

CSE 202 - Algorithms

Dynamic Programming

5/1/03

CSE 202 - Dynamic Programming



Chess Endgames

- Ken Thompson (Bell Labs) solved all chess endgame positions with ≤ 5 pieces (1986). Some surprises:
 - K+Q beats K+2B. (Since 1634, believed to be draw).
 - 50 moves without a pawn move or capture is not sufficient to ensure it's a draw.
 - By now, most (all?) 6-piece endgames are solved too.
- Searching all possible games with 5 pieces, going forward 50 or so moves, is infeasible (today).
 - If typical position has 4 moves, search is 2^{100} long
 - $2^{100} = 2^{45}$ ops/second $\times 2^{25}$ seconds/year $\times 1$ billion years.
- There is a better way!

"Retrograde Analysis"

Insight: there aren't so many *different* positions.

With 5 specific pieces, there are $\leq 64^5$ positions.

64^5 is about 10^9 (large, but quite feasible).

Make memory location for each position.

"Position" includes whose move it is.

Store count of possible moves (for Black moves).

Initialize work list to all won positions for white.

For each "win" P in list, process all positions Q that can move to P (unless Q was already known to win for white).

If Q is a white move, mark Q as win and add it to list .

If Q is a black move, decrease possible move count by 1

If count == 0, mark Q as win for white and put it on list.

3

CSE 202 - Dynamic Programming

Why did this reduce work??

Can evaluate game tree by divide & conquer

Win-for-white (P) {

let Q_1, \dots, Q_k be all possible moves from P;

$W_i = \text{Win-for-white}(Q_i);$

if P is a white move and *any* $W_i = T$ then return T;

if P is a black move and *all* $W_i = T$ then return T;

return F; }

Don't allow repeated positions

Divide ← Combine

Inefficient due to overlapping subproblems.

Many subproblems share the same sub-subproblems

4

CSE 202 - Dynamic Programming

Dynamic Programming

Motto: "It's not dynamic and it's not programming"

For a problem with "optimal substructure" property...

Means that, like D&C, you can build solution to bigger problem from solutions to subproblems.

... and "overlapping subproblems"

which makes D&C inefficient.

Dynamic Programming: builds a big table and fills it in from small subproblems to large.

Another approach is memoization: ←

More overhead, but
(maybe) more intuitive

Start with the big subproblems,

Store answers in table as you compute them.

Before starting on subproblem, check if you've already done it.

5

CSE 202 - Dynamic Programming

Example: Longest Common Substring

- $Z = z_1 z_2 \dots z_k$ is a substring of $X = x_1 x_2 \dots x_n$ if you can get Z by dropping letters of X .
 - Example: "Hello world" is a substring of "Help!
I'm lost without our landrover".
- LCS problem: given strings X and Y , find the length of the longest Z that is a substring of both (or perhaps find that Z).

6

CSE 202 - Dynamic Programming

LCS has "optimal substructure"

Suppose $\text{length}(X) = x$ and $\text{length}(Y) = y$.

Let $X[a:b]$ mean the a -th to b -th character of X .

Observation: If Z is a substring of X and Y , then the last character of Z is the last character of both X and Y ...

so $\text{LCS}(X, Y) = \text{LCS}(X[1:x-1], Y[1:y-1]) \parallel X[x]$

or of just X , so $\text{LCS}(X, Y) = \text{LCS}(X, Y[1:y-1])$

or of just Y , so $\text{LCS}(X, Y) = \text{LCS}(X[1:x-1], Y)$

or neither so $\text{LCS}(X, Y) = \text{LCS}(X[1:x-1], Y[1:y-1])$

Dynamic Programming for LCS

Table: $L(i, j) = \text{length of LCS}(X[1:i], Y[1:j])$.

If the last character of both X and Y is the same ...

$\text{LCS}(X, Y) = \text{LCS}(X[1:x-1], Y[1:y-1]) \parallel X[x]$

otherwise, $\text{LCS}(X, Y) = \text{LCS}(X, Y[1:y-1])$

or $\text{LCS}(X, Y) = \text{LCS}(X[1:x-1], Y)$

or $\text{LCS}(X, Y) = \text{LCS}(X[1:x-1], Y[1:y-1])$

tells us:

$L(i, j) = \max \{ L(i-1, j-1) + (X[i]=Y[j]), L(i, j-1), L(i-1, j) \}$

Each table entry is computed from 3 earlier entries.

