BMI-203: Biocomputing Algorithms Lecture 3: Dynamic Programming

Instructor: Conrad Huang

E-mail: conrad@cgl.ucsf.edu

Phone: 415-476-0415

Dynamic Programming

- Divide and conquer
- Example applications
  - Knapsack problem
  - Partition problem
  - Sequence alignment using local similarity
    - Matching one sequence onto another
    - Matching parts of one sequence onto parts of another

Divide and Conquer

- Formulate the solution to a large problem in terms of solutions to smaller problems
  - Binary search
  - Dynamic programming

Dynamic Programming Components

- Solution must be formulated as a recurrence relationship or recursive algorithm
- There must be an evaluation order that solves smaller problems before larger ones
- Storing the solutions to smaller problems makes solving the larger problem computable via table lookup
  - There cannot be too many “smaller” problems

Fibonacci Number by Recursion

- $F(n) = F(n - 1) + F(n - 2)$
  - where $F(0) = 0$, $F(1) = 1$
  - 0, 1, 1, 2, 3, 5, 8, 13, …
- Compute using recursion

```
def F(n):
    if n < 2:
        return n
    else:
        return F(n - 1) + F(n - 2)
```

*Recursion Complexity**

- Let f(n) be “the number of steps to compute the nth Fibonacci number using this algorithm”. We'll compute both upper bounds and lower bounds on f(n).
- We know that for n > 1 and constant c,

$$f(n) = f(n-1) + f(n-2) + c$$

*from http://www.math.grin.edu/~rebelsky/Courses/CS152/99S/Assignments/notes.06.html#Fibonacci

Complexity Upper Bound

- Since $f(n-1) > f(n-2)$ and c is expected to be small, we can say that $f(n) \leq 2*f(n-1)$. That means that:
  - $f(n) \leq 2*2*f(n-2)$, or
  - $f(n) \leq 2*2*2*f(n-3)$, or more generally,
  - $f(n) \leq 2^k*f(n-k)$.
  - For $k = (n-1)$, $f(n) \leq 2^{n-1}*f(1)$.
- Since $f(1)$ is a constant, $F(n)$ is $O(2^n)$.

Complexity Lower Bound

- Since f(n-2) < f(n-1), and c is expected to be small, we can say that f(n) >= 2*f(n-2). Using a similar analyses to the one above, we get that F(n) is in $\Omega(2^{n/2})$.
- Both lower bound and upper bound are exponential. We can therefore say that F(n) is an exponential algorithm.

Alternate Analysis

- $F(n) = (\Phi^n - (-\Phi)^{-n}) / \sqrt{5}$
  - The Golden Ratio Φ is approximately 1.618
- Because the leaves of our recursion are F(1) = 1, there must be at least F(n) leaves in order to sum up to F(n), which means the entire recursion runs in exponential time

Fibonacci Number by Dynamic Programming

- We have a recurrence relationship
  - F(n) = F(n - 1) + F(n - 2)
- We have an evaluation order which solves smaller problems before larger ones
  - F(1), F(2), …, F(n-2), F(n-1), F(n)
- There are not too many smaller problems
  - Exactly (n - 1)
- Dynamic programming is applicable

Fibonacci Number by Dynamic Programming

- Compute using dynamic programming

```
def F(n):
    f = [0, 1]
    for k in range(2, n + 1):
        f.append(f[k - 1] + f[k - 2])
    return f[n]
```

- Complexity is O(n) from “for” loop
  - Improvement over exponential time comes from storing and looking up intermediate results

Sequence Alignment using Dynamic Programming

- Similar to dynamic programming solutions to the approximate string matching problem
- Needleman, S.B. and Wunsch, C.D. A General Method Applicable to the Search for Similarities in Amino Acid Sequence of Two Proteins. *J. Mol. Biol.,* **48**, pp. 443-453, 1970.
- Smith, T.F. and Waterman, M.S. Identification of Common Molecular Subsequences. *J. Mol. Biol.,* **147**, pp. 195-197, 1981.

Approximate String Matching

- Given two character strings A (length n) and B (length m), what is the minimum number of substitutions, insertions and deletions required to transform A into B?
- Alternatively, what fragments of A and B are matched?

Cost Metric

- Best solution has the minimum total cost of:
  - substitution cost
    - replacing A_i with B_j
  - insertion cost
    - inserting B_j
  - deletion cost
    - skipping A_i

String Matching using Dynamic Programming

- Consider the last character from strings A and B. The possibilities are: (1) they match, (2) A_n is matched to something before B_m, (3) B_m is matched to something before A_n.

	xxxxxxxA$_n$
(1)	**xxxxxxxB**$_m$
(2)	**xxxxxxxxxxB**$_m$
(3)	**xxxxxB**$_m$

String Matching using Dynamic Programming

- Recurrence relationship
  - Let cost(A_n, B_m) be the cost of matching two strings where A_n and B_m are the last characters:

    cost(A_n, B_m) = minimum(

    cost(A_{n-1}, B_{m-1}) + substitution(A_n, B_m),

    cost(A_n, B_{m-1}) + insertion(B_m),

    cost(A_{n-1}, B_m) + deletion(A_n)

)

String Matching using Dynamic Programming

- Boundary conditions
  - Let A_1 and B_1 denote the first character of each string and insert dummy characters A_0 and B_0,

    cost(A_0, B_j) = initial_insertion(B_0 through B_j)

    cost(A_j, B_0) = initial_deletion(A_0 through A_i)

    cost(A_0, B_0) = 0
  - Note that initial insertion and deletion costs may be different than internal ones

String Matching using Dynamic Programming

- Order of evaluation
  - To compute cost(A_n, B_m), we need the results from cost(A_{n-1}, B_{m-1}), cost(A_n, B_{m-1}), and cost(A_{n-1}, B_m)
  - By induction, we need to compute cost(A_i, B_j) for all $i < n$ and $j < m$
- The number of intermediate solutions is $n \times m$

String Matching using Dynamic Programming

- Computing the cost
  - The n×m nature of the intermediate solutions suggests that they may be stored in a two-dimensional array, "H"
  - The evaluation order requires that the array be filled in a left-to-right, top-to-bottom fashion
  - The cost of aligning the two strings is in the cell at the bottom right corner

String Matching using Dynamic Programming

- Example
  - Substitution cost = 0 if $A_i = B_j$; 1 otherwise
  - Insertion and deletion costs = 1
  - Match "abbcd" to "accd"

H		a	b	b	c	d
	0	1	2	3	4	5
a	1	0	1	2	3	4
c	2	1	1	2	2	3
c	3	2	2	2	2	3
d	4	3	3	3	3	2

The best score is 2

String Matching using Dynamic Programming

- Recovering the alignment
  - Trace back from $H_{n,m}$
  - Find which operation resulted in the value of the cell and proceed to corresponding cell:
    - match → above-left
    - insert → above
    - delete → left

H		a	b	b	c	d
	0	1	2	3	4	5
a	1	0	1	2	3	4
c	2	1	1	2	2	3
c	3	2	2	2	2	3
d	4	3	3	3	3	2

```
a-ccd      ac-cd
abbcd      abbcd
```

String Matching using Dynamic Programming

- Recovering the alignment
  - The operation that resulted in a particular cell value may either be recorded when computing H, or recomputed during trace back
  - There are multiple back traces when a cell on the optimal path may be reached via more than one operation
  - All these back traces share the same best score and there are no back traces with a better score

Sequence Alignment

- Given two sequences, where are the similar fragments ...
  - if the two sequences are mostly similar?
    - global alignment where all residues are matched
  - if only parts are similar?
    - local alignment where only some residues are matched

Sequence Alignment

- This is similar to approximate string matching
- Algorithm transformation
  - substitution costs replaced by similarity scores
  - insertion and deletion costs replaced by gap penalties
  - best solution being maximum instead of minimum

Sequence Alignment

- The recurrence relationship is typically written as

  $H_{i,j}$ = maximum(

  $H_{i-1,j-1} + S(A_i, B_j)$,

  $H_{i-1,j}$ – gap penalty,

  $H_{i,j-1}$ – gap penalty

)

Amino Acid Score Matrices

- Substitutions scores are typically stored in a matrix whose rows and columns are residue types and whose cells are the similarity between the two types of residues
  - Genetic code matrix
  - PAM 250 (Dayhoff)
  - BLOSUM

Amino Acid Score Matrices

- Dynamic programming algorithm for sequence alignment is the same regardless of which matrix is used
- Matrix construction will be covered in another lecture

Needleman & Wunsch

- Global alignment method for finding identical matching residues
- Used multiple genetic code score matrix for testing evolutionary distance hypotheses

Needleman & Wunsch

- Genetic code score matrix
  - 1 for identical amino acids
  - 0 for amino acid pairs whose codons have no possible corresponding base
  - Range of values between 0 and 1 for pairs with maximum of one or two corresponding bases
- Constant gap penalty per insertion/deletion
  - Values range from 0 to 25

Needleman & Wunsch

- Procedure for comparing A to B
  - Produce a set of sequences by randomizing B
  - Align randomized set against A to obtain “random” score average and standard deviation
  - Align B to A to find a “maximum match” score
  - Compute number of standard deviations “maximum match” score is from “random” score

Needleman & Wunsch

- Why not examine just the best solution?
  - Dynamic programming will always produce the best answer for the problem at hand, whether the question is meaningful or not
  - The significance of the question can only be measured relative to some control
  - If “maximum match” score is more than 3 standard deviations above “random” score, the result is considered significant

Needleman & Wunsch

- Results and conclusions
  - Alignments between β-hemoglobin and myoglobin were significant for all seven sets of parameters tested
  - Alignments between ribonuclease and lysozyme were not significant for any of the seven sets of parameters tested
  - Beware of global alignments when the two sequences are not “closely” related

Smith & Waterman

- Local alignment method for identifying best matching fragment
- Score matrix remains unchanged
- Extends gap penalty to be length-dependent
  - Recurrence relationship changes

Smith & Waterman

- Recurrence relationship with length-dependent gap penalty

$H_{i,j}$ = maximum(

$H_{i-1,j-1} + S(A_i, B_j)$,

maximum($H_{i-k,j} - W_k$, $1 \leq k < i$),

maximum($H_{i,j-m} - W_m$, $1 \leq m < j$),

0

)

Smith & Waterman

- W_x is the penalty for a gap of length x
- Score at any cell should not drop below zero, which would penalize subsequent fragment alignments

Smith & Waterman

- Best aligned fragments are found by starting at cell with highest score and trace back to cell with zero score
- More aligned fragments may be found by back traces starting at other high scoring cells

Summary

- Dynamic programming is a divide-and-conquer method for solving problems with recurrence relationships
  - Results from intermediate results are stored so they do not need to be recomputed (space-time trade-off)
  - There is always a "best" solution, but it still may not be a reasonable solution

Homework

- Implement the Smