Horn SAT

Implications:

And of positive literals imply **one** positive literal.

$$x \wedge y \rightarrow z$$

Negative clauses:

Or of negative literals.

$$\overline{u} \vee \overline{v}$$

Taint Example:

$$\implies A,\ A \implies B,\ \overline{B}.$$

Is this satisfiable?

Horn Sat: another view.

$$\begin{aligned} x_1 \wedge x_2 &\implies x_4 \\ x_3 &\implies x_2 \\ x_1 &\implies x_3 \\ x_5 \wedge x_1 &\implies x_3 \\ x_2 \wedge x_6 &\implies x_5 \\ &\implies x_1 \end{aligned}$$

Problem: Find consistent assignment with fewest "True" variables.

Greedy algorithm: Only set variables to true if you have to.

Example:
x_1 must be true so x_3 must be true
so x_2 must be true so x_4 must be true
Solution: $\{x_1, x_2, x_3, x_4\}$ are True

Could also set x_5 to true, or both x_5 and x_6 to true...but don't!

Same as horn sat!

Why same as HornSAT?

Horn SAT had negative clauses.

No negative clauses for above algorithm.

Algorithm: Set a variable true ..if you have to!

Property: any variable set to true must be true in *any* satisfying assignment.

By induction. First k set to true... must be!
The $k+1$ set variable set to true
is set to true to satisfy a clause
so it must be true.

Horn has negative clauses.

Negative clauses only problem for true variables.

Any variable that is true must be true.

So if a negative clause is false, it must be.

Efficient implementation

$$
\begin{aligned}
x_1 \wedge x_2 &\implies x_4 \\
x_3 &\implies x_2 \\
x_1 &\implies x_3 \\
x_5 \wedge x_1 &\implies x_3 \\
x_2 \wedge x_6 &\implies x_5 \\
&\implies x_1
\end{aligned}
$$

For each clause: keep count of true antecedents:

When all antecedents true, than make consequent true.

Data Structure:

Connect variable to clauses with var as antecendent.

When variable is set to true see if connected clauses are invoked.

Any SAT formula?

$$x_1 \implies x_2 \vee x_3.$$

x_1 being true may mean nothing for x_3?

don't have to set it to true.

No known polynomial time algorithm.

...no polynomial time algorithm unless NP = P ...

More about this... later in the course.

Set Cover.

Input:
Items: $B = \{1, \ldots, n\}$
Sets: $S_1, \ldots, S_m \subseteq B$

Find fewest sets that cover B (so that union is B)

Items: City Blocks.
Sets: Possible cellphone tower location.
Each cell phone tower location covers some subset of blocks.

Items: Customers.
Sets: Walmart locations covers subset of customers.

Items: Job responsibilities (ruby, perl, python, web, unix,...).
Sets: People with job capabilities.

Items: factory needs (touch screens, chips).
Sets: suppliers.

Greedy Algorithm

Choose set S_i that has largest number of elements.
    Remove elements in S_i from all sets.
    Repeat.

Number of sets is number of iterations.
How many iterations?

Property: Set cover of size k (best solution)
$\implies$ there exists a set that contains $\frac{1}{k}$ of remaining elements.

Analysis:
n_t elements remain at time t (after using t sets.)

In iteration t, cover $\frac{1}{k}n_t$ remaining elements.

$$n_{t+1} \leq n_t - \frac{1}{k}n_t = (1 - \frac{1}{k})n_t.$$

$$n_t \leq (1 - \frac{1}{k})^t n_0$$

When do we stop?

Bound iterations.

When do we stop?

When $n_t < 1$?

Recall: $n_t \leq (1 - \frac{1}{k})^t n_0$

For what t must $n_t < 1$?

(A) $t = \log n$

(B) $t = k$

(C) $t = k \ln n$.

(C).

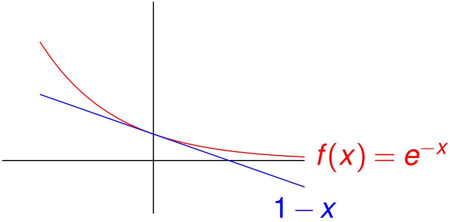
Plug in $t = k \ln n + 1$ and clearly $n_t < 1$. (More in a moment.)

