Lecture 10

CS 170

Sanjam Garg.

Horn Formulas

$x_1, x_2, \ldots x_n$ are n boolean variables (can be TRUE/FALSE)

Input: $C_1 \ldots C_m$ s.t. $\forall i \in [m]$ C_i is either $(\bar{x}_1 \vee \bar{x}_2 \ldots)$] "pure negative"
or $(\bar{x}_1 \vee \bar{x}_2 \ldots \vee x)$ "implication"

$\hookrightarrow (x_1 \wedge x_2 \wedge \ldots) \Rightarrow x$

Goal: Is $F = C_1 \wedge C_2 \ldots C_m$ satisfiable?

Special case $(\Rightarrow x) \rightarrow (x)$

Example:

$(w \wedge y \wedge z) \Rightarrow x$

$(x \wedge z) \Rightarrow w$

$x \Rightarrow y$

$\Rightarrow x$

$(x \wedge y) \Rightarrow w$

$(\bar{w} \vee \bar{x} \vee \bar{y})$

$(\bar{z})$

Greedy Approach

- Set all variables to FALSE
- Set a variable to TRUE only if absolutely necessary.

Horn(F)

> Set $x_1 \dots x_n$ to FALSE
> **While** $\exists$ an unsatisfied implication clause C
>   set the right-hand variable in C
>   to TRUE
>
> **if** all pure negative clauses are satisfied
>   then return the assignment $x_1 \dots x_n$
> **else** return "F is unsatisfiable."

Run-time: $O(\underbrace{|F|}_{\text{size of the formula}} \times n)$ $\nearrow m$

Local to Global

Lemma: If Horn(F) sets a variable $x = \text{TRUE}$ then this is TRUE in all satisfying assignments of F.

Proof: $k = 1 \rightarrow \quad \Rightarrow x \longrightarrow$ then any satisfying assignment must set x to TRUE

$k \rightarrow k+1 \quad x_{i_1}, x_{i_2} \dots x_{i_k} \rightarrow$ set to TRUE

$x_{i_{k+1}}$ is new variable being set to TRUE.

$$(x_{i_1} \wedge x_{i_2} \dots x_{i_k}) \Rightarrow x_{i_{k+1}} \qquad \square$$

or a sub-set of them

Thm: Horn(F) output is correct.

Proof Case I: Horn(F) outputs an assignment. ✓

Case II: Horn(F) outputs "F is unsatisfiable."

only sets those variables to be TRUE that are TRUE in all satisfying assignments.

pure-negative clauses are unsatisfied

$\Downarrow$

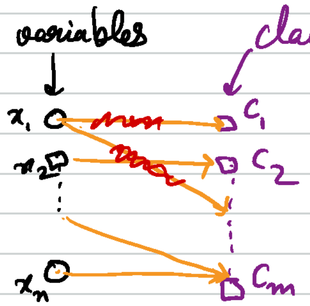
F has no satisfying assignment.

Can we improve the running time?

Idea: Don't need to recompute F every time a variable is set to TRUE.

$\overbrace{x_1 \Rightarrow x_2}^{C_1}$

$\overbrace{x_2 \Rightarrow x_3}^{C_2}$



1) right hand side variable in clause with in-degree 0 need to set to TRUE
2) when a variable is set to TRUE all outgoing edges (from it) can be removed.

Algorithm:

$Q = \phi$ // variables set to TRUE are pushed to the queue

Find nodes of indegree 0 and add them to Q also set the variable to TRUE

while (Q is non-empty)

- Remove x from Q
- Delete edges out of x
- For any node with indegree 0 add it to Q and set variable to TRUE

Run-time: $O(|F| + n)$

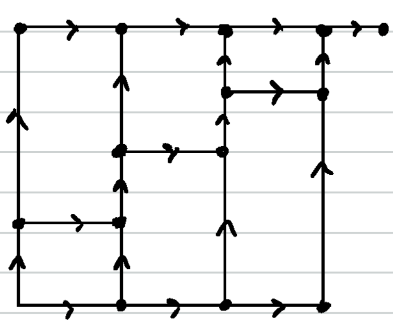
Dynamic Programming

versatile & powerful algorithm design tool.

Longest Path in a DAG.

