the minimum weight spanning tree. We already have one MST, T, which is known to be the MST for G. Increasing the weight of some edge in G that is not in T should not change T as it only increases the weight of G. □

Running Time: $O(1)$.

Justification: Since (according to Piazza) we are given whether or not $\hat{e}$ is in E', we do not have to check that and therefore doing nothing takes constant time (otherwise it would be linear).

(b) **Main Idea:** In this case, it is possible that the edge that is updated changes the possible MSTs. Therefore, we have to check if an improvement can be made to T. Let $\hat{e} = (u, v)$. We search for the longest edge e^* on the path in T that connects u and v. If e^*'s weight is greater than $\hat{w}$, then we replace e^* with $\hat{e}$.

Pseudocode:

Procedure 2 Given G, the original graph, T, the original MST $\hat{e}$ the edge that is updated, and $\hat{w}$ the new weight for $\hat{e}$.

```
1: $prev \leftarrow$ dictionary for every $v$ in $V$
2: function CASE_B($G, T, \hat{e} = (u, v), \hat{w}$)
3:     VISITED($v$) = FALSE
4:     EXPLORE($G, v$, NULL)                ▷ there is no prev for $v$, since we start there
5:     $runningPosition = v$               ▷ at this point, we have $prev[v]$ for every $v$ in $V$
6:     $path = []$
7:     $maxEdge = runningPosition$
8:     $maxValue = -\infty$
9:     while $runningPosition$ does not equal $u$ do
10:        APPEND($path, runningPosition$)
11:        $runningPosition = prev[runningPosition]$
12:        if W(runningPosition) $> maxValue$ then
13:            $maxEdge = runningPosition$
14:            $maxValue =$ W($runningPosition$)
15:    if $maxWeight > \hat{w}$ then
16:        REMOVE_EDGE($T, maxEdge$)
17:        ADD_EDGE($T, \hat{e}$)
18: function EXPLORE($G, u, v$)
19:     for all edges $(u, v) \in E$ do
20:         if not VISITED($v$) then
21:             $prev[v] = u$                ▷ change the value for $v$ in $prev$ to be $u$
22:             EXPLORE($v$)
```

Proof: Let $\hat{e} = (u, v)$. We know that $\hat{e}$ is not in T. Since it is the case that $\hat{w}$ decreased, it is possible that the MST is changed. We need to know how T connected u and v in order to assess whether or not replacing some edge in T with $\hat{e}$ is beneficial. In order to do this, we perform DFS on T from u to v. If it is the case that an edge, e^* along this path has greater weight than $\hat{e}$, then it is necessarily the case that there exists another MST, identical to T in every respect except that $\hat{e}$ replaced e^*, that is of lower weight than T. Therefore, we replace e^* with $\hat{e}$.

This works and is still an MST because:

T was definitely an MST before. Therefore, we know that u and v were connected via some path p. p is also necessarily the only path from u to v in T by Lemma 1. The only way that the change to $\hat{e}$ changes T is the case in which it is now included in T. This is only possible if we remove an edge from T. There are only a subset of the edges in T that are possible to be removed and replaced by $\hat{e}$. We have to remove an edge that is on the only path from u to v as otherwise adding $\hat{e}$ would create a cycle. The only way adding $\hat{e}$ is beneficial is when $\hat{w}$ is less than the weight of the greatest weight edge on p. As this would maintain all the MST properties of T but would decrease the weight. Therefore, removing e^* and replacing it with $\hat{e}$ would only be necessary and possible under the condition that the weight of e^* is greater than the weight of $\hat{e}$.

Removing e^* necessarily disconnects T. This can be shown simply by taking Lemma 1 and recognizing that removing e^* removes an edge on p from u to v, necessarily disconnecting them. If we reconnected u and v then the MST would be connected again and there would necessaily not be any cycles since we removed the only other path that could make a cycle through T beforehand.

□

Running Time: $O(|V|)$.

Justification: DFS is $\in O(|V|+|E|)$ and in an MST $O(|E|) = O(|V|) \implies O(|V|+|E|) = O(|V|)$. Traversing the DFS tree is equivalent to following at most the max path through the tree, which is in $O(|V|)$. Thus the final running time is $O(|V|)$.

