

CS 577 - Dynamic Programming

Marc Renault

Department of Computer Sciences
University of Wisconsin – Madison

Fall 2024

TopHat Section 001 Join Code: 728978

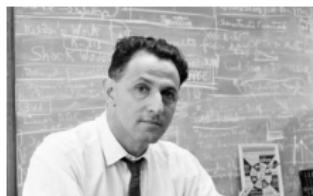
TopHat Section 002 Join Code: 412331



[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN*](#)[LS*](#)[RNA*](#)

DYNAMIC PROGRAMMING

DYNAMIC PROGRAMMING



It is “programming” that is “dynamic”!

Richard Bellman

Why “Dynamic Programming”?

Reasons for the name:

- In the 1950s, “programming” was about “planning” rather than coding.
- “Dynamic” is exciting – Air Force director didn’t like research and wanted pizzazz.
- “Dynamic” sounds better than “linear” (Re: rival Dantzig).

DYNAMIC PROGRAMMING



It is “programming” that is “dynamic”!

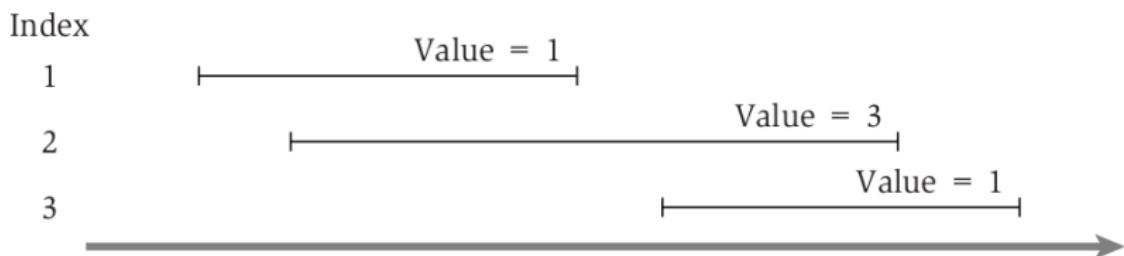
Richard Bellman

What is it?

- Your new favourite algorithmic technique.
- Extreme Divide and Conquer
- Many sub-problems, but not quite brute-force.
- Dynamic in that it calculates a bunch of solutions from the “smallest” to the “largest”.

WEIGHTED INTERVAL SCHEDULING

WEIGHTED INTERVAL SCHEDULING



Problem Definition

- Requests: $\sigma = \{r_1, \dots, r_n\}$
- A request $r_i = (s_i, f_i, v_i)$, where s_i is the start time, f_i is the finish time, and v_i is the value.
- Objective: Produce a *compatible* schedule S that has maximum value.
- Compatible schedule S : $\forall r_i, r_j \in S, f_i \leq s_j \vee f_j \leq s_i$.

TH1: What is the value of the FF heuristic? 2.

TH2: What is the optimal value? 3.

RECURSIVE SOLUTION

Recursive Procedure

- ① Assume σ ordered by finish time (asc).
- ② Find the optimal value in sorted σ of first j items:
 - ① Find largest $i < j$ such that $f_i \leq s_j$.
 - ② $\text{OPT}(j) = \max(\text{OPT}(j - 1), \text{OPT}(i) + v_j)$

Proof of optimality.

By strong induction on j .

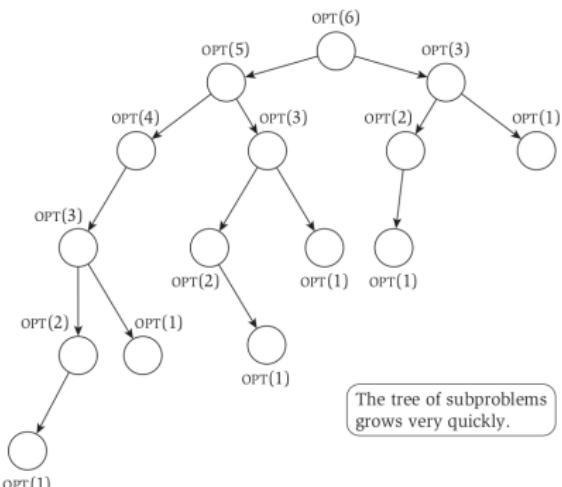
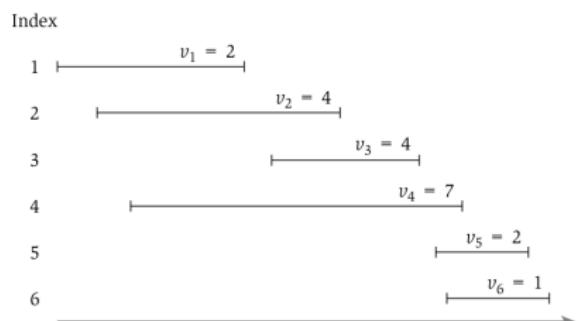
Base cases: $j = 0$ or $j = 1$: Only 1 possible optimal solution.

Inductive step:

- By ind hyp, we have opt for $j - 1$ and opt for i .
- Sorting order assures the dichotomy that the last interval is either in the solution or not.
- Take the max of whether or not a given interval is included. □

CONSIDER THE RECURSION

$$\text{OPT}(j) = \max(\text{OPT}(j - 1), \text{OPT}(i) + v_j)$$



TH3: What is the asymptotic number of recursive calls with n jobs? $O(2^n)$

MEMOIZING THE RECURSION

Memoization

- Not a typo.
- Coined in 1989 by Donald Michie.
- Derived from latin “memorandum”, meaning “to be remembered”.

