CS 170 HW 6

Due on 2018-10-07, at 9:59 pm

1 (★) Study Group

List the names and SIDs of the members in your study group.

2 (★★★) Horn Formulas

Describe an algorithm which finds a satisfying assignment for a Horn formula, or reports that none exists, in time linear in the size of the formula. Prove that your algorithm is correct and runs in linear time.

Solution:

Main Idea

We first build a graph of the implications of the Horn formula. We have a node for each implication, and a node for each variable. If a variable x appears on the LHS of the implication I, then there is a directed edge $(x, I) \in E$. If x appears on the RHS of I, there is a directed edge (I, x). When we set a variable x to true, we delete it from the graph. If this causes an implication I to have indegree 0, we delete it and its consequent from the graph. Once there are no implications with indegree 0, we check whether the negative clauses are satisfied by setting the remaining variables to false. Initially we have to search the graph for implications with indegree 0 and add them to a queue. Then we add every implication whose indegree is newly 0 after deleting some vertex to that queue.

Proof of Correctness The truth assignment corresponding to a graph G is given by setting every variable to false if the corresponding vertex is in G, and true otherwise. When the algorithm terminates, every implication in G has nonzero indegree. This means that for every implication in G, at least one of its antecedents is false, and so the consequent can be set to false. We only delete a vertex corresponding to a variable if it *must* be set to true. The Horn formula is then satisifiable if and only if the assignment corresponding to G satisfies the negative clauses.

Running Time If we use the appropriate data structure for the graph, building it takes time linear in the size of the formula. Every step deletes some vertex from the graph, and so we can only do linearly many. The total cost of deleting all the vertices and edges is linear in the size of the graph, and so the total running time is linear.

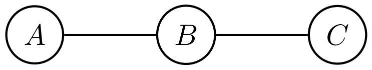
3 (★★) Graph Game

Given an undirected, unweighted graph G, with each node v having a value $\ell(v) \geq 0$, consider the following game.

1. All nodes are initially *unmarked* and your score is 0.
2. Choose an unmarked node u. Let $M(u)$ be the *marked* neighbours of u. Add $\sum_{v \in M(u)} \ell(v)$ to your score. Then mark u.

3. Repeat the last step for as many turns as you like, or until all the nodes are marked.

For instance, suppose we had the graph:



with $\ell(A) = 3$, $\ell(B) = 2$, $\ell(C) = 3$. Then, an optimal strategy is to mark A then C then B giving you a score of $0 + 0 + 6$. We can check that no other order will give us a better score.

(a) Is it ever better to leave a node unmarked? Briefly justify your answer.

(b) Give a greedy algorithm to find the order which gives the best score. Describe your algorithm, prove its optimality, and analyse its running time.

(c) Now suppose that $\ell(v)$ can be negative. Give an example where your algorithm fails.

(d) Your friend suggests the following modified algorithm: delete all v with $\ell(v) < 0$, then run your greedy algorithm on the resulting graph. Give an example where this algorithm fails.

Solution:

(a) Marking a node can only ever increase your score, since all values are positive.

(b) Sorting the nodes by value largest-first (breaking ties arbitrarily), and taking them in that order will lead to an optimal solution. We show this by an exchange argument. Suppose that we have an ordering $v_1, \ldots, v_n$ which is not sorted by decreasing value, and let i be such that $\ell(v_i) < \ell(v_{i+1})$. Note first that if we swap the order of these nodes, only the scores that we obtain in steps i and $i+1$ could change, since in all other steps the set of marked nodes is unaffected. There are two cases.

   (a) Case 1: v_i and v_{i+1} do not have an edge between them. Then the score when marking v_i is not affected by whether v_{i+1} is marked, and vice versa.

   (b) Case 2: π_i and π_{i+1} do have an edge between them. Then if we swap them, v_{i+1}'s score decreases by $\ell(v_i)$, and v_i's score increases by $\ell(v_{i+1})$. Since $\ell(v_{i+1}) > \ell(v_i)$, the total score increases.

   Thus, by the exchange argument, sorting the nodes will give us an optimal solution. The running time of this algorithm is $O(|V| \log |V|)$, since we just sort the vertices.

(c) We can take the example graph and set $\ell(A) = 1$, $\ell(B) = -1$, $\ell(C) = -2$. Then the greedy algorithm gives A, B, C with value 0. The optimum is A, B with value 1.

(d) Again we take the example graph and set $\ell(A) = \ell(C) = 3$, $\ell(B) = -1$. The modified algorithm gives A, C which has value 0. The optimum is A, C, B with value 6.

4 (★★) Tree Perfect Matching

A *perfect matching* in an undirected graph $G = (V, E)$ is a set of edges $E' \subseteq E$ such that for every vertex $v \in V$, there is exactly one edge in E' which is incident to v.

Give an algorithm which finds a perfect matching *in a tree*, or reports that no such matching exists. Describe your algorithm, prove that it is correct and analyse its running time.