Dynamic Programming for LCS

↘ = match (increment)

		H	E	L	L	O	W	O	R	L	D
		0	0	0	0	0	0	0	0	0	0
L	0	0	0	1	1	1	1	1	1	1	1
A	0	0	0	1	1	1	1	1	1	1	1
N	0	0	0	1	1	1	1	1	1	1	1
D	0	0	0	1	1	1	1	1	1	1	2
R	0	0	0	1	1	1	1	1	2	2	2
O	0	0	0	1	1	2	2	2	2	2	2
V	0	0	0	1	1	2	2	2	2	2	2
E	0	0	1	1	1	2	2	2	2	2	2
R	0	0	1	1	1	2	2	2	3	3	3

9

CSE 202 - Dynamic Programming

But - there are other substructures

- (Work out the following at board):
 - Can we have $T(i) = \text{LCS}(X[1:i], Y[1,i])$?
 - How about, "the first half of X matches up with some part of Y, and the rest with the rest."
 - Suggests $\text{LCS}(X,Y) = \text{maximum (for } i=1,\dots,y) \text{ of } \text{LCS}(X[1:x/2], Y[1:i]) + \text{LCS}(X[x/2+1,x], Y[i+1,y])$
 - We can make the table 1-D instead of 2-D!

10

CSE 202 - Dynamic Programming

Summary

For a good dynamic programming algorithm:

- Table should be low-dimensional
 - keeps memory requirements low.
- Each table entry should be fast to calculate
 - keeps complexity low.

May decompose problems differently than a good Divide & Conquer algorithm.

How to get your name on an algorithm: Use Dynamic Programming

- Earley's algorithm:
Parse a string using a CFG.
- Viterbi's algorithm:
Break string into codewords
- Smith-Waterman algorithm:
Match up two strings of DNA or protein
- Floyd-Warshall:
All-pairs shortest paths.

How to find a Dynamic Programming algorithm

- For a moderately small problem, draw a tree representing possible choices you have in finding an optimal solution.
- Look for instances of common subproblems.
- Figure out how you could store the subproblem solutions in a table.
 - make a careful statement of what the table entries mean
- Figure out where to start filling in the table.
 - typically the smallest subproblems at the tree's bottom.
- Figure out how to fill in other table entries.
 - recursive step: how do you solve a larger subproblem given the solutions to smaller subproblems.

13

CSE 202 - Dynamic Programming

Example: Viterbi's Algorithm

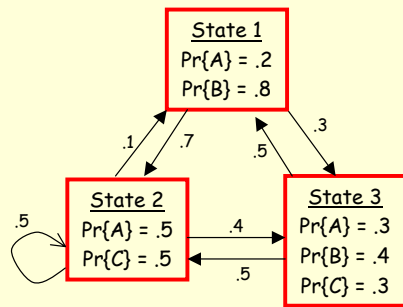
- Given a Hidden Markov Model (HMM) and a string $s = s_1 s_2 \dots s_n$, find most probable path taken by the Markov model that produces s .
 - HMM is a set of states, a set of transition probabilities between the states, and, for each state and each output character, a probability of producing that character at that state.
 - The probability of a path is the product of the transition probabilities along the path and the probabilities of producing the observed character are the states along the path.

14

CSE 202 - Dynamic Programming

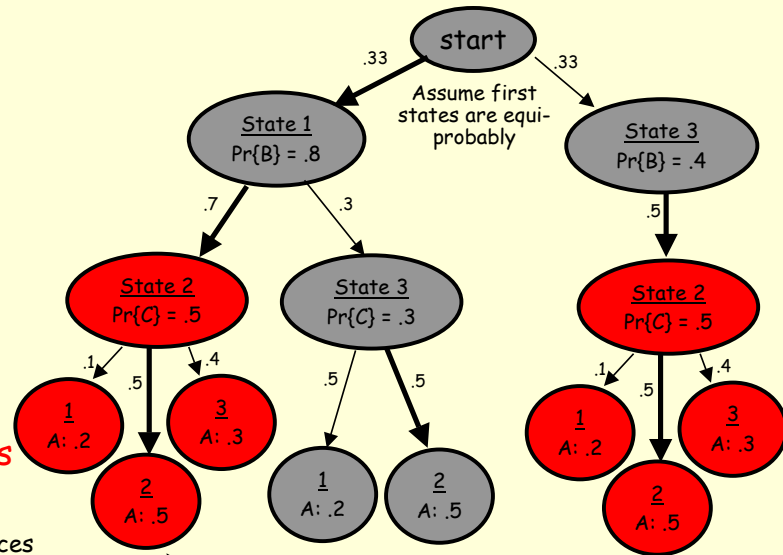
Discovering Viterbi's algorithm

Example HMM:



Tree for producing "BCA":

Only paths to leaves that produce BCA are shown.



The two red subtrees are identical!

They both represent the choices for producing "CA" starting in state 2.

Most probable path that produces "BCA":
(It's probability is $.33 \times .8 \times .7 \times .5 \times .5 = .0233$)

CSE 202 - Dynamic Programming

15

How to find a Dynamic Programming algorithm

- Figure out how you could store the subproblem solutions in a table.
 - Index the table by a state and a level in the tree
- Make a careful statement of what the table entries mean.
 - $T(i,k)$ = Probability of the most-probable path that starts in state i and produces the string s_k, s_{k+1}, \dots, s_n .
- Figure out where to start filling in the table.
 - You can fill in $T(i,n) = \Pr\{\text{state } i \text{ produces } s_n\}$ for each i .
- Figure out how to fill in other table entries.
 - $T(i,k) = \Pr\{i \text{ produces } s_k\} \times \text{Max}_{\text{state } j} [\Pr\{i \text{ moves to } j\} \times T(j,k+1)]$

16

CSE 202 - Dynamic Programming

Yet another example

Disclaimer: dynamic programming is NOT the best way to solve this problem!

- Given matrices M_1, M_2, \dots, M_n , where M_i has r_i rows and c_i columns, find the best order of performing the computation.
 - It will be some parenthesization, e.g.
 $(M_1 \times M_2) \times (M_3 \times (\dots \times M_n) \dots)$
 - Note that for $i = 1, \dots, n-1$, $c_i = r_{i+1}$.
 - Assume time to multiply an r by s matrix with an s by t matrix is rst .
- We worked out an example, found subproblems like "What is time to compute $M_i \times M_{i+1} \times \dots \times M_k$?"
- This led us to an $\Theta(n^3)$ algorithm.

17

CSE 202 - Dynamic Programming

Protein (or DNA) String Matching

Biological sequences:

A proteins is a string of amino acids (20 "characters").

DNA is a string of base pairs (4 "characters").

Databases of known sequences:

SDSC handles the Protein Data Base (PDB).

Human Genome (3×10^9 base pairs) computed in '01.

String matching problem:

Given a query string, find similar strings in database.

Smith-Waterman algorithm is most accurate method.

FastA and BLAST are faster approximations.

18

CSE 202 - Dynamic Programming

Smith-Waterman in $O(n^2)$ time

At cell i,j of table keep three numbers:

$s(i,j)$ = score of best matching of $X[1:i]$ and $Y[1:j]$.

whether or not it ends in a skip

$t(i,j)$ = best score of $X[1:i]$ and $Y[1:j]$ that skips X_i .

$u(i,j)$ = best score of $X[1:i]$ and $Y[1:j]$ that skips Y_j .

At each cell, compute:

$t(i,j) = \max \{ s(i-1,j) - c_0 - c_1, t(i-1,j) - c_1 \}$

$u(i,j) = \max \{ s(i,j-1) - c_0 - c_1, u(i,j-1) - c_1 \}$

$s(i,j) = \max \{ s(i-1,j-1) + A(x_i, y_j), t(i,j), u(i,j), 0 \}$