Basic Technique

- Calculate once: store the value in array and retrieve for future calls.
- Can be implemented recursively, but tends to be more natural as an iterative process.

DYNAMIC PROGRAM SOLUTION

Algorithm: WEIGHTINTDP

Sort σ by finish time

$m[0] := 0$

for $j = 1$ to n **do**

 Find index i

$m[j] = \max(m[j - 1], m[i] + v_j)$

end

DP Solutions

- DP algorithms are formulaic.
- We understand how loops work.
- NO Pseudocode.

We want:

- Definitions required for algorithm to work
- Description of matrix
- Bellman Equation
- Location of solution, order to populate the matrix

DYNAMIC PROGRAM SOLUTION

Definitions required for algorithm to work

- σ sorted by finish time, ascending order.
- For a given job at index j , $i_j < j$ is the largest index such that $f_{i_j} \leq s_j$.

Description of matrix

- 1D array M , where $M[j]$ is the maximum value of a compatible schedule for the first j items in sorted σ . Initialize $M[1] = v_1$.

Bellman Equation

- $M[j] = \max\{M[j - 1], M[i_j] + v_j\}$

Solution, order to populate

- The maximum value of a compatible schedule for the n jobs is found at $M[n]$. Populate from 2 to n .

ANALYZE THE ALGORITHM

DP Solution

- σ sorted by finish time, ascending order.
- For a given job at index j , $i < j$ is the largest index such that $f_i \leq s_j$.
- Bellman Equation: $m[j] = \max(m[j - 1], m[i] + v_j)$

Runtime

- Preprocessing:
 - Sorting jobs: $O(n \log n)$.
- Populate the matrix:
 - Number of cells: $O(n)$
 - Cost per cell: Finding i : $O(n)$ linear search, $O(\log n)$ binary search

Overall: $O(n^2)$ linear search, $O(n \log n)$ binary search

ANALYZE THE ALGORITHM

DP Solution

- σ sorted by finish time, ascending order.
- For a given job at index j , $i < j$ is the largest index such that $f_i \leq s_j$.
- Bellman Equation: $m[j] = \max(m[j - 1], m[i] + v_j)$

What about the schedule S ?

Trace back from the optimal value:

- Job j is part of the optimal schedule from 1 to j iff $v_j + \text{OPT}(i) \geq \text{OPT}(j - 1)$

BASIC DP OUTLINE

Algorithm Template

- Preprocessing of data
- Populate the matrix:
 - Iterate over the cells in the correct order.
 - Understand the work done per cell.

Algorithm Guidelines

- ❶ There are only a polynomial number of subproblems.
- ❷ The solution to the larger problem can be efficiently calculated from the subproblems.
- ❸ Natural ordering of the subproblems from “smallest” to “largest”.

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN*](#)[LS*](#)[RNA*](#)

LONGEST INCREASING SUBSEQUENCE

LONGEST INCREASING SUBSEQUENCE

Problem

- Given an integer array $A[1..n]$.
- Find the longest increasing subsequence. That is, let i be a sequence of indexes, we have $A[i_k] < A[i_{k+1}]$ for all k .

Subsequence

- For a sequence A , a subsequence S is a subset of A that maintains the same relative order.
- Ex: I like watching the puddles gather rain.
 - puddles: subsequence, substring (contiguous)
 - late train: subsequence, not substring (not contiguous)

TH1: For an array of length n , how many subsequences? 2^n

RECURSIVE APPROACH

Algorithm: LIS

Input : Integer k , and array of integers $A[1..n]$.

Output: Return length of LIS where every value $> k$.

if $n = 0$ **then return** 0

else if $A[1] \leq k$ **then**

return LIS($k, A[2..n]$)

else

skip := LIS($k, A[2..n]$)

take := LIS($A[1], A[2..n]$) + 1

return max{*skip, take*}

end

TH2: For an array $A[1..n]$, how would you find the length of the LIS using the LIS(\cdot) algorithm? LIS($-\infty, A[1..n]$)

RECURSIVE APPROACH

Algorithm: LIS

Input : Integer k , and array of integers $A[1..n]$.

Output: Return length of LIS where every value $> k$.

if $n = 0$ **then return** 0

else if $A[1] \leq k$ **then**

return LIS($k, A[2..n]$)

else

skip := LIS($k, A[2..n]$)

take := LIS($A[1], A[2..n]$) + 1

return max{ skip, take }

end

TH3: Run time of the algorithm for a length n array? $O(2^n)$

TH4: How many distinct recursive calls for a length n array?

$O(n^2)$

DYNAMIC PROGRAM FOR LIS

Description of matrix

2D array L , where $L[i, j]$ is the maximum LIS of $A[j..n]$ with every item $> A[i], i < j$.

Bellman Equation

$$L[i, j] = \begin{cases} 0, & \text{if } j > n \\ L[i, j + 1], & \text{if } A[i] \geq A[j] \\ \max\{L[i, j + 1], L[j, j + 1] + 1\}, & \text{otherwise} \end{cases}$$

Solution and populating L

- Solution in $L[0][1]$; add $A[0] = -\infty$.
- Populate j from n to 1 ; i from 0 to $j - 1$ or $j - 1$ to 0 .
- Run time: $O(n^2)$

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN*](#)[LS*](#)[RNA*](#)

DYNAMIC PROGRAMMING FOR GAMES

DYNAMIC PROGRAMMING FOR GAMES

Games

- Some number of players (1 to many).
- Set of rules with some objective.
- Huge domain, started by Von Neumann, that spans many fields such as Economics, Math, Biology, and Computer Science.