Input: $G = (V, E)$ a DAG

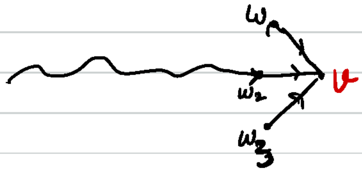
Goal: Find ℓ the length of the longest path in G.



$\boxed{\text{Subproblem:}}$ $L(v)$ = length of the longest path ending in v

$$\ell = \max_{v \in V} L(v)$$

Connecting the subproblems – Recurrence relation



$$L(v) = 1 + \max_{(w,v)\in E} L(w)$$

$\hookrightarrow$ 0 if v has no incoming edges.

Recursive Algorithm

$L(v)$

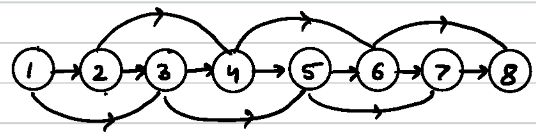
If no incoming edge to v

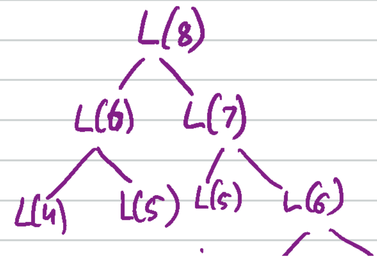
then

return 0

else

$1 + \max_{(w,v)\in V} L(w)$





$\boxed{\text{Avoiding repeat computation}}$ (memorization – don't recompute if computed once)

Input $G=(V,E)$

- Sort in topological order – i is the i^{th} vertex in topological ordering s.t. $\forall\ (i,j)\in E$ $j>i$
- For all i, set $L(i)=0$
- For all $i=1 \dots n$, set $L(i) = 1 + \max_{(j,i)\in E} L(j)$

  $\hookrightarrow$ 0 if no incoming edges.

Run-time.

$O(|V|+|E|)$

Longest Increasing Subsequence.

$4 \to 5 \quad 6 \quad 6 \quad 9$

$4 \to 2 \quad 3 \quad 6 \quad 9$

$\vdots$

$2 \leftrightarrow 2 \quad 9$

5 2 8 6 3 6 9 7

Input: $a_1 \quad a_2 \quad \ldots \quad a_n$

Goal: l = length of the longest increasing subsequence.

Reduces to finding longest path in a graph.

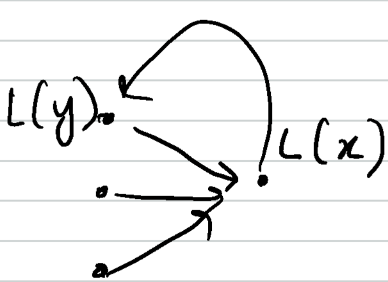
$G = (V, E)$

$V = \{1 \ldots n\}$ $\quad\quad E = \{ (i,j) \mid i < j \;\&\; a_i \le a_j \}$

Approach

1) Define appropriate subproblems
2) Write a recurrence relation
3) Determine order of computation.

↳ induces a DAG structure



Edit Distance

Input : $x[1....n]$ & $y[1....m]$

Goal : Find minimum # of keystrokes needed to edit x to y

1 keystroke is needed to add a character
remove a character
substitute a character

Example #

CAP → CUP → 1

AAPPL → APPLE → 2

SUNNY → SNOWY

SUNNY $\xrightarrow[u]{del}$ SNNY $\xrightarrow[N\to O]{sub}$ SNO␣Y $\xrightarrow[W]{insert}$ SNOWY

How to visualize?

DISK ⟶

D	S	I	K
N	N	␣	Y
_	O	W	Y

S	U	N	N	Y
S	U	N	N	Y

$\xrightarrow[u]{del}$

S	U	N	N	Y
S	_	N	N	Y

$\xrightarrow[N\to O]{sub}$

S	U	N	N	Y
S	_	N	O	Y

$\xrightarrow[W]{insert}$

S	U	N	N	_	Y
S	_	N	O	W	Y

S	U	N	N	Y	␣	␣	␣	␣	␣
␣	␣	␣	␣	␣	S	N	O	W	Y