(c) **Main Idea:** In this case, this simply improves our MST and therefore we do not need to change it at all.

Pseudocode:

Procedure 3 Given G, the original graph, T, the original MST $\hat{e}$ the edge that is updated, and $\hat{w}$ the new weight for $\hat{e}$.

```
function CASE_C($G, T, \hat{e}, \hat{w}$)
    return
```

Proof: If the only edge that changes is $\hat{e}$ and $\hat{e} \in E'$, this implies that the weight of the MST decreases by the differences in weights. If there is any other MST that is at most as good as T, the best that it can do is improve by this same difference. Therefore, T does not have to change. □

Running Time: $O(1)$.

Justification: Doing nothing is, you know, pretty constant.

(d) **Main Idea:** Once again, it is possible now that the MST changed because the weight of an edge in E' has increased. Now, our approach will take advantage of the cut property to disconnect T at $\hat{e}$, find the two disjoint trees, T_1, T_2, that result from this disconnection and then find the new edge e^* such that we can reconnect T using the least cost edge between T_1 and T_2.

We can use vanilla DFS and then simply use the visited list generated from DFS in order to find the subsets of V that are visited in each call.

Pseudocode:

Procedure 4 Given G, the original graph, T, the original MST $\hat{e}$ the edge that is updated, and $\hat{w}$ the new weight for $\hat{e}$.

```
function CASE_D($G, T, \hat{e} = (\hat{e}, \hat{v}), \hat{w}$)
    $visited = []$
    REMOVE_EDGE($T, \hat{e}$)
    DFS($T, u$)
    $T_1 \leftarrow visited$
    $visited = []$
    DFS($T, v$)
    $T_2 \leftarrow visited$
    $minEdge = \hat{e}$
    $minWeight = \hat{w}$
    for all edges $e = (u, v) \in E$ do
        if $u \in T_1$ and $v \in T_2$ or $u \in T_2$ and $v \in T_1$ then
            if W(e) $< minWeight$ then
                $minWeight =$ W(e)
                $minEdge = e$
    ADD_EDGE($T, minEdge$)
```

Proof: The only case in which the change to $\hat{e}$ affects T is the case in which T should include some other edge than $\hat{e}$. Therefore, we can remove $\hat{e}$ from T and define the two trees created from the removal as two disjoint trees, T_1 and T_2. By the cut property, it is the case that the min edge connecting these cuts is an edge in the MST. Therefore, we find this edge and add it to T. □

Running Time: $O(|E|+|V|)$.

Justification: DFS on an MST takes $O(|V|)$ time as we showed in (b). We do this twice. Subsequently, we iterate through all edges in E. This takes $O(|E|)$ time. They are done sequentially so we can add their running times to obtain $O(|E|+|V|)$.

4. (15 pts.) Horn Implementation

In class, we saw a greedy algorithm to find a satisfying assignment for a Horn formula, but we did not actually see how to implement it efficiently. Show how to implement the stingy algorithm for Horn formula satisfiability in time linear in the length of the formula (the number of occurrences of literals in it). You should use the four part algorithm format, but your proof of correctness can just involve showing that your pseudocode is doing the same thing as the algorithm we saw in class.

Solution:

Main Idea: Do almost exactly what it says in the book. All we have to do is make an adjacency matrix in order to store the relationships for every implication expression. For any single implication (one that must be true), we remove all of the implications from the adjacency list because they must be true. If this results in a single implication, we have to consider it as one and repeat. The rest of the algorithm works as in the book.

Pseudocode:

Procedure 5 I is the list of all implications, N is the list of all negative clauses, V is the list of all variables.

```
1: function HORN($I = [i_1, i_2, \ldots, i_n], N = [n_1, n_2, \ldots, n_m], V = [v_1, v_2, \ldots, v_l]$)
2:     $Q \leftarrow$ queue for single-literal implications
3:     $a \leftarrow []$                          ▷ a list of lists or adjacency list for the variables and implications
4:     $assignments = []$                         ▷ a list of assignments for all variables
5:     for all $i \in I$ do                       ▷ to find all the single-literal implications
6:         if $i$ contains only one literal then
7:             ADD$(Q, i)$
8:         for all variables $u \in i$ do
9:             $assignments[u]$ = FALSE
10:            APPEND$(a[u], i)$                  ▷ add implication $i$ to list for variable $u$
11:    while NOT_EMPTY$(Q)$ do
12:        $q \leftarrow$ POP$(Q)$                ▷ this is a single-literal implication, treat $q$ as this literal
13:        $assignments[q]$ = TRUE
14:        for all implications $i \in a[q]$ do
15:            REMOVE_LITERAL$(i, q)$
16:            if $i$ is now a single-literal then
17:                ADD$(Q, i)$
18:    for all $n \in N$ do
19:        if NO_SATISFYING_ASSIGNMENT$(n, assignments)$ then
20:            return FALSE                       ▷ there is no satisfying assignment
    return TRUE
```

Proof: The only thing that this does that the book's solution doesn't is that it does what the book does in the while loop in linear time. This is done by keeping track of all of the single-literal expressions known at any point in time and all implications that contain a variable on the left for any variable. If it is the case that we have a single-literal, then it must be the case that any implication with this literal on the left have this variable satisfied. Thus, we can remove this variable from the implication. If this implication is now a single-literal, then we add it to the queue. This considers every variable only a constant number of times.

The while loop cannot loop more than the number of times that there are variables (this is the case that we eliminate every variable from the left for every implication). We do a constant amount of work for every iteration of the loop (since for every variable in the queue, there is necessarily one less implication in the list for each variable).

The logic as to why this works is the same as the book. □

Running Time:

$O(n)$.

Justification: We do linear work with respect to the number of literals in place of the while loop from the book. There are at most n literals in the queue but for every literal in the queue, there is one less implication to consider in the for loop. This yields linear running time for the entire while loop. The rest is the same as the book. Therefore, the final running time is linear.

5. (15 pts.) Ternary Huffman

Trimedia Disks Inc. has developed "ternary" hard disks. Each cell on a disk can now store values 0, 1, or 2 (instead of just 0 or 1). To take advantage of this new technology, provide a modified Huffman algorithm for compressing sequences of characters from an alphabet of size n, where the characters occur with known frequencies $f_1, f_2, \ldots, f_n$. Your algorithm should encode each character with a variable-length codeword over the values 0, 1, 2 such that no codeword is a prefix of another codeword and so as to obtain the maximum possible compression. Your proof of correctness should prove that your algorithm achieves the maximum possible compression.

Solution:

Main Idea: Ternary would be the almost same case as binary except there are three elements per subtree. The weird case in the case in which there are an even number of values. In this case, we have to add an extra "null" node in order to avoid an ineffieciency explained in the proof.

Pseudocode:

Procedure 6 Find the Ternary Huffman encoding that achieves maximum possible compression.

```
function TERNARY_HUFFMAN($f = [f_1, \ldots, f_n]$)
    $trees \leftarrow$ minimum priority queue ordered by element weight
    for all integers $i$ in the range of 1 to $n$ do
        INSERT($trees$, TREE($f[i]$))                ▷ makes new tree of element
    if there are an even number of elements in $trees$ then
        INSERT($trees$, TREE(0))        ▷ "null" node for the inefficiency mentioned below
    while the number of elements in $trees$ is greater than 1 do
        $i, j, k \leftarrow$ REMOVE_MIN($trees$), REMOVE_MIN($trees$), REMOVE_MIN($trees$)
        $newTree \leftarrow$ TERNARY_TREE($i, j, k, i+j+k$) ▷ makes new ternary tree of $i, j, k$ where weight is $i+j+k$
        INSERT($trees$, $newTree$)        ▷ if NULL, this is the end so it doesn't matter at all
    return the only tree left in $trees$
```

Proof: The first case to consider is the case in which there are an even number of nodes. In this case, we would continually remove 2 trees but in the end be left with two on the final iteration. Combining these trees would be inefficient as it would leave a blank slot for the last ternary combination. This "empty slot" could be considered to be a node of value 0. Therefore, by the exchange argument, there is a superior solution. To satisfy this issue, we insert an extra node of value 0 at the beginning (in the case that there are even number of nodes) in order to necessarily be used in the first ternary tree made such that the "null" node is at the bottom of the tree. This is the optimal solution by the exchange argument mentioned above. I.e., if the "null" node was anywhere else in the tree, there is a superior solution that involves swapping the null node for whatever is lower in the tree.