DP for Games

In many games, DP is a natural paradigm for an optimal strategy.

COINS IN A LINE

Players

Two players:



Alice
(Player A)



Bob
(Player B)

Rules

- n (even) coins in a line; each coin has a value.
- Starting with Alice, each player will pick a coin from the head or the tail.
- Winner: Player with the max value at the end; winning player keeps the coins.

GREEDY APPROACHES

Largest Coin

[1,3,6,3]

A: 3; [1,3,6]

B: 6; [1,3]

A: 6; [1]

B: 7; []

GREEDY APPROACHES

Largest Coin

Even or Odd

[1,3,6,3,1,3]

A: 3; [1,3,6,3,1]

B: 1; [1,3,6,3]

A: 6; [1,3,6]

B: 7; [1,3]

A: 9; [1]

B: 8; []

[1,3,6,3,1,3]

A: 3; [1,3,6,3,1]

B: 1; [3,6,3,1]

A: 4; [3,6,3]

B: 4; [6,3]

A: 10; [3]

B: 7; []

- Alice can always win.
- But are we optimal? No

NATURAL DICHOTOMY

Head or Tail?

- Two players: Assume that Bob will play optimally.
- For Alice's k th turn:
 - Coin array: $C[i..j]$
 - $\max\{c[i] + \text{BobOpt}(c[i+1..j]), c[j] + \text{BobOpt}(c[i..j-1])\}$
- $\text{BobOpt}(c[i..j]) := \min\{\text{AliceOpt}(c[i+1..j]), \text{AliceOpt}(c[i..j-1])\}$

TH1: How many dimensions for DP array? 2

HEAD OR TAIL DP

DP Description

- 2D array M :
 - $M[i, j]$ is the maximum value possible for Alice when choosing from $c[i..j]$, assuming Bob plays optimally.

- Bellman Equation:

$$M[i, j] = \max\{c[i] + \min\{M[i + 2, j], M[i + 1, j - 1]\}, \\ c[j] + \min\{M[i + 1, j - 1], M[i, j - 2]\}\}$$

- $M[i, i] = c[i]$ for all i .
- $M[i, j] = \max\{c[i], c[j]\}$ for $i = j - 1$.
- Populate i from $n - 2$ to 1 ; j from n to 3 for $i < j - 1$.
- Solution: $M[1, n]$
- Runtime: $O(n^2)$
- Proof of correctness: Strong induction on the cell population order.

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN*](#)[LS*](#)[RNA*](#)

MAX SUBARRAY

MAX SUBARRAY

Problem

Given an array A of integers, find the (non-empty) contiguous subarray of A of maximum sum.

Exercise – Teams of 3 or so

- Solve the problem in $\Theta(n^2)$.
- Solve the problem in $O(n \log n)$.
- Prove correctness and complexity.

PART 1: GIVE A $\Theta(n^2)$ SOLUTION.

Algorithm: CHECKALLSUBARRAY

```

Input : Array  $A$  of  $n$  ints.
Output: Max subarray in  $A$ .
Let  $M$  be an empty array
for  $i := 1$  to  $\text{len}(A)$  do
    for  $j := i$  to  $\text{len}(A)$  do
        if  $\text{sum}(A[i..j]) > \text{sum}(M)$ 
             $M := A[i..j]$ 
        end
    end
end
return  $M$ 
```

Analysis

- Correct: Checks all possible contiguous subarrays.
- Complexity:
 - Re-calculating the sum will make it $O(n^3)$. Key is to calculate the sum as you iterate.
 - For each i , check $n - i + 1$ ends. Overall:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} = \Theta(n^2)$$

PART 2: GIVE AN $O(n \log n)$ SOLUTION.

Algorithm: MAXSUBARRAY

Input : Array A of n ints.

Output: Max subarray in A .

if $|A| = 1$ **then return** $A[1]$

$A_1 := \text{MAXSUBARRAY}(\text{Front-half of } A)$

$A_2 := \text{MAXSUBARRAY}(\text{Back-half of } A)$

$M := \text{MIDMAXSUBARRAY}(A)$

return *Array with max sum of $\{A_1, A_2, M\}$*

Algorithm: MIDMAXSUBARRAY

Input : Array A of n ints.

Output: Max subarray that crosses midpoint A .

$m := \text{mid-point of } A$

$L := \text{max subarray in } A[i, m - 1] \text{ for } i = m - 1 \rightarrow 1$

$R := \text{max subarray in } A[m, j] \text{ for } j = m \rightarrow n$

return $L \cup R // \text{ subarray formed by combining } L \text{ and } R.$

PART 2: GIVE AN $O(n \log n)$ SOLUTION.

Algorithm: MAXSUBARRAY

Input : Array A of n ints.

Output: Max subarray in A .

if $|A| = 1$ **then return** $A[1]$

$A_1 := \text{MAXSUBARRAY}(\text{Front-half of } A)$

$A_2 := \text{MAXSUBARRAY}(\text{Back-half of } A)$

$M := \text{MIDMAXSUBARRAY}(A)$

return *Array with max sum of $\{A_1, A_2, M\}$*

Analysis

- Correctness: By induction, A_1 and A_2 are max for subarray and M is max mid-crossing array.
- Complexity: Same recurrence as MERGESORT.

