CS 170 Efficient Algorithms and Intractable Problems
Spring 2016 Alessandro Chiesa and Umesh Vazirani

HW 6 Solution

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:

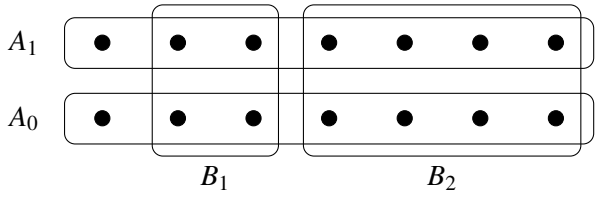
i. There are $2^k - 2$ elements in the base set

ii. The optimal cover uses just two sets

iii. The greedy algorithm picks at least k sets

Solution:

The solution is most easily explained via picture. Shown below is the solution for $k = 4$:



For every k, we create an instance I_k, with base set $\{1,2,3,\ldots,2^k-2\}$ (The picture above illustrates I_4). The optimal cover is A_0, A_1 where A_0 consists of all odd numbers in the base set and A_1 consists of all even numbers in the base set (the horizontal boxes in the above picture). The other sets in the instance will be denoted $B_1, B_2, \ldots, B_{k-2}$, with $|B_j| = 2^{j+1}$, and we will ensure that the greedy algorithm will always choose all sets (including A_0 and A_1).

Note that in the first step, greedy picks the largest set B_{k-2} (in the picture above B_2), which is of size 2^{k-1}. Since the B_i's are disjoint, the remaining set of $2^{k-1}-2$ elements must be covered by $B_0, \ldots, B_{k-3}$ and the even/odd sets A_0, A_1. The crucial observation is that deleting all elements of B_{k-2} (from the base set as well as all other sets in the instance) leaves us with the instance I_{k-1}. A simple induction now shows that greedy will keep picking the B_j's in turn for $j = k-2, \ldots, 1$. At the end, it still must pick the A's, thus giving a grand total of k sets.

More formally, we will prove by induction on k that greedy picks k sets on instance I_k.

The base case is when $k = 2$, the base set is just $\{1,2\}$, and there are just two sets $A_0 = 1$ and $A_1 = 2$. Clearly the optimal cover and the cover chosen by the greedy algorithm both consist of two sets.

Assume for induction that greedy picks $k-1$ sets on instance I_{k-1}. Now, on instance I_k, greedy starts by picking the largest set B_{k-2}. Eliminating all elements of B_{k-2} from I_k results in the instance I_{k-1}, and by the induction hypothesis, greedy picks $k-1$ more sets on this instance, for a grand total of k.

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? Prove your answer correct.

Solution:

Note that once an element x is popped off of S_1 and pushed onto S_2, it is not removed from S_2 until it is removed from the entire Queue. Thus x is popped from S_2 at most once and pushed onto S_2 at most once. And since the only time x is pushed onto S_1 is when it is added to the Queue, it is pushed onto S_1 at most once. And since if x is popped from S_1 it must first have been pushed onto S_1, x is popped from S_1 at most once. Thus x undergoes a total of at most 4 pushes and pops, so it suffices to put $\boxed{\$4}$ next to x.

(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:

Note that the total running time is proportional to the total number of pushes and pops onto/off of S_1 and S_2, plus a constant amount of additional work for every call to remove (to account for the calls to remove when the Queue is empty). And we showed in part (a) that if we put \$4 next to every element that enters the Queue, we will always have enough money to pay for all pushes and pops of that element. Thus if we add n elements, we only need at most $\$4n$ to pay for all pushes and pops, so the total running time is proportional to $\$4n$ plus an additional n for the remove operations– i.e. $O(n)$.

Comment: The purpose of this problem was to show how amortized analysis can make analyzing the running time of a series of operations relatively easy when it might otherwise be tricky. In this case, a single "remove" operation can take up to $\Theta(n)$ time if n elements are in S_1 and S_2 is empty. On the other hand, for every expensive remove operation, there are many subsequent cheap remove operations since we can simply pop a single element off S_2 for each remove until S_2 becomes empty again. It is not immediately clear how to take this tradeoff between expensive and cheap remove operations into account. Amortized

analysis nicely sidesteps the whole issue by simply examining the life of a single element, which is much easier to analyze. To more fully appreciate the power of amortized analysis, please look at the analysis in the textbook of the running time of the union-find data structure.

