CS 170

Lecture 10

- use iClicker — join.iclicker.com/RGWR
- Midterm 2 on 11/06

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"

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

Special case $(\Rightarrow x)$ → (x)

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

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

* Start by assigning all variables to FALSE
* Switch a variable to TRUE if it must be set so in any satisfying assignment

Horn(F)

Set $x_1, \dots x_n$ to FALSE
While ∃ an unsatisfied implication clause C] Part A
  set the right-hand variable in C
    to TRUE

if all pure negative clauses are satisfied] Part B
  then return the assignment $x_1, \dots x_n$
else return "F is unsatisfiable."

Run-time: $O(|F| \times n)$
↳ size of the formula.

Local to Global

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

Proof:

$k = 1 \rightarrow \Rightarrow x$
↳ but x must be TRUE in all satisfying assignments.

$k \rightarrow k+1 \rightarrow$

$(x_1 \wedge x_2 \dots x_k) \Rightarrow x_{k+1}$

or a subset of them

they are all TRUE in every satisfying assignment

Thus, x_{k+1} must also be TRUE in every satisfying assignment □

Thm: Horn(F) output is correct.

Proof

Case I: Horn(F) outputs as assignment. then that assignment passed both parts A & B and thus is satisfying.

Case II: Horn(F) outputs no assignment.

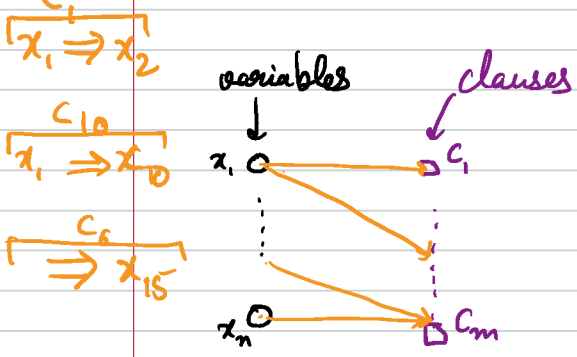
↳ (a) we only set a variable to TRUE if this is required in every satisfying assignment

(b) pure-negative clauses are unsatisfied with this assignment

→ 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.



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

Find nodes of indegree 0 (clause) and add them to Q (right hand variable) also set the variable to TRUE

while (Q is non-empty)

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

$O(|F| + n)$

Dynamic Programming

versatile & powerful algorithm design tool.

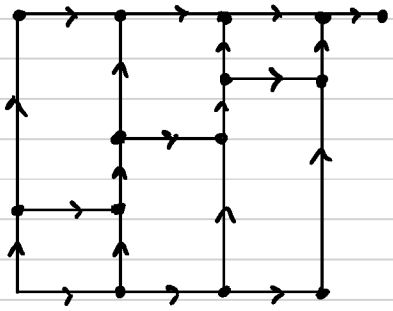
Approach

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

Longest Path in a DAG.

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

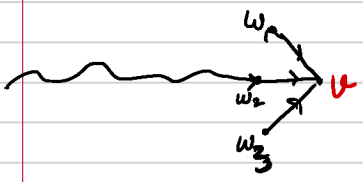
Goal: Find ℓ the length of the longest path in G.



Subproblem: $L(v)$ = length of the longest path ending in v

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

Connecting the subproblems – Recurrence relation

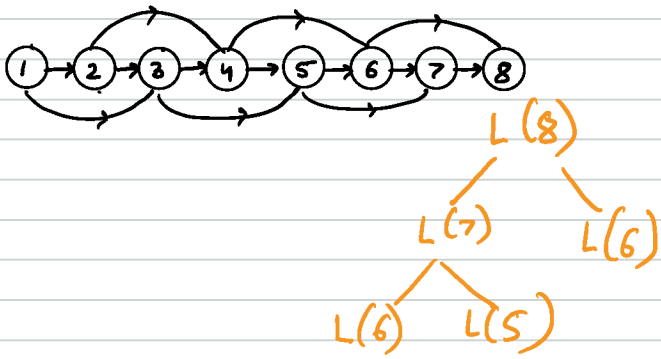


$$L(v) = \begin{cases} 1 + \max_{(w,v) \in E} L(w) \\ 0 \end{cases}$$

if v has no incoming edges

Recursive Algorithm

```
L(v)
    If no incoming edge to v
        then
            return 0
        else
            1 + max_{(w,v)∈V} L(w)
```



