CS 331, Fall 2026
Lecture 6 (9/14)

Today: - Knapsack
- LCS & friends
- Game theory

Knapsack (Part III, Section 3.3)

Recall subset sum from last time:

Input: L: list of n natural #'s
$U \in \mathbb{N}$: target value

Output: True/False, $\exists S \subseteq [n]$, $\sum_{i \in S} L[i] = U$?

Key idea: 2D memoization, prefix x value

General strategy applies to integer-constrained optimization problems, e.g. knapsack

Input: W: n natural #'s

$B \in \mathbb{N}$: target value

Output: $\exists\, S \subseteq [n]$, $\sum_{i \in S} W(i) = B$?

Subset sum

Input: W: n natural #'s (weights)

$B \in \mathbb{N}$: weight budget

V: n real positive #'s (values)

0-1 knapsack

Output: $\max_{S \subseteq [n]} \sum_{i \in S} V(i)$

$\sum_{i \in S} W(i) \leq B$

Applications: budget-constrained decision making problems.

Before: $S[j][b]$ = can you hit target b using first j items?

$j \in [n], b \in [B]$

Intuition: decision to take/not take $L[j]$ affects remaining value, must store

Now: $S[j][b]$ = max value w/ weight budget b using first j items

$j \in [n], b \in [B]$

DP formula:

$$S[j][b] = S[j-1][b]$$
$$\text{OR } S[j-1][b-W[j]]$$

subset sum

$$S[j][b] = \max(S[j-1][b], V[j] + S[j-1][b-W[j]])$$

0-1 knapsack

Also runs in $O(nB)$ time. (row-by-row)

Extension Unbounded knapsack

You can take multiple copies of any item.

Goal: maximize $\sum_{i \in [n]} c_i V(i)$ s.t.

$$c \in \mathbb{Z}^n_{\geq 0} \quad \text{(counts)}$$

$$\sum_{i \in [n]} c_i W(i) \leq B$$

DP: $S(b)$ = max achievable w/ budget b

$$S(b) = \max\left(0, \max_{\substack{i \in [n] \\ W(i) \leq b}} S(b - W(i)) + V(i)\right)$$

0: take nothing

$S(b - W(i)) + V(i)$: take item i

Runtime: $O(B) \times O(n) = O(nB)$

$O(B)$: # subprobs

$O(n)$: time / subprob

Longest Common Subsequence (Part III, Section 4.1)

Strings Ω = universe

e.g. $\{$'a', 'b', ..., 'z', '1', ..., '0'$\}$

Characters

String of length n: ordered list of n characters from Ω "algorithms"

$= \{$'a', 'l', ... 'm', 's'$\}$

Substring: contiguous sublist "algo"

Subsequence: delete any characters, concatenate the rest "arms"

LCS: natural distance measure on strings

Input: X, length m string

Y, length n string

(e.g. DNA)

Output: $|Z|$, largest possible length of common subsequence Z of X & Y

Example

X = "algorithms"

Y = "complexity"

$LCS(X, Y) = 3$

Conclusion: they are both "it"

$\underset{Z}{\operatorname{argmax}} |Z|$

Key idea: 2-D DP again

$S(i)(j) = LCS(X(:i), Y(:j))$

↑ prefixes ↑

2 cases: Can we match $X(i)$, $Y(j)$?

Case 1: No ($X(i) \neq Y(j)$)

e.g. "algor", "compl" ($i=5, j=5$)

What is last char of Z?

$X(i)$, $Y(j)$, or neither. (not both)

If not $X(i)$, Plan A: $S(i-1)(j)$

If not $Y(j)$, Plan B: $S(i)(j-1)$

Case 2: Yes $(X(i) = Y(j))$

e.g. "al", "compl" $(i=2, j=5)$

Now we can make $X(i) = Y(j)$ last character in Z.

Plan C: $1 + S[i-1][j-1]$

Summary:

$$S[i][j] = \max \begin{cases} S[i-1][j], & \text{no } X[i] \\ S[i][j-1], & \text{no } Y[j] \\ S[i-1][j-1] + \mathbb{1}(X[i] = Y[j]) & \text{both} \end{cases}$$

Runtime: $O(mn)$

Extension — Multiple strings

$LCS(W, X, Y)$: longest mutually common subsequence

lengths ℓ m n

Same idea.

$$S(i)(j)(k) = LCS(W(:i), X(:j), Y(:k))$$

$$= \max\Big(S(i-1)(j)(k), S(i)(j-1)(k), S(i)(j)(k-1), S(i-1)(j-1)(k-1) + 1\big(W(i)=X(j)=Y(k)\big)\Big)$$

Extension — Edit distance

How many ops needed to turn X to Y?

Ops: Insert, Delete, Substitute

Example

X = "kitten"

Y = "sitting"

3 steps: "kitten" → "kittin"

→ "sittin" → "sitting"

Observation: Suppose no substitutions.

Optimal edit sequence:

$X \xrightarrow{\text{(deletions)}} Z$

$Z \xrightarrow{\text{(insertions)}} Y$

Z is LCS of X & Y

Proof: 1) All deletions from X in optimal moves.

2) Rearrange so all deletions first.

3) $X \to Z \to Y$ shortest if Z longest.

Edit distance: Same idea.

All moves are 1) Delete from X

2) Insert from Y

3) Substitute X to Y

or sequence can be more shorter.

$S(i)(j)$ = Edit distance of $X(:i)$, $Y(:j)$

$$S(i)(j) = \min\Big(S(i)(j-1) + 1 \quad \text{insert } Y(j)$$

$$S(i-1)(j) + 1 \quad \text{delete } X(i)$$

$$S(i-1)(j-1) + \mathbb{1}(X(i) \neq Y(j)) \Big) \quad \text{sub } X(i) \to Y(j)$$

not necessary if equal

Runtime: Again $O(mn)$.

Game theory (Part III, Section 5.1)

Consider two-player win-lose game.

Alice vs. Bob. E.g. Tic-tac-toe, chess, ...

Move 1 Alice

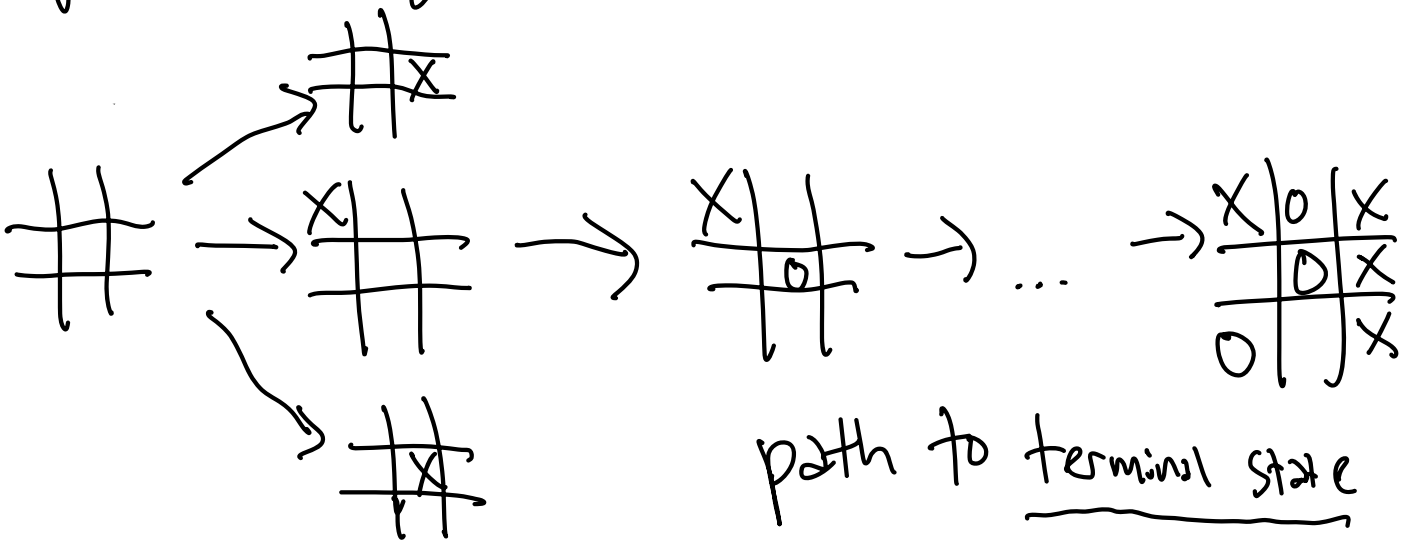
Move 2 Bob

⋮

Move k (game over, Alice wins)

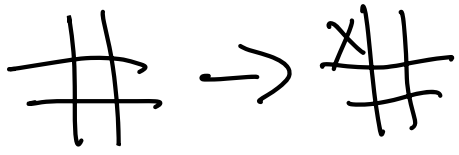
Game graph (potentially huge, pruning in practice)

Vertices: game states



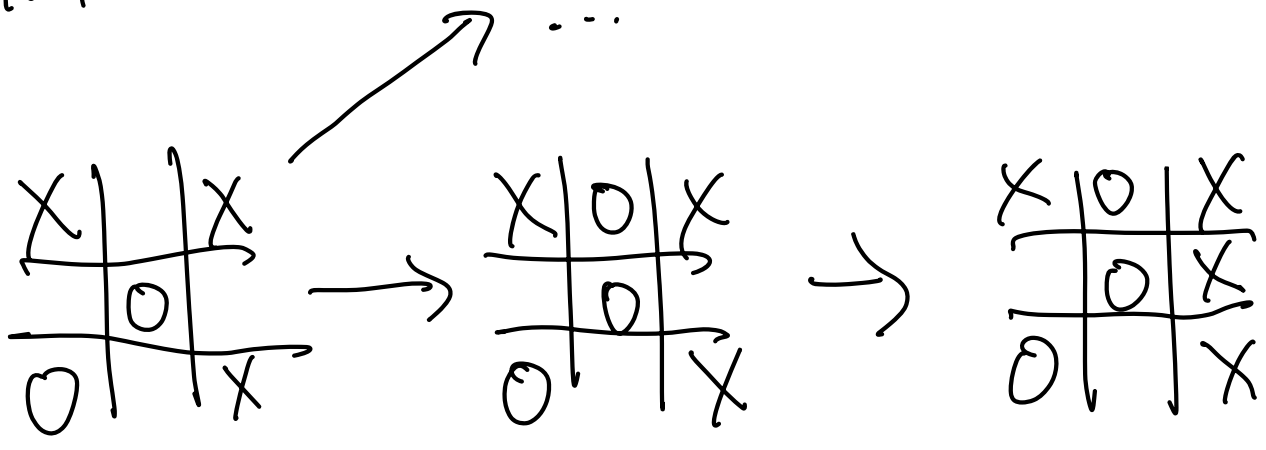
Terminal state: Vertex where game is over.