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 pseduocode 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:

Main Idea

When working with MSTs, there are usually two things to try: look at cycles and look at cuts. When trying to decide if an edge will improve a spanning tree, we look at the cycle it forms when added to the MST. When trying to decide whether a spanning tree can be improved by removing an edge, we can look at the cut formed when we remove the edge from the tree. This is exactly what the algorithm below does.

Pseudocode

UPDATE MST($G = (V, E)$, $T = (V, E')$, w, e, $\hat{w}(e)$)
// Case 1
1. If $e \notin E'$ and $\hat{w}(e) > w(e)$: pass
// Case 2
2. If $e \notin E'$ and $\hat{w}(e) < w(e)$:
3. Add e to T and use BFS to find a cycle in T
4. Remove some heaviest edge in this cycle from T
// Case 3
5. If $e \in E'$ and $\hat{w}(e) < w(e)$: pass
// Case 4
6. If $e \in E'$ and $\hat{w}(e) > w(e)$:
7. Remove e from T creating two connected components of T, i.e. a cut of G
8. Use BFS to find every vertex on one side of this cut
9. Add the lightest edge crossing the cut to T

Proof of Correctness

Case 1: Note that increasing the weight of e either increases or doesn't affect every spanning tree of G. Since it doesn't affect T, it must still be an MST.

Case 2: First note that if an edge is not in an MST then it must be the heaviest edge in some cycle. Since no edges increased in weight, for every edge not in E' besides e, this is still the case. So we only have to decide if we need to add e to the tree, which is exactly what the algorithm does.

Case 3: Note that the weight of every spanning tree of G either stayed the same or decreased by $w(e) - \hat{w}(e)$. Since the weight of T decreased by $w(e) - \hat{w}(e)$ it is still an MST.

Case 4: First note that if some edge is in an MST then it must be the lightest edge across some cut (see discussion 5). Since no edges decreased in weight, for every edge in E' besides e this is still the case. So we only need to decide if we should remove e and if so, what edge should replace it. But we can only remove e if it is not the lightest edge across the cut examined by the algorithm, and if it is not then we must replace it by the lightest edge, which is just what the algorithm does.

Running Time First note that we can detect which case we are in simply by looping over the edges in T and checking if they are equal to e. Since T has $|V| - 1$ edges, this takes linear time.

Cases 1 and 3 take constant time since they do nothing. For case 2, it takes $O(|V|)$ time to run BFS on $T \cup \{e\}$ (since T has $|V| - 1$ edges) and find the heaviest edge in the cycle. In case 4, it takes $O(|V|)$ time to run BFS on $T - \{e\}$ in order to find the cut that only e (out of all the edges of T) crosses and then $O(|V| + |E|)$ time to find every edge crossing this cut and pick the lightest one (since in the worst case all but $|V| - 1$ edges in the graph cross the cut).

Thus the overall running time is $O(|V| + |E|)$. In fact, if we are not in case 4 then the running time is $O(|V|)$.

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 The only place where the algorithm from class does not seem efficient is in the loop where we repeatedly identify an unsatisfied implication and set a variable to true to satisfy it. More precisely, we identify an implication such that every variable in the conjunction on the LHS (left hand side) of the implication is set to true, but the variable on the RHS (right hand side) is false in the current truth assignment. To satisfy the implication, we then set the RHS variable to be true. The challenge is to keep track of which new implications become unsatisfied as a result of setting this variable to true.

We note that a simple way to solve this problem is to keep a count, for each implication, of the number of false variables on the LHS. When we set a variable x to true, we decrement the count for every implication such that x occurs on the LHS of that implication. When the count for an implication decreases all the way to 0, we check whether the RHS is set to false. If so, the implication is unsatisfied.