PART 3: GIVE AN $O(n)$ SOLUTION.

DP Solution

- 1D array s , where $s[i]$ contains the value of the max subarray ending at i . ($O(n)$ cells)
- Bellman equation: $s[i] = \max(s[i - 1] + A[i], A[i])$. ($O(1)$ time)
- Solutions is: $\max_j\{s[j]\}$. ($O(n)$ time)

But we need the subarray not the value!

- Use a parallel array that memoizes the starting index of the subarray ending at i :

$$\text{start}[i] = \begin{cases} \text{start}[i - 1] & \text{if } s[i - 1] + a[i] > a[i] \\ i & \text{, otherwise} \end{cases}$$

- Or, trace back from max value at index j until $s[i] = A[i]$.

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN*](#)[LS*](#)[RNA*](#)

SUBSET AND KNAPSACK

SUBSET PROBLEM

Problem Definition

- A single machine that we can use for time W .
- A set of jobs: $1, 2, \dots, n$.
- Each job has a run time: w_1, w_2, \dots, w_n .
- What is the subset S of jobs to run that maximizes $\sum_{i \in S} w_i \leq W$?

Greedy Heuristics

- Decreasing weights: $\{W/2 + 1, W/2, W/2\}$
- Increasing weights: $\{1, W/2, W/2\}$

DYNAMIC PROGRAMMING APPROACH

1D Approach

- if $n \notin S$, then $v[n] = v[n - 1]$
- if $n \in S$, then $v[n] = ?$
 - Accepting n does automatically exclude other items.

Need to consider more

To solve $v[n]$, we need to consider:

- the best solution with $n - 1$ previous items restricted by W , and
- the best solution with $n - 1$ previous items restricted by $W - w_n$

DYNAMIC PROGRAMMING APPROACH

2D Approach

- 2D Matrix v :
 - i : Item indices from 0 to n .
 - w : Max weight from 0 to W .
 - $v[i, w]$ is the subset of the first i items of maximum sum $\leq w$.
- Indicator: $x_{i,w} := 0$ if $w_i > w$ and 1 otherwise.
- Bellman Equation:

$$v[i, w] = \max(v[i - 1, w], x_{i,w} \cdot (v[i - 1, w - w_i] + w_i))$$

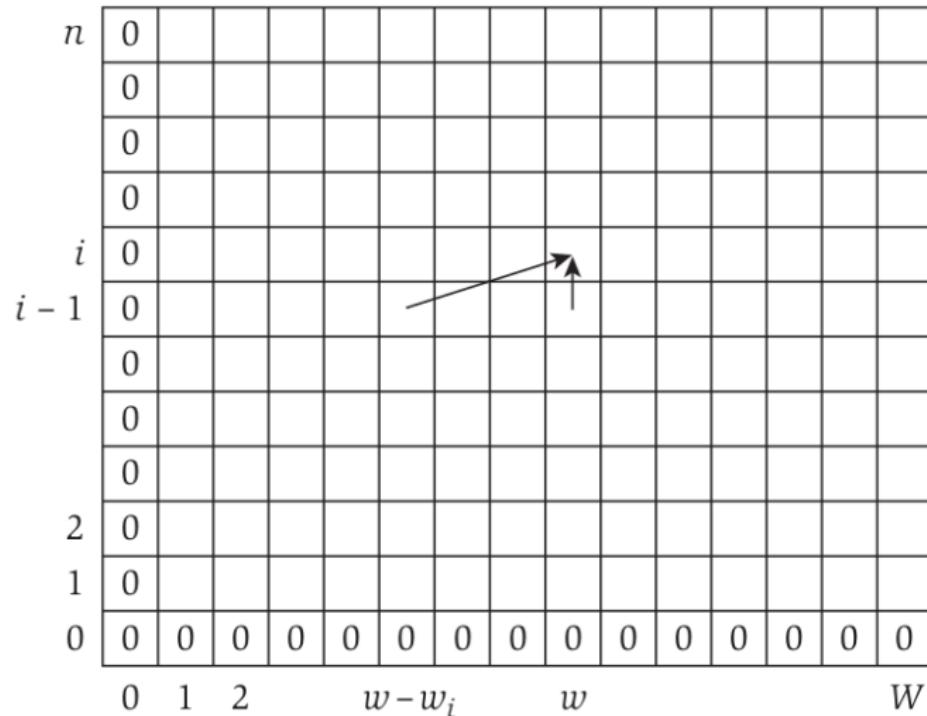
- $v[0, w] := 0$ for all w and $v[i, 0] := 0$ for all i
- Solution value: $v[n, W]$.

TH7: Running time to populate the matrix: $O(nW)$

TH8: Is this polynomial? No, *pseudo-polynomial* because of W which is unbounded.

SUBSET VISUALIZATION

Matrix Visualization:



SUBSET VISUALIZATION

Example Run:

$$W = 6, \text{ items } w_1 = 2, w_2 = 2, w_3 = 3$$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 1$

3							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 2$

3							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 3$

DYNAMIC PROGRAMMING APPROACH

2D Approach

- 2D Matrix v :
 - i : Item indices from 0 to n .
 - w : Max weight from 0 to W .
 - $v[i, w]$ is the subset of the first i items of maximum sum $\leq w$.
- Indicator: $x_{i,w} := 0$ if $w_i > w$ and 1 otherwise.
- Bellman Equation:

$$v[i, w] = \max(v[i - 1, w], x_{i,w} \cdot (v[i - 1, w - w_i] + w_i))$$

- $v[0, w] := 0$ for all w and $v[i, 0] := 0$ for all i
- Solution value: $v[n, W]$.