1) Alice wins
2) Bob wins
3) Tie (we'll mostly ignore)

Game moves: directed edges 

Q: Can Alice always force a win?

Intuition:



"forced win"

Whatever Bob does, Alice can win.

DP solution: label all vertices v True "forced win"
False

Work backwards from leaves ≡ terminal states
(next class)

(With ties, label them False.)

$$S(v) = \begin{cases} \underset{\text{edge } (v,w)}{\text{OR}} \left(S(w) \right) & \text{Alice plays} \\ \underset{\text{edge } (v,w)}{\text{AND}} \left(S(w) \right) & \text{Bob plays} \end{cases}$$

Alice wins if she can move to another winning state.

Bob wins if he can move to another losing state.

So Alice needs all possible moves to be True.

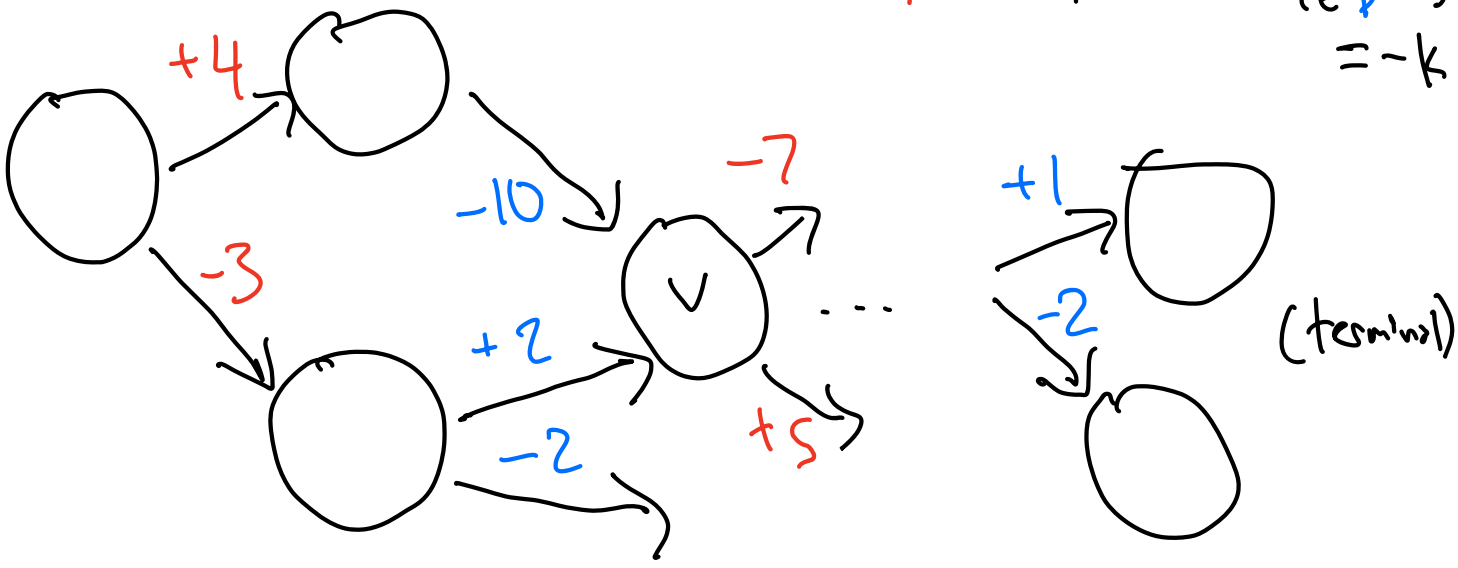
Extension — Zero-sum games

Alice and Bob have scores.

@ end of game, sum = 0.

e.g. win-lose (score at end is winner +1, loser −1)

dividing items ($v_1 + \dots + v_n = T$, split to A, B; score: $\sum_{i \in A} v_i = k$, $-T + \sum_{i \in B} v_i = -k$)



$$S[v] = \begin{cases} \max_{\text{edge}(v,u)} & S[u] + \text{score change}(v,u) \quad \text{Alice} \\ \min_{\text{edge}(v,u)} & S[u] + \text{score change}(v,u) \quad \text{Bob} \end{cases}$$

↑ max guaranteed score if dropped in at state v.