Solution: Let v be a leaf vertex in T. Since v has only one incident edge $e = (u, v)$, e must be in every perfect matching in T. Add e to E' and remove u, v from T. Repeat until there are no edges remaining. If there are no vertices remaining, return E', otherwise output 'no matching'.

One point to note: removing u, v might cause the graph to become disconnected. In that case we just run the algorithm on each connected component.

The running time of this algorithm is $O(|V|)$. We visit every node at most twice: once when we search into its subtree, and once when we remove it from T after it's been matched.

5 (★★) Steel Beams

You're a construction engineer tasked with building a new transit center for a large city. The design for the center calls for a T-foot-long steel beam for integer $T > 0$. Your supplier can provide you with an *unlimited* number of steel beams of integer lengths $0 < c_1 < \ldots < c_k$ feet. You can weld as many beams as you like together; if you weld together an a-foot beam and a b-foot beam you'll have an $(a + b)$-foot beam. Unfortunately, every weld increases the chance that the beam might break, so you want as few as possible.

Your task is to design an algorithm which outputs how many beams of each length you need to obtain a T-foot beam with the minimum number of welds, or 'not possible' if there's no way to make a T-foot beam from the lengths you're given. (If there are multiple optimal solutions, your algorithm may return any of them.)

(a) Consider the following greedy strategy. Start with zero beams of each type. While the total length of all the beams you have is less than T, add the longest beam you can without the total length going over T.

  (i) Suppose that we have 1-foot, 2-foot and 5-foot beams. Show that the greedy strategy always finds the optimum.

  (ii) Find a (short) list of beam sizes $c_1, \ldots, c_k$ and target T such that the greedy strategy fails to find the optimum. Briefly justify your choice.

(b) Give a dynamic programming algorithm which always finds the optimum. Describe your algorithm, including a clear statement of your recurrence, show that it is correct and prove its running time. How much space does your algorithm use?

Solution: Formal statement of problem: Given a list of integers $C = (c_1, \ldots, c_k)$ with $0 < c_1 < \ldots < c_k$ and a target $T > 0$, the algorithm should output **nonnegative** integers $(a_1, \ldots, a_k)$ such that $\sum_{i=1}^{k} a_i c_i = T$ where $\sum_{i=1}^{k} a_i$ is as small as possible, or return 'not possible' if no such integers exist.

(a) (i) Let a_1, a_2, a_3 be some optimum solution. We know that $a_1 < 2$ since if $a_1 \geq 2$ we can improve the solution by taking $a_1 - 2, a_2 + 1, a_3$. If $a_1 = 1$ then $a_2 < 2$ because otherwise we could improve by taking $0, a_2 - 2, a_3 + 1$. So the possible values of the optimum are $0, 1, j$, $0, 2, j$, $1, 1, j$ for some $j \geq 0$. In each case the greedy algorithm would give the same answer.

(ii) $C = (4, 5)$, $T = 8$ is one possibility.

(b) Description: We create a dynamic programming algorithm where, for each $n \leq A$, we will find the minimum integer combination that sums to n. The recurrence is $f(n) = \min_{i=1}^{k} f(n - c_i) + 1$, with $f(0) = 0$. Maintain pointers so we can trace back the solution. We initialize the array to ∞. If $f(T) = \infty$ then return 'not possible', otherwise return the solution.

Proof of correctness: By induction. Base case is easy, so fix $n > 0$. Suppose that $f(n')$ is optimal for all $n < n'$. Firstly note that if $a'_1, \ldots, a'_k$ is a minimum integer combination summing to $n - c_i$, then $a'_1, \ldots, a'_i + 1, \ldots, a'_k$ is an integer combination summing to n (not necessarily minimum). Let $a_1, \ldots, a_k$ be a minimum integer combination summing to n. Then for every i with $a_i > 0$, $a_1, \ldots, a_i - 1, \ldots, a_k$ is a minimum integer combination summing to $n - c_i$ (otherwise $a_1, \ldots, a_k$ wouldn't be optimal). We know that some $a_i > 0$ (we don't know which), so we take the minimum over all i.

Running time: We compute T subproblems, each one being a minimum of k values, so running time is $O(Tk)$. The space requirement is $O(T)$.

6 (★) Longest Increasing Subsequence

Given an array A of n integers, here is a linear time algorithm to find the length of the longest **strictly** increasing subsequence, where $M[j]$ represents the length of the longest increasing subsequence of the first j integers of A.

$$M[1] = 1$$
$$M[i] = \begin{cases} M[i-1] & \text{if } A[i] \leq A[i-1] \\ M[i-1] + 1 & \text{otherwise} \end{cases}$$

Verify that this algorithm takes linear time. Is the algorithm correct? Prove it is or give a counter-example and explain.

Solution: The algorithm starts from the left of A and works its way up, comparing each integer with the one before it and the one after it. Therefore, each integer is only touched twice, resulting in a linear run-time.

The algorithm is incorrect.

A counterexample is $A = [3, 4, 1, 2]$

The longest increasing subsequence length is 2, but the algorithm just counts (one plus) the number of indices i such that $A[i] > A[i-1]$, and then it returns 3 (as if $A[1], A[2], A[4]$ were an increasing subsequence).