How can we recover the subset itself?

TH9: Running time of recovery of subset: $O(n)$

KNAPSACK EXTENSION



Problem Definition

- You are a thief with a knapsack that can carry W weight of goods.
- A set of items: $1, 2, \dots, n$.
- Each item has a weight: w_1, w_2, \dots, w_n .
- Each item has a value: v_1, v_2, \dots, v_n .
- What is the subset S of items to steal that maximizes $\sum_{i \in S} v_i$ with the constraint that $\sum_{i \in S} w_i \leq W$?

EXERCISE: SOLVE THIS WITH DP IN $O(nW)$.

DP Solution

- 2D Matrix:
 - i : Item indices from 0 to n .
 - w : Max weight from 0 to W .
 - $v[i, w]$ is the subset of the first i items of maximum total value with a sum of weights $\leq w$.
- Indicator: $x_{i,w} := 0$ if $w_i > w$ and 1 otherwise.
- Bellman Equation:

$$v[i, w] = \max(v[i - 1, w], x_{i,w} \cdot (v[i - 1, w - w_i] + v_i))$$

- $v[0, w] := 0$ for all w and $v[i, 0] := 0$ for all i
- Solution value: $v[n, W]$.

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN*](#)[LS*](#)[RNA*](#)

EDIT DISTANCE

EDIT DISTANCE

Problem

Minimum number of letter

- insertions: adding a letter,
- deletions: removing a letter,
- substitutions: replacing a letter

to change string $A[1..m]$ to string $B[1..n]$.

Ex: TUESDAY → THUESDAY → THURSDAY

Or, align and count mismatched letters

T UESDAY

THURSDAY

RECURSIVE APPROACH

Smaller Subproblems

- Let $A[1..m]$ and $B[1..n]$ be the 2 input strings.
- What is the edit distance for $A[1..i]$ and $B[1..j]$:
 - Insertion: $\text{Edit}(i, j) = \text{Edit}(i, j - 1) + 1$.
 - Deletion: $\text{Edit}(i, j) = \text{Edit}(i - 1, j) + 1$.
 - Substitution: $\text{Edit}(i, j) = \text{Edit}(i - 1, j - 1) + A[i] \neq B[j]$
 - $i = 0$: $\text{Edit}(i, j) = j$.
 - $j = 0$: $\text{Edit}(i, j) = i$.

DYNAMIC PROGRAM FOR EDIT DISTANCE

Description of matrix

2D array E , where $E[i, j]$ is the edit distance for $A[1..i]$ and $B[1..j]$.

Bellman Equation

$$E[i, j] = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \min\{E[i, j - 1] + 1, E[i - 1, j] + 1, \\ & E[i - 1, j - 1] + A[i] \neq B[j]\}, & \text{otherwise} \end{cases}$$

Solution and populating L

- Solution in $E[m, n]$
- Set $E[0, j] = j$; $E[i, 0] = i$; populate from 1 to n , 1 to m .
- Run time: $O(mn)$

SPACE SAVINGS

Bellman Equation

$$E[i, j] = \begin{cases} i, & \text{if } j = 0 \\ j, & \text{if } i = 0 \\ \min\{E[i, j - 1] + 1, E[i - 1, j] + 1, \\ & E[i - 1, j - 1] + A[i] \neq B[j]\}, & \text{otherwise} \end{cases}$$

How much space do we need?

- Notice that $E[i][j]$ depends on $E[i, j - 1]$, $E[i - 1, j]$, and $E[i - 1, j - 1]$.
- We only need previous and current row of matrix for calculations.

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN*](#)[LS*](#)[RNA*](#)

SHORTEST PATH

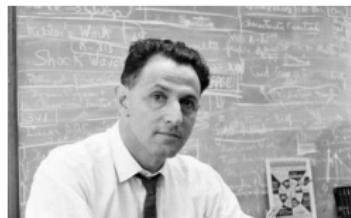
SHORTEST PATH

GOING NEGATIVE

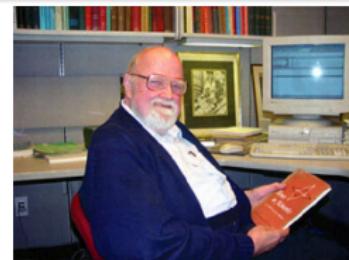
Problem Definition

We have a directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$ and a node s that has a path to every other node in V . For each edge $e = (i, j)$, c_{ij} is the weight of the edge, and there are no cycles with negative weight.

- What is the shortest path from s to each other node?



Richard Bellman



L R Ford Jr.

SHORTEST PATH

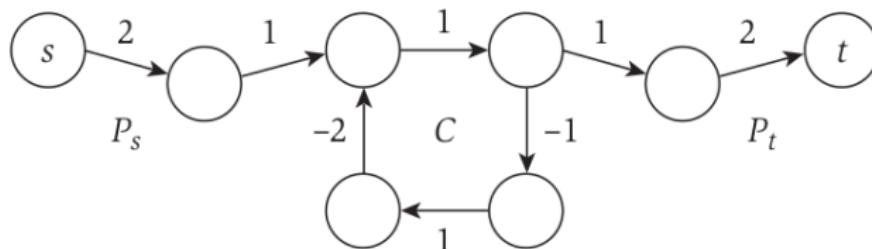
GOING NEGATIVE

Problem Definition

We have a directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$ and a node s that has a path to every other node in V . For each edge $e = (i, j)$, c_{ij} is the weight of the edge, and there are no cycles with negative weight.