Avoiding repeat computation (memorization – don't recompute if computed once)

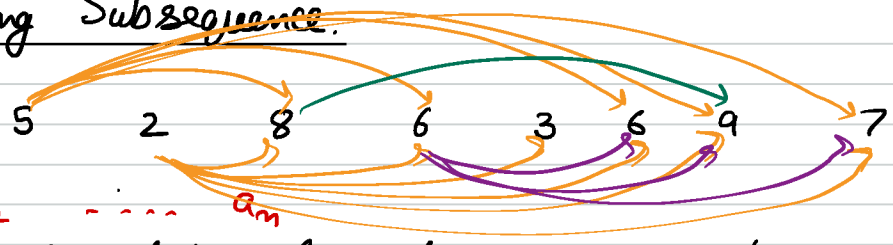
Input $G = (V, E)$

- Sort in topological order – i is the i^{th} vertex in topological ordering s.t. $\nexists\ (i, j) \in E \quad j > i$
- For all i, set $L(i) = 0$
- For all $i = 1 \dots n$, set $L(i) = 1 + \max_{(j,i) \in E} L(j)$
  - ↳ 0 if no incoming edges.

Run-time.

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

Longest Increasing Subsequence.



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

Goal: ℓ = length of the longest increasing subsequence.

Reduces to the problem of finding longest path in a Graph.

$G = (V, E)$ where $V = \{1 \dots n\}$ $\quad E = \{(i, j) \mid i < j \ \&\ a_i \leq a_j\}$

Edit Distance

Input : $x[1 \ldots n]$ & $y[1 \ldots m]$

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 → 3

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

How to visualize?

DISK →

D	I	S	K
N	␣	N	Y
–	W	O	Y

S	U	N	N	Y	␣	U	U	U	U
U	␣	U	U	␣	S	N	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

Step 1: Define subproblems

$$ED(x[1 \ldots i), y[1 \ldots j)) = ED(i, j)$$

as the edit distance between $x[1 \ldots i]$ & $y[1 \ldots j]$

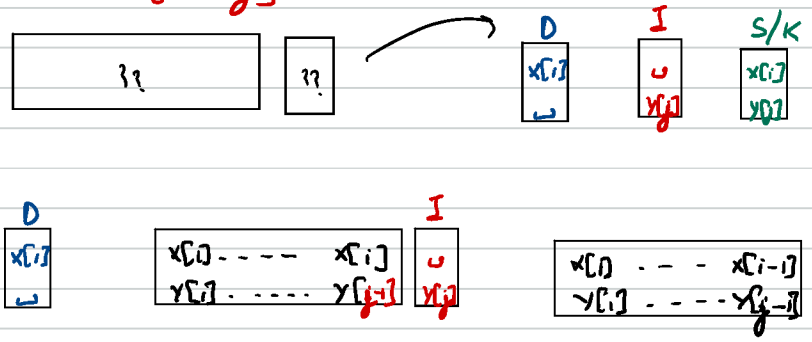
$\phi, y[1] \to 1$

$\phi, y[2] \to 2$

$\phi, \phi \to 0$

Step 2: Recurrence relation

edit $x[1 \ldots i] \rightarrow y[1 \ldots j]$



Base Case

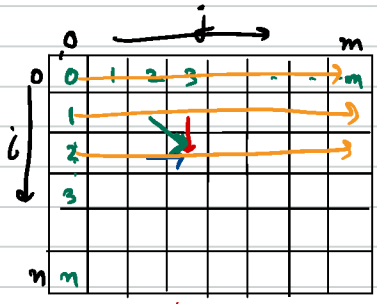
$$E(0,i) = E(i,0) = i$$

$$ED(i,j) \xrightarrow{\min} \begin{cases} ED(i-1, j) + 1 \\ ED(i, j-1) + 1 \\ ED(i-1, j-1) + 1 \rightarrow \text{if substitution} \\ \quad\quad + 0 \rightarrow \text{if match} \end{cases}$$

Step 3: Pick the order of computation

$E(i,j)$

Dependency



$\hookrightarrow O(m\,n)$

	ϕ	S	N	O	W	Y
ϕ	0	1	2	3	4	5
S	1	0				
O	2					
N	3					
N	4					
Y	5					