If there are an odd number of nodes, then we continually remove 3 until we have 1, in which case we are done by the base case and thus this ineffieciency only occurs in the even case.

This is exactly the same thing as in the case for binary Huffman encoding except there are three nodes now instead of 2. There is, fundamentally, no modification to the functioning of this algorithm with respect to the code shown in class. The "null" node does not change the result of the algorithm other than that it must exist in the lowest portion of the tree. It is not a "result" as it will never be used (we do not use the empty character, for instance, if we are using this encoding). But it is necessary to include to avoid the ineffieciency of being left with two trees as your final iteration. □

Running Time: $O(n \log n)$.

Justification: The for loop trivially runs in linear time.

For the while loop, there are always three elements removed from $trees$ until there are less than 3 elements left (in which case this is the last iteration). Therefore, the total number of iterations for this loop is on the order of $\frac{n}{3} \in O(n)$. For each iteration, there are three logarithmic operations to the priority queue. Therefore, the running time is in $O(n \log n)$.

6. (20 pts.) Finding Palindromes

A subsequence is *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsquences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is *not* palindromic). Devise an algorithm that takes a sequence $A[1...n]$ and returns the length of the longest palindromic subsequence. Its running time should be $O(n^2)$.

Note: A subsequence does not have to be contiguous in the original sequence. For instance, A, G, G, A is a subsequence of the above sequence.

Solution:

Main Idea: There are n^2 possible pairs of indices. In order to find the answer, we need to find the longest possible palindrome between any pair of indicies. If we construct a 2-dimensional array, we can iterate through the possibilities from the "bottom up" in order to find the optimal answer.

Pseudocode:

```
function FIND_PALINDROME($S = [s_1, s_2, \dots, s_n]$)
    $a \leftarrow [n \times n]$                ▷ 2-dimensional square array of side length $n$
    for all $i$ in range $n$ do
        $a[i][i] \leftarrow 1$
    for $i \in$ RANGE($n$) do
        for $j \in$ RANGE($n$) do
            if $i$ is not less than $j$ then
                $a[i][j] \leftarrow 0$
            else if $i == j$ then
                $a[i][j] \leftarrow 1$
            else if $S[i] == S[j]$ then
                $a[i][j] \leftarrow 2 + a[i-1][j-1]$
            else
                $a[i][j] \leftarrow$ MAX($a[i-1][j], a[i][j-1]$)
    return $a[0][n-1]$
```

Proof: There are at most n^2 pairs of vertices. If we know the longest palindrome between each pair of vertices then we necessarily know the longest solution. Running this recursively adds a lot of work. Therefore, we memoize the results and use them when we find the same subproblem that needs to be computed again.

This solution uses an array of the results. The indicies of the array correspond to the indicies of the substring of the string that we are considering and the value at this pair of indicies corresponds to the length of the longest palindrome from the first index to the other. We can work from the bottom up to find the solution in linear time with respect to the number of pairs of vertices, solving each subproblem prior to trying to solve the next subproblem.

This works because it is always the case that the longest palindrome of any string is either 2+ the max palindrome of the inner string (in the case that the ends are equal) or the maximum of the longest palindrome of the substring ignoring the first index and the substring ignoring the last index. This is exactly what this algorithm does using the array as a reference. This is true because if the two ends are the same, then the longest palindrome must include them. Otherwise, the longest palindrome cannot include them and therefore, the longest palindrome is either the longest palindrome ignoring the first one or vice versa.

The 2 is added in the first case because the ends are equal and therefore they can and must be used in the longest palindrome. Otherwise, it must be the case that one of them is excluded and therefore we max over the only two possibilities.

All dependencies are resolved when we reach i, j if we work from the bottom left going right and up in that order such that each iteration only requires a constant amount of work.

We return the top right as this corresponds to the longest palindrome going from index 1 to n. $\square$

Running Time: $O(n^2)$.

Justification: There are at most n^2 elements in a that need to be computed. Each computation takes constant time. Therefore, the total running time is in $O(n^2)$.