- What is the shortest path from s to each other node?

Why no negative cycles?



DIJKSTRA'S

Algorithm: *Dijkstra's*

Let S be the set of explored nodes.

For each $u \in S$, we store a distance value $d(u)$.

Initialize $S = \{s\}$ and $d(s) = 0$

while $S \neq V$ **do**

Choose $v \notin S$ with at least one incoming edge
originating from a node in S with the smallest

$$d'(v) = \min_{e=(u,v): u \in S} d(u) + \ell_e$$

Append v to S and define $d(v) = d'(v)$.

end

return S

DIJKSTRA'S

Negative Problem

- Lose guarantee that minimum edge between S and $V \setminus S$ is part of minimum path.

Why not just boost all edges by max negative value plus a bit (β)?

- A path with x edges: Cost increases $x \cdot \beta$.
- Solution in new graph is not guaranteed to be optimal in original graph.

BELLMAN-FORD

Observation 1

If G has no negative cycles, then there exists a shortest path from s to t that is simple, and has at most $n - 1$ edges.

Dynamic Program

- 2D matrix M of # edges in path \times vertices.
 - $M[i][v]$ is the shortest path from v to t using $\leq i$ edges.
 - Solution: $M[n - 1][s]$
- Dichotomy:
 - Use $\leq i - 1$ edges.
 - Use $\leq i$ edges.

$$M[i][v] = \min\{M[i - 1][v], \min_{w \in V}\{M[i - 1][w] + c_{vw}\}\},$$

where $c_{vw} = \infty$ if no edge from v to w .

BELLMAN-FORD ANALYSIS

$$M[i][v] = \min\{M[i-1][v], \min_{w \in V}\{M[i-1][w] + c_{vw}\}\}$$

Worst Case: n nodes

- # of Cells: $O(n^2)$.
- Cost per cell: $O(n)$.
- Overall: $O(n^3)$.

Worst Case: n nodes, m edges

- For each node v , we only need to consider outgoing edges to w (denoted by η_v).
- For every node v , we need to do this calculation for $0 \leq i \leq n - 1$ lengths.
- Overall: $O(n \sum_{v \in V} \eta_v) = O(mn)$.

BELLMAN-FORD ANALYSIS

$$M[i][v] = \min\{M[i - 1][v], \min_{w \in V}\{M[i - 1][w] + c_{vw}\}\}$$

Worst Case: n nodes, m edges

- Overall: $O(n \sum_{v \in V} \eta_v) = O(mn)$.

Space Saving: $O(n)$.

- To build row i :
 - We only need $i - 1$ values for each node.
 - $M[v] = \min\{M[v], \min_{w \in V}\{M[w] + c_{vw}\}\}$ for each i .
- Recovery of actual path: An additional array $first[v]$ that maintains the first hop from v to t .

NEGATIVE CYCLES

Observation 2

If there is a negative cycle along the path from s to t , then the shortest path is $-\infty$.

Observation 3

$M[i][v] = M[n - 1][v]$ for all $i > n - 1$ and all nodes v if there are no negative cycles on the paths to t .

Augmented Graph for Negative Cycle Finding

- Add a node t with an incoming edge from all other nodes with cost 0.
- Run Bellman-Ford from any node s to t until number of edges n .
- If, for some v , $M[n][v] \neq M[n - 1][v]$, then there is a negative cycle.

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN*](#)[LS*](#)[RNA*](#)

SEQUENCE ALIGNMENT

SEQUENCE ALIGNMENT

Scorites	CTTAGATCGTACCCAA---ATAATTAC
Carenum	CTTAGATCGTACCCAA-TAC-TTTAC
Pasimachus	ATTAAGATCGTACCCACTATAAG
Pheropsophus	CTTAGATCGTTCCAC---ACATTAC
Brachinus armiger	ATTAAGATCGTACCCAC---ATATATTAC
Brachinus hirsutus	ATTAAGATCGTACCCAC---ATAATTAC
Aptinus	CTTAGATCGTACCCAC---ACATTAC
Pseudomorpha	CTTAGATCGTACCC---ACATTAC

Needleman–Wunsch Problem

- An alphabet S .
- Strings $X = x_1x_2 \dots x_m$ and $Y = y_1y_2 \dots y_n$ from S .
- A matching $M = \{(i, j)\}$ of pairs without crossings, where $i \in [1, m]$ and $j \in [1, n]$.
- Cost:
 - Gaps (unmatched indexes) have a cost of δ .
 - For each symbol pair $p, q \in S$, α_{pq} is the matching cost.
- Goal: Find the matching that minimizes the cost.

DESIGNING NEEDLEMAN–WUNSCH ALGORITHM

Basic Dichotomy

In optimal alignment M , either $(m, n) \in M$ or $(m, n) \notin M$.

Lemma 1

Let M be any alignment of X and Y . If $(m, n) \notin M$, then either the m th position of X , or the n th position of Y is not matched in M .

Proof.

- By way of contradiction, assume that $(m, n) \notin M$, and $(m, j), (i, n) \in M$ for $i < m$ and $j < n$.
- Contradicts the non-crossing requirement.