7 (★★★★) Propositional Parentheses

You are given a propositional logic formula using only $\wedge$, $\vee$, T, and F that does not have parentheses. You want to find out how many different ways there are to *correctly parenthesize* the formula so that the resulting formula evaluates to true.

A formula A is correctly parenthesized if $A = T$, $A = F$, or $A = (B \wedge C)$ or $A = (B \vee C)$ where B, C are correctly parenthesized formulas. For example, the formula $T \vee F \vee T \wedge F$ can be correctly parenthesized in 5 ways:

$$(T \vee (F \vee (T \wedge F))) \quad (T \vee ((F \vee T) \wedge F)) \quad ((T \vee F) \vee (T \wedge F))$$
$$(((T \vee F) \vee T) \wedge F) \quad ((T \vee (F \vee T)) \wedge F)$$

of which 3 evaluate to true: $((T\vee F)\vee(T\wedge F))$, $(T\vee((F\vee T)\wedge F))$, and $(T\vee(F\vee(T\wedge F)))$.

(a) Give a dynamic programming algorithm to solve this problem. Describe your algorithm, including a clear statement of your recurrence, show that it is correct, and prove its running time.

(b) Briefly explain how you could use your algorithm to find the probability that, under a *uniformly randomly chosen* correct parenthesization, the formula evaluates to true.

Solution:

(a) Choosing parentheses amounts to deciding how to represent the formula as a binary tree. Let $A = x_1 o_1 x_2 o_2 \dots x_{n-1} o_{n-1} x_n$, where $x_i \in \{T, F\}$ and $o_i \in \{\wedge, \vee\}$. We first choose an operator o_r to be the root. This splits A into subformulas $x_1 o_1 \dots x_{r-1} o_{r-1} x_r$ and $x_{r+1} o_{r+1} \dots x_{n-1} o_{n-1} x_n$. We then recurse on each side, choosing operators to be the roots of the left and right subtrees, until we reach a formula which is just T or F. Every correct parenthesization corresponds to exactly one such binary tree.

Given such a tree it is clear how to evaluate it: take any node v labelled with $o \in \{\wedge, \vee\}$ with children labelled $x_l, x_r \in \{T, F\}$ and replace v's label with the value of $x_l \, o \, x_r$. Keep doing this until the root is labelled with T or F. This is the same as evaluating the corresponding parenthesized formula.

The subproblems in our dynamic program will be defined by pairs (i, j) with $i \leq j$. Let $A_{i,j} = x_i o_i \dots x_j o_{j-1} x_j$. Let $t(i, j)$ be the number of ways to parenthesize $A_{i,j}$ which evaluate to T; let $f(i, j)$ be the number of ways to parenthesize $A_{i,j}$ which evaluate to

F. We obtain the following recurrences:

$$t(i,i) = \begin{cases} 1 & \text{if } x_i = T \\ 0 & \text{if } x_i = F \end{cases}$$

$$t(i,j) = \sum_{\substack{i \le k < j \\ o_k = \vee}} t(i,k) \cdot t(k+1,j) + f(i,k) \cdot t(k+1,j) + t(i,k) \cdot f(k+1,j) + \sum_{\substack{i \le k < j \\ o_k = \wedge}} t(i,k) \cdot t(k+1,j)$$

$$f(i,i) = \begin{cases} 0 & \text{if } x_i = T \\ 1 & \text{if } x_i = F \end{cases}$$

$$f(i,j) = \sum_{\substack{i \le k < j \\ o_k = \wedge}} f(i,k) \cdot f(k+1,j) + f(i,k) \cdot t(k+1,j) + t(i,k) \cdot f(k+1,j) + \sum_{\substack{i \le k < j \\ o_k = \vee}} f(i,k) \cdot f(k+1,j)$$

The proof is by induction on the size of an interval. Every interval of size 1 is correct since there's only one way to parenthesize T or F (i.e. by not giving them parentheses). Now let $m := j - i + 1$ and suppose that t and f are correct for every interval of size less than m. Every correct parenthesization is given by choosing a root o_k and then choosing some correct parenthesization of the left and right subformulas. We'll look at the t recurrence; f is similar. If $o_k = \vee$, then such a parenthesization evaluates to T if and only if at least one of its children does. If $o_k = \wedge$, then we need both children to evaluate to T. Since we can choose the left and right parenthesizations independently, we obtain the expression above.

Our algorithm must compute the $t(i,j)$ and $f(i,j)$ in order of interval size: we start with all intervals of size 1, then size 2, etc. The running time is $O(n^3)$ because there are $O(n^2)$ intervals and it takes $O(n)$ time to compute $t(i,j)$ and $f(i,j)$. The result is found in $T(1,n)$.

(b) Each correct parenthesization either evaluates to true or false. So the total number of correct parenthesizations of the given formula is $t(1,n) + f(1,n)$. Thus the probability that a randomly drawn correct parenthesization evaluates to true is $t(1,n)/(t(1,n) + f(1,n))$.