Algorithm

In the algorithm below, $I_1, I_2, \ldots, I_m$ indicate the implications and $N_1, N_2, \ldots, N_l$ the negative clauses of a horn formula with variables $x_1, x_2, \ldots, x_n$.

HORN SOLVER($I_1, I_2, \ldots, I_m, N_1, N_2, \ldots, N_l$):

1. Initialize an array of linked lists, *Implications*, of length n
2. Initialize an array *Counts* of length m
3. Set all variables to False and initialize an empty queue Q
4. For $i = 1$ to m:

5. Set $Counts[i] = 0$
6. For each variable x_j that appears in the left hand side of I_i:
7. Increment $Counts[i]$ and add i to $Implications[j]$
8. If $Counts[i] = 0$, add the variable on the right hand side of I_i to Q
9. While Q is not empty:
10. Remove x_j from Q
11. If x_j is False:
12. Set x_j to True
13. For each implication I_i in $Implications[j]$:
14. Decrement $Counts[i]$ and if $Counts[i] = 0$ add the variable on the right hand side of I_i to Q
15. If all negative clauses are satisfied, report the assignment
16. Else report "unsatisfiable"

Proof of Correctness We note that once a variable is set to true its truth value is never again changed. Now a simple proof by induction (on the number of times through the main loop) shows that the count associated with each implication is the number of false variables on the LHS in the current truth assignment. If follows that an implication is unsatisfied if and only if its count is 0 and the variable on its RHS is currently set to false, thus justifying this approach for identifying unsatisfied clauses.

If a more detailed, line by line proof of correctness is desired, we prove that for each iteration of the while loop starting on line 9, the algorithm either does nothing or picks some unsatisfied implication and sets the variable on its right hand side to True, and that the while loop does not end unless all implications are satisfied. This is accomplished by proving the following loop invariants for the while loop:

At the beginning of each iteration,

- For each $i \leq m$, exactly $Counts[i]$ of the variables on the left hand side of I_i are set to False
- If some implication I_i is unsatisfied, then the variable on its right hand side is in Q
- If a variable is in Q then either it is already set to True or it is on the right hand side of some unsatisfied implication

Note that all three of these conditions are met before the first iteration of the while loop. Assume for induction that they hold true at the beginning of some iteration of the while loop, and we will show that they still hold at the beginning of the next iteration. Let x_j be the variable removed from Q. If x_j is already set to True, then our job is easy because nothing changes except that x_j is removed from the queue (which does not violate the second invariant since no implication with x_j on its right hand side can be unsatisfied).

Suppose instead that x_j is set to False at the beginning of the iteration. Then the algorithm first sets x_j to True. This means that all formulas with x_j on their right hand side are now satisfied, so it is okay to remove x_j from Q. Also, all formulas with x_j on their left hand side should now have their counts decremented by one, which is just what the algorithm does. Finally, if an implication became unsatisfied when x_j was set to True then it must be the case that x_j was on its left hand side and that x_j was the only variable still set to False on its left hand side. In which case, by the first invariant the implication must have had a count of 1. So the algorithm must have set the count to 0 after setting x_j to True and thus added the variable on the right hand side of the implication to Q. Thus the second invariant is still satisfied. And since this is the only way that the algorithm adds variables to Q, the third invariant is still satsified as well.

By induction, all three invariants hold on every iteration of the while loop. So at the beginning of every while loop, whatever variable is removed from Q is either already set to True or is on the right hand side of some unsatisfied implication (by the third invariant). And if the while loop ends then by the second invarant, no implications are unsatisfied.

Running Time Building the arrays in the first portion of the algorithm takes time proportional to the length of the horn formula, since for each variable appearing in each clause we do a constant amount of work. Since a variable is never changed from True to False, a variable is only processed by the while loop at most once (it may be removed from the queue more than once, but every time after the first it is ignored). And when a variable is processed, the work done is proportional to the number of implications it appears in. So the total amount of work done to process variables in the while loop is at most proportional to the length of the formula. And since each implication adds at most one item to the Queue (since its count can only become 0 once and it only has one variable on its right hand side), the number of iterations of the while loop is at most the number of implications. Finally, checking all the negative clauses takes time at most proportional to the length of the horn formula. Thus the entire algorithm is linear in the length of the formula.

