# CS 170 DIS 05

**Released on 2018-10-01**

## 1 Horn Formula Practice

Find the variable assignment that solves the following horn formulas:

1. $(w \wedge y \wedge z) \Rightarrow x, (x \wedge z) \Rightarrow w, x \Rightarrow y, \Rightarrow x, (x \wedge y) \Rightarrow w, (\bar{w} \vee \bar{x}, \vee \bar{y}), (\bar{z})$

2. $(x \wedge z) \Rightarrow y, z \Rightarrow w, (y \wedge z) \Rightarrow x, \Rightarrow z, (\bar{z} \vee \bar{x}), (\bar{w} \vee \bar{y} \vee \bar{z})$

**Solution:**

1. - Set everything to false initially
   - $x$ must be true since we have the statement $\Rightarrow x$
   - $y$ must be true since $x \Rightarrow y$
   - $w$ must be true since $(x \wedge y) \Rightarrow w)$
   - Not all negative clauses are satisfied at this point, so there is no satisfying assignment.
2. - $z$ must be true since we have the statement $\Rightarrow z$.
   - $w$ must be true since $z \Rightarrow w$
   - $x$ and $y$ need not be changed, as all our implications are satisfied.
   - All negative clauses are now satisfied, so we've found our satisfying assignment.

## 2 Huffman Proofs

1. Prove that in the Huffman coding scheme, if some character occurs with frequency more than $\frac{2}{5}$, then there is guaranteed to be a codeword of length 1. Also prove that if all characters occur with frequency less than $\frac{1}{3}$, then there is guaranteed to be no codeword of length 1.

2. Under a Huffman encoding of $n$ symbols with frequencies $f_1, f_2, \ldots, f_n$, what is the longest a codeword could possibly be? Give an example set of frequencies that would produce this case, and argue that it is the longest possible.

**Solution:**

1. Suppose all codewords have length at least 2 – the tree must have at least 2 levels. Let the weight of a node be the sum of the frequencies of all leaves that can be reached from that node. Suppose the weights of the level-2 nodes are (from left to right for a tree rooted on top) $a$, $b$, $c$, and $d$. Without loss of generality, assume A and B are joined first, then C and D. So, $a, b \leq c, d \leq a + b$.

   If $a$ or $b$ is greater than $\frac{2}{5}$, then both $c$ and $d$ are greater than $\frac{2}{5}$, so $a+b+c+d > \frac{6}{5} > 1$ (impossible). Now suppose $c$ is greater than $\frac{2}{5}$ (similar idea if $d$ is greater than $\frac{2}{5}$). Then $a + b > \frac{2}{5}$, so either $a > \frac{1}{5}$ or $b > \frac{1}{5}$ which implies $d > \frac{1}{5}$. We obtain $a + b + c + d > 1$ (impossible).

   For the second part suppose there is a codeword of length 1. We have 3 cases. Either the tree will consist of 1 single level-1 leaf node, 2 level-1 leaf nodes, or 1 level-1 leaf node, and 2 level-2 nodes with an arbitrary number of leaves below them in the tree. We will prove the contrapositive of the original statement, that is, that if there is a codeword of length 1, there must be a node with frequency greater than $\frac{1}{3}$

   In the first case, our leaf must have frequency 1, so we've immediately found a leaf of frequency more than $\frac{1}{3}$. If the tree has two nodes, one of them has frequency at least $\frac{1}{2}$, so condition is again satsified. In the last case, the tree has one level-1 leaf (weight $a$), and two level-2 nodes (weights $c$ and $d$). We have: $b, c \leq a$. So $1 = a + b + c \leq 3a$, or $a \geq \frac{1}{3}$. Again, our condition has been satisfied.

2. The longest codeword can be of length $n - 1$. An encoding of $n$ symbols with $n - 2$ of them having probabilities $1/2, 1/4, \ldots, 1/2^{n-2}$ and two of them having probability $1/2^{n-1}$ achieves this value. No codeword can ever by longer than length $n - 1$. To see why, we consider a prefix tree of the code. If a codeword has length $n$ or greater, then the prefix tree would have height $n$ or greater, so it would have at least $n + 1$ leaves. Our alphabet is of size $n$, so the prefix tree has exactly $n$ leaves.

## 3 Finding Counterexamples

In this problem, we give example greedy algorithms for various problems, and your goal is to find an example where they are not optimal.

(a) In the travelling salesman problem, we have a weighted undirected graph $G(V, E)$ with all possible edges. Our goal is to find the cycle that visits all the vertices exactly once with minimum length.

One greedy algorithm is: Build the cycle starting from an arbitrary start point $s$, and initialize the set of visited vertices to just $s$. At each step, if we are currently at vertex $u$ and our cycle has not visited all the vertices yet, add the shortest edge from $u$ to an unvisited vertex $v$ to the cycle, and then move to $v$ and mark $v$ as visited. Otherwise, add an edge from the current vertex to $s$ the cycle, and return the now complete cycle.

(b) In the maximum matching problem, we have an undirected graph $G(V, E)$ and our goal is to find the largest matching $E'$ in $E$, i.e. the largest subset $E'$ of $E$ such that no two edges in $E'$ share an endpoint.

One greedy algorithm is: While there is an edge $e = (u, v)$ in $E$ such that neither $u$ or $v$ is already an endpoint of an edge in $E'$, add any such edge to $E'$. (Can you prove that this algorithm still finds a solution whose size is at least half the size of the best solution?)

**Solution:**
Note: For each part, there are many counterexamples.

(a) One counterexample is to have a four vertex graph with vertices $a, b, c, d$, where the edges $(a, b), (b, c), (c, d)$ cost 1, the edge $(a, d)$ costs 100, and all other edges cost 2. If the greedy algorithm starts at vertex $a$, it will add the edges $(a, b), (b, c), (c, d)$ to the cycle, and then be forced to add the very expensive edge $(a, d)$ at the end to find a cycle of cost 103. The optimal cycle is $(a, b), (b, d), (d, c), (c, a)$ which costs 6. The key idea here is that by using a path of low-weight edges, we forced the algorithm into a position where it had to pick a high-weight edge to complete its solution.

(b) The simplest example is a path graph with three edges. The greedy algorithm will pick the middle edge, the optimal solution is to pick the two outer edges.

To show this algorithm always finds a matching at least half the size of the best solution, let the size of the solution found by the algorithm be $m$. Any edge in the best solution shares an endpoint with one of the edges in the algorithm's solution (otherwise, the greedy algorithm could have added it). None of the edges in the best solution can share an endpoint, and the $m$ edges in the algorithm's solution have $2m$ endpoints, so the best solution must have at most $2m$ edges.

## 4 Worst-case Instances for Greedy Set-Cover

Recall the set cover problem:

Input: A set of elements $B$ and sets $S_1, \ldots, S_m \subseteq B$.

Output: A selection of the $S_i$ whose union is $B$ (i.e. that contain every element of $B$).

Cost: Number of sets picked.

The natural strategy to solve this problem is a greedy approach: At every step, pick the set that covers the most uncovered elements of $B$. In the book, we proved that this greedy strategy over-estimates the optimal number of sets by a factor of at most $O(\log n)$, where $n = |B|$. In this problem we will prove that this bound is tight.

Show that for any integer $n$ that is a power of 2, there is an instance of the set cover problem (i.e. a collection of sets $S_1, \ldots, S_m$) with the following properties:

i. There are $n$ elements in the base set $B$.

ii. The optimal cover uses just two sets.

iii. The greedy algorithm picks at least $O(\log n)$ sets.

**Solution:** Consider the base set $U = \{1, 2, 3, \ldots, 2^k\}$ for some $k \geq 2$. Let $T_1 = \{1, 3, 5, \ldots, 2^k - 1\}$ and $T_2 = \{2, 4, 6, \ldots, 2^k\}$. These two sets comprise an optimal cover. We add sets $S_1, \ldots, S_{k-1}$ by defining $l_i = 2 + \sum_{j=1}^{i} 2^{k-j}$ (take $l_0 = 0$) and letting $S_i = \{l_{i-1} + 1, \ldots, l_i\}$. We think of $S_i$ as covering a tiny bit more than a $1/2^i$-fraction of the universe, and in particular $S_1$ is larger than both $T_1$ and $T_2$.

Since $S_1$ contains $2^{k-1}+2$ elements, it will be picked first. After the algorithm has picked $\{S_1, S_2, \ldots, S_i\}$, each of $T_1$ and $T_2$ covers $2^{k-i-1}-1$ new elements while $S_{i+1}$ covers $2^{k-i-1}$ new elements. Hence, the algorithm picks the cover $S_1, \ldots, S_{k-1}$ containing $k-1 = \log n - 1$ sets.

An example of the above algorithm is with, say, $n = 64$. Let $T_1 = \{1, 3 \ldots, 63\}$, and $T_2 = \{2, 4, \ldots, 64\}$. Let picking $T_1$ and $T_2$ be the optimal answer. Now let $S_1 = \{1, 2, \ldots, 32, 33, 34\}$, or 2 elements more than 64/2. Let $S_2$ contain the next $64/4 = 16$ elements, let $S_3$ contain the next $64/8 = 8$ elements, $S_4$ contain the next $64/16 = 4$ elements, and $S_5$ contain the remaining 2 elements. This works because $64 = 32 + 16 + 8 + 4 + 2 + 2$. We are just regrouping the terms as follows $64 = (32 + 2) + 16 + 8 + 4 + 2$ and creating sets out of them.

Our greedy algorithm will pick $S_1, S_2, S_3, S_4, S_5$ instead of $T_1, T_2$. Our greedy algorithm ended up picking $\log(k) - 1$ sets, since $k = 6$ for $n = 64$. The first two paragraphs of the solutions explain how this is generalized for all $k$.

## 5 Planting Trees

This problem will guide you through the process of writing a dynamic programming algorithm.

You have a garden and want to plant some apple trees in your garden, so that they produce as many apples as possible. There are $n$ adjacent spots numbered 1 to $n$ in your garden where you can place a tree. Based on the quality of the soil in each spot, you know that if you plant a tree in the $i$th spot, it will produce exactly $x_i$ apples. However, each tree needs space to grow, so if you place a tree in the $i$th spot, you can't place a tree in spots $i-1$ or $i+1$. What is the maximum number of apples you can produce in your garden?

(a) Give an example of an input for which:

- Starting from either the first or second spot and then picking every other spot (e.g. either planting the trees in spots $1, 3, 5 \ldots$ or in spots $2, 4, 6 \ldots$) does not produce an optimal solution.
- The following algorithm does not produce an optimal solution: While it is possible to plant another tree, plant a tree in the spot where we are allowed to plant a tree with the largest $x_i$ value.

(b) To solve this problem, we'll thinking about solving the following, more general problem: "What is the maximum number of apples that can be produced using only spots 1 to $i$?". Let $f(i)$ denote the answer to this question for any $i$. Define $f(0) = 0$, as when we have no spots, we can't plant any trees. What is $f(1)$? What is $f(2)$?

(c) Suppose you know that the best way to plant trees using only spots 1 to $i$ does not place a tree in spot $i$. In this case, express $f(i)$ in terms of $x_i$ and $f(j)$ for $j < i$. (Hint: What spots are we left with? What is the best way to plant trees in these spots?)

(d) Suppose you know that the best way to plant trees using only spots 1 to $i$ places a tree in spot $i$. In this case, express $f(i)$ in terms of $x_i$ and $f(j)$ for $j < i$.

(e) Describe a linear-time algorithm to compute the maximum number of apples you can produce. (Hint: Compute $f(i)$ for every $i$. You should be able to combine your results from the previous two parts to perform each computation in $O(1)$ time).

**Solution:**

(a) For the first algorithm, a simple input where this fails is $[2, 1, 1, 2]$. Here, the best solution is to plant trees in spots 1 and 4. For the second algorithm, a simple input where this fails is $[2, 3, 2]$. Here, the greedy algorithm plants a tree in spot 2, but the best solution is to plant a tree in spots 1 and 3.

(b) $f(1) = x_1$, $f(2) = \max\{x_1, x_2\}$

(c) If we don't plant a tree in spot $i$, then the best way to plant trees in spots 1 to $i$ is the same as the best way to plant trees in spots 1 to $i - 1$. Then, $f(i) = f(i - 1)$.

(d) If we plant a tree in spot $i$, then we get $x_i$ apples from it. However, we cannot plant a tree in spot $i - 1$, so we are only allowed to place trees in spots 1 to $i - 2$. In turn, in this case we can pick the best way to plant trees in spots 1 to $i - 2$ and then add a tree at $i$ to this solution to get the best way to plant trees in spots 1 to $i$. So we get $f(i) = f(i - 2) + x_i$.

(e) Initialize a length $n$ array, where the $i$th entry of the array will store $f(i)$. Fill in $f(1)$, and then use the formula $f(i) = \max\{f(i - 1), x_i + f(i - 2)\}$ to fill out the rest of the table in order. Then, return $f(n)$ from the table.