Bound iterations (really)

$n_t \leq (1 - \frac{1}{k})^t n_0$

When must $n_t < 1$?

Remember: $(1 - x) \leq e^{-x}$



So, $n_t \leq (1 - \frac{1}{k})^t n < (e^{-\frac{1}{k}})^t n \leq (e^{-\frac{t}{k}})n$.

For $t = k \ln n$, $n_t < (e^{-\ln n})n = (\frac{1}{n})n = 1$.
No elements are uncovered at this time!

So $t \leq k \ln n + 1$. Number of sets for greedy is at most $k \ln n + 1$!

Within $\ln n$ factor (almost) of the best possible!

Can we do better?

We did not find optimal solution!

Is there a better analysis?

No. Problem 5.33!

Is there a better algorithm?

“Probably” not!

Again, only if P=NP.

More later in the course.

Recipe for Dynamic Programming

- Define a set of problems, such that
  - base case - easy to solve
  - final case - matches (closely) the final problem we want to solve.
- Write it as a recursion: Solve bigger problem in terms of the smaller problems. (Should be a DAG on the problem instances!)
- Compute the problems on the DAG in the linearized order!

Longest Increasing Subsequence

Given sequence of numbers: $a_1, a_2, \ldots, a_n$.

Find longest increasing sequence of numbers.

7 3 5 6 1 2 9 4 8

Greedy??

(A) Choose first number, delete higher, continue.

(B) Choose lowest number, continue with rest of sequence.

(A) seems bad.
Try method (B): 1 2 4 8

7 3 5 6 8 2 9 4 1

Not good!

What to do?

Dynamic Programming Solution.

$L(i)$ is length of longest increasing subsequence ending at position i.

Do I know $L(1)$? Is $L(n)$ good enough for the answer?($\max_j L(j)$)

Recursion

$$L(j) = \max_{j<i \,\wedge\, a[j]<a[i]} \{L(j)+1\}$$

Think of the DAG? **For** $i = 1, 2, \ldots, n$

$L(i) = 1$

For $j = 1, \ldots, i-1$

if $a[j] < a[i]$

$L(i) = \max(L(i), L(j)+1)$

$O(n^2)$ time $O(n)$ space. *Find* longest subsequence?(maintain pointers)

For $i = 1, 2, \ldots, n$

$L(i) = 1$, prev$(i) = i$

For $j = 1, \ldots, i-1$

if $a[j] < a[i]$

if $L(j)+1 > L(i)$

$L(i) = L(j)+1$; prev$(i) = j$

Chase prev(j) pointers backwards to construct path! Similar to finding sp tree.

Dynamic Programming

- Define a set of problems, such that
  - base case - easy to solve
  - final case - matches (closely) the final problem we want to solve.
- Write it as a recursion: Solve bigger problem in terms of the smaller problems. (Should be a DAG on the problem instances!)
- Compute the problems on the DAG in the linearized order!

Longest increasing subsequence.

Subproblems:
$L(i)$ - longest increasing subsequence ending at i.
Only need $L(j)$ for $j < i$ to find $L(i)$.

"DAG of problems to solve."

Dynamic Programming and Recursion.

$L(i)$ - longest increasing subsequence ending at i.
Only need $L(j)$ for $j < i$ to find $L(i)$.

Recursion $\equiv$ dynamic programming?
sub problem solutions...built from smaller subproblems.

Recursive instead of iterative?

def L(i):
  val = 1
  for $j = 1, \ldots, n$:
    if $a[j] < a[i]$:
      if $L(j) + 1 > val$:
        val = L(j) + 1

Enumerates all paths in "DAG". Exponential time!!

Memoization.
Answer $L(i)$ same each time, so remember, return.

Only n different arguments provided to $L(\cdot)$.
Each call takes $O(n)$ time the *first* time.

Same as "iterative".