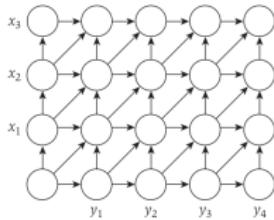
DESIGNING NEEDLEMAN–WUNSCH ALGORITHM

Key Concepts for Optimality

In an optimal alignment M , at least one of the following is true:

- ① $(m, n) \in M$; or
 - ② the m th position of X is not matched; or
 - ③ the n th position of Y is not matched.
-
- 2D matrix called A , where $A[i][j]$ is alignment of minimum cost for $x_1x_2\dots x_i$ and $y_1y_2\dots y_j$.
 - $A[i][j] = \min\{\alpha_{x_i y_j} + A[i-1][j-1], \delta + A[i-1][j], \delta + A[i][j-1]\}$
 - Runtime: $O(mn)$.

GRAPHING THE ALGORITHM



Theorem 2

Let $f(i, j)$ denote the minimum cost of a path from $(0, 0)$ to (i, j) in G_{XY} . Then, $\forall i, j f(i, j) = A[i][j]$.

Proof.

- By strong induction on $(i + j)$.
- Base case: $i + j = 0$. We have $f(0, 0) = 0 = A[0][0]$.
- Induction hypothesis: The claim holds for all pairs (i', j') such that $i' + j' < i + j$.
- Inductive step:

$$\begin{aligned}
 f(i, j) &= \min\{\alpha_{x_i y_j} + f(i - 1, j - 1), \delta + f(i - 1, j), \delta + f(i, j - 1)\} \\
 &= \min\{\alpha_{x_i y_j} + A[i - 1][j - 1], \delta + A[i - 1][j], \delta + A[i][j - 1]\} \\
 &= A[i][j]
 \end{aligned}$$

□

SEQUENCE ALIGNMENT EXAMPLE

$$A[i][j] = \min\{\alpha_{x_i y_j} + A[i-1][j-1], \delta + A[i-1][j], \delta + A[i][j-1]\}$$

- “mean” vs “name”

- $\delta = 2; \alpha = \begin{cases} 0 & \text{if same letter} \\ 3 & \text{if vowel to consonant} \\ 1 & \text{otherwise} \end{cases}$

n	8	6	5	4	6
a	6	5	3	5	5
e	4	3	2	4	4
m	2	1	3	4	6
-	0	2	4	6	8
	-	n	a	m	e

SEQUENCE ALIGNMENT EXAMPLE

$$A[i][j] = \min\{\alpha_{x_i y_j} + A[i-1][j-1], \delta + A[i-1][j], \delta + A[i][j-1]\}$$

- “mean” vs “name”
- $\delta = 2; \alpha = \begin{cases} 0 & \text{if same letter} \\ 3 & \text{if vowel to consonant} \\ 2 & \text{otherwise} \end{cases}$

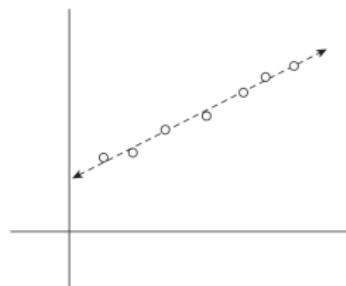
n	8	6	6	6	8
a	6	6	4	6	6
e	4	4	4	6	4
m	2	2	4	4	6
-	0	2	4	6	8
	-	n	a	m	e

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN*](#)[LS*](#)[RNA*](#)

LEAST SQUARES

SEGMENTED LEAST SQUARES

Problem Setup



- Set of n points: $P := \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ on the plane.
- Suppose $x_1 < x_2 < \dots < x_n$.
- Find $L : y = ax + b$ that minimizes:
$$\text{Error}(L, P) = \sum_{i=1}^n (y_i - ax_i - b)^2 .$$

Problem Formulation

- Partition the points (by x) into contiguous subsets.
- Minimize the sum of $\text{Error}(L, p_i) + C$ for all subsets, where C is a fixed cost per subset.

DP SOLUTION

$$s[j] = \min_{1 \leq i \leq j} (e_{i,j} + C + s[i - 1])$$

Notes

- $e_{i,j}$ is the min error for a partition from i to j .
- C is added each time as we are adding a new partition.
- $s[i]$ is optimum up to point i .

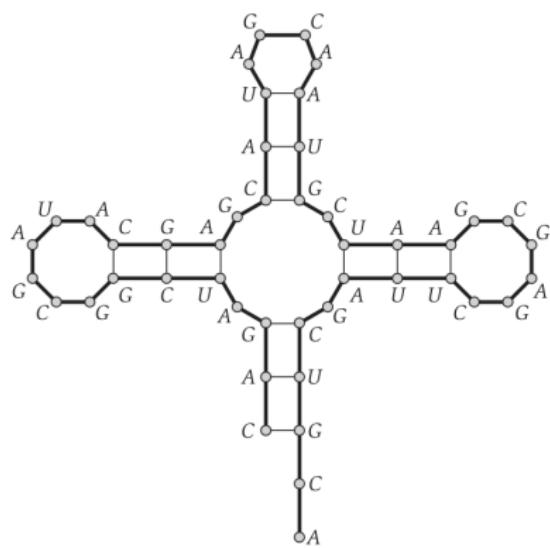
Complexity

- Preprocessing error calc $e_{i,j}$ can be done in $O(n^3)$.
- Number of cells: $O(n)$.
- Work done for cell j : $O(j)$.
- Overall: $O(n^2)$.

[DP](#)[WIS](#)[LIS](#)[GAMES](#)[MAX SUBARRAY](#)[SUBSET](#)[EDIT](#)[SP](#)[ALIGN*](#)[LS*](#)[RNA*](#)

RNA SECONDARY STRUCTURE

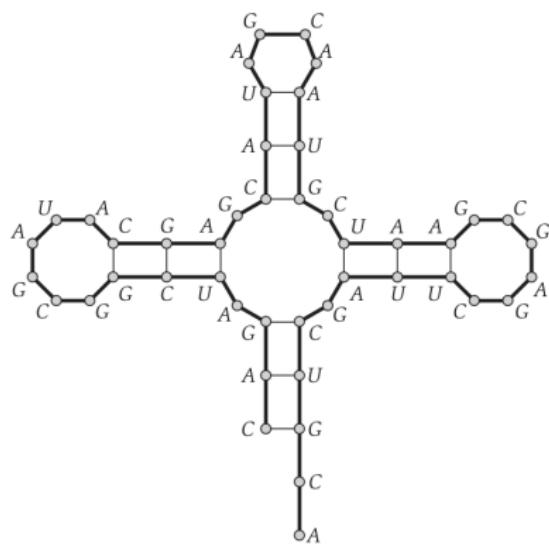
RNA SECONDARY STRUCTURE



Problem Definition

- RNA tends to loop back on itself, forming base pairs.
- RNA alphabet: $\{A, C, G, U\}$.
- Valid pairs: (A, U) or (C, G) .
- Input: n length string:
 $B = b_1 b_2 \dots b_n$
- Output: Determine a secondary structure with maximum number of base pairs.

RNA SECONDARY STRUCTURE



Secondary Structure

$S = \{(i, j)\}$, where $i < j$ and $i, j \in \{1, \dots, n\}$, such that:

- ① No Sharp turns: $i < j - d$ for some constant d .
- ② All pairs are valid.
- ③ S is a matching: no base appears more than once.
- ④ Non-crossing: For any $(i, j), (i', j') \in S$, we cannot have $i < i' < j < j'$.

FIRST DYNAMIC PROGRAMMING ATTEMPT

1D Approach

- 1D array m , where $m[j]$ is the maximum # of pairs among: $b_1 b_2 \dots b_j$.
- No sharp turns: $m[j] = 0$ for $j \leq d + 1$.
- Solution: $m[n]$.

Recursive Sub-problems

Dichotomy:

- ① j is not a pair: $m[j] = m[j - 1]$.
- ② j is paired with $t < j - d$:
 - Non-crossing: No pairs between $[1, t - 1]$ and $[t + 1, j - 1]$.
 - Sub-problems:
 - ① Max pairs in $[1, t - 1]$: $m[t - 1]$.
 - ② Max pairs in $[t + 1, j - 1]$: Restricted to $b_{t+1} b_{t+2} \dots b_{j-1}$ which current DP does not calculate.

SECOND DYNAMIC PROGRAMMING ATTEMPT

2D Approach

- 2D array m , where $m[i][j]$ is the maximum # of pairs among: $b_i b_{i+1} \dots b_j$.
- No sharp turns: $m[i][j] = 0$ for $i \geq j - d$.
- Solution: $m[1][n]$.

Recursive Sub-problems

Dichotomy:

- ➊ j is not a pair: $m[i][j] = m[i][j - 1]$.
- ➋ j is paired with $i \leq t < j - d$
 - v_{ij} as indicator: 1 if valid pair, 0 otherwise
 - Non-crossing: No pairs between $[i, t - 1]$ and $[t + 1, j - 1]$.
 - Sub-problems:
 - ➌ Max pairs in $[i, t - 1]$: $m[i][t - 1]$.
 - ➍ Max pairs in $[t + 1, j - 1]$: $m[t + 1][j - 1]$.

SECOND DYNAMIC PROGRAMMING ATTEMPT

2D Approach

- 2D array m , where $m[i][j]$ is the maximum # of pairs among: $b_i b_{i+1} \dots b_j$.

Recursive Sub-problems

Dichotomy:

- ① j is not a pair: $m[i][j] = m[i][j - 1]$.
- ② j is paired with $i \leq t < j - d$
 - v_{ij} as indicator: 1 if valid pair, 0 otherwise
 - Non-crossing: No pairs between $[i, t - 1]$ and $[t + 1, j - 1]$.
 - Sub-problems:
 - ① Max pairs in $[i, t - 1]$: $m[i][t - 1]$.
 - ② Max pairs in $[t + 1, j - 1]$: $m[t + 1][j - 1]$.

$$m[i][j] = \max(m[i][j - 1], \max_{i \leq t < j - d} \{v_{tj} \cdot (1 + m[i][t - 1] + m[t + 1][j - 1])\})$$

RNA SECONDARY STRUCTURE EXAMPLE

$$m[i][j] = \max \left(m[i][j-1], \max_{i \leq t < j-d} \{ v_{tj} \cdot (1 + m[i][t-1] + m[t+1][j-1]) \} \right)$$

- $B = ACCGUAGU$ and $d = 4$

i					
4	0	0	0	0	0
3	0	0	1	1	
2	0	0	1	1	
1	1	1	1	1	2
j	6	7	8	8	9

Running Time

- # of cells: $O(n^2)$.
- Work per cell: $O(n)$.
- Overall: $O(n^3)$.