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 As in the binary case, the idea is to repeatedly delete the three characters of smallest frequency, and replace them by a new character whose frequency is their sum. We wish to show that the ternary tree we get by this process is optimal. There is just one hitch — what happens if in the last step of the process we are left with only two characters. To see when this could happen, note that each step of the above process removes three characters and adds one, for a net decrease of 2. So if we start with an odd number of characters, we will eventually get down to exactly 3 characters, whereas if we start with an even number we will eventually get down to exactly 2 characters, leading to a problem. The most elegant way to resolve this problem is to add a dummy character of frequency 0 if you start with an even number of characters, thus making sure that the above procedure always results in a full ternary tree.

Algorithm

TERNARY HUFFMAN($f[1..n]$):

1. If n is even, set $f[n+1] = 0$ and set $n = n+1$
2. Let H be a priority queue of integers
3. For $i = 1$ to n: insert i into H with priority $f[i]$
4. Set $l = n$
5. While H has more than one element
6. Set $l = l+1$
7. i = deletemin(H), j = deletemin(H), k = deletemin(H)
8. Create a node numbered l with children i, j, k
9. $f[l] = f[i] + f[j] + f[k]$
10. Insert l into H with priority $f[l]$
11. Return the tree rooted at node l

After running the above algorithm, we can assign codewords in the same manner as in normal Huffman encoding– i.e. write 0, 1 and 2 on the outgoing edges of each node in the tree and encode a character by

the sequence of digits encountered along the path from the root of the tree to the leaf corresponding to that character.

Proof of Correctness First we observe that adding a character with a frequency of 0 does not change the maximum possible compression since the length of the codeword associated to that character does not affect the length of any encoded message. So it suffices to prove that the above algorithm gives the optimal encoding tree when n is odd.

Now observe that when n is odd, any optimal encoding tree must be a full ternary tree. i.e. every internal node has exactly three children. To see this, note that if a node has two children and is not at the second from the bottom level of the tree, then we can simply move a leaf from the bottom level to be its child, and improve the cost of the solution. So there can be at most one node in the tree at the second from the bottom level, which has two children. But as we argued earlier, if the total number of leaves is odd, such a node cannot exist.

Now we argue as in the binary case that the three lowest frequency leaves must be at the bottom level, since otherwise we could exchange them with whichever leaves were at the bottom level to decrease the cost. Finally, we can always move around leaves at the same level without affecting the cost, and therefore we can assume that the three lowest frequency characters are siblings on the bottom level of the tree.

At this point, the rest of the proof is nearly identical to the proof of optimality for binary Huffman encoding, but we present it here for completeness. We will proceed by induction on the number of characters. The base case, $n = 3$, is trivial (it is clearly optimal to assign each character a length 1 code, which is exactly what the algorithm above does). So assume for induction that n is an odd integer greater than 3 and ternary Huffman encoding is optimal when there are $n-2$ characters. Let $f_1, f_2, \ldots, f_n$ be a set of frequencies for n characters and let T be the tree constructed by ternary Huffman encoding from these frequencies. Let S be an optimal encoding tree for a ternary prefix free encoding of the n characters and assume that the three lowest frequency characters are siblings on the bottom level of S.

Let i, j and k be the indices of the three lowest frequency characters. Consider the set of characters that results from eliminating i, j and k and replacing them with a new character l with frequency $f_i + f_j + f_k$. Let S' and T' denote the result of deleting i, j and k from S and T respectively. S' and T' can be thought of as encoding trees for this new set of characters. Observe also that T' is exactly the tree that ternary Huffman encoding would construct given this new set of characters, so by the inductive assumption T' is optimal.

For an encoding tree R, let $d_R(x)$ denote the depth of x in R and let $Cost(R)$ denote the expected length of text encoded using R. We have:

$$\begin{aligned}
Cost(T) &= \sum_{x=1}^{n} f_x d_T(x) \\
&= \sum_{x \neq i,j,k} f_x d_T(x) + d_T(i)(f_i + f_j + f_k) \\
&= \sum_{x \neq i,j,k} f_x d_{T'}(x) + (d'_T(l) + 1)(f_i + f_j + f_k) \\
&= Cost(T') + (f_i + f_j + f_k)
\end{aligned}$$

Similar calculations show that $Cost(S) = Cost(S') + (f_i + f_j + f_k)$. Since T' is optimal, $Cost(T') \leq Cost(S')$, which implies that

$$Cost(T) = Cost(T') + (f_i + f_j + f_k) \leq Cost(S') + (f_i + f_j + f_k) = Cost(S)$$

And so T is optimal.

Running Time Adding all characters to the priority queue takes $O(n\log(n))$ time, assuming we use a standard binary heap. Then on each iteration of the while loop, removing three elements and adding one new one takes $O(\log(n))$ time. Furthermore, since the number of elements in the priority queue is reduced by two on each iteration (and started out odd), after $(n-1)/2$ iterations there is only one element left in the priority queue. So the algorithm takes at most $O(n\log(n))$ time.

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 subsqunences, 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 Suppose instead of finding the longest palindromic subsequence in the entire string, we simply want to find the longest palindromic subsequence that lies between the i^{th} and j^{th} letters of the original string. Call the length of this longest palidromic subsequence $L(i,j)$. Then there are two cases:

1. The i^{th} and j^{th} letters are the same. Then $L(i,j) = 2 + L(i+1, j-1)$.
2. The i^{th} and j^{th} letters are not the same. So we cannot include both of them. Thus $L(i,j) = \max\{L(i+1,j), L(i,j-1)\}$.

Algorithm

LONGEST PALINDROME($A[1..n]$):
1. Initialize an empty 2D array L
2. For $i = 1$ to n: set $L[i,i] = 1$ and $L[i,i-1] = 0$
3. For $k = 1$ to $n-1$:
4. For $i = 1$ to $n-k$:
5. Set $j = i+k$
6. If $A[i] = A[j]$: set $L[i,j] = 2 + L[i+1, j-1]$
7. Else: set $L[i,j] = \max\{L[i+1,j], L[i,j-1]\}$
8. Return $L[1,n]$

Proof of Correctness We need to prove that the base case and the recursive formula are correct and that the subproblems are arranged in the right order– i.e. that when solving one subproblem we have already solved the subproblems it relies on.

The base cases are straightforward since if $i = j$ then there is only one possible subsequence and it is palindromic since it has one character. If $j = i-1$ then there are no subsequences so there are no palindromic subsequences.

Now we need to verify that the recursive formula for $L(i,j)$ is correct. Suppose that $i < j$. Consider the longest palindromic subsequence, S, that lies inbetween characters i and j (inclusive). Suppose $A[i] = A[j]$. If S does not contain $A[i]$ or $A[j]$ then it cannot have maximal length since we could append $A[i]$ and $A[j]$ to the end of S to get a longer palindromic subsequence. If it contains only $A[i]$ then the last character in

S must also match $A[i]$ so we can remove the last character and append $A[j]$ without changing the length. Similar reasoning applies if it contains $A[j]$ but not $A[i]$. Thus we can assume S includes both $A[i]$ and $A[j]$. Furthermore, if we remove both these characters then we should get a longest palindromic subsequence between $i+1$ and $j-1$ since otherwise we could replace the rest of S with such a longest palindromic subsequence. So $L(i,j) = 2 + L(i+1, j-1)$.

Now suppose that $A[i] \neq A[j]$. Then S cannot include both $A[i]$ and $A[j]$ so it must either lie between i and $j-1$ or between $i+1$ and j. Thus $L(i,j) = \max\{L(i,j-1), L(i+1,j)\}$.

Finally, note that for any i and j, $L(i,j)$ depends only on subproblems for which the distance between i and j is smaller. And since the algorithm solves subproblems in order of the distance between i and j, it will always have already solved the subproblems that the current subproblem depends on.

Running Time The algorithm simply fills in some portion of an $n \times n$ array, and takes a constant amount of time to fill in each entry. So the entire algorithm runs in $O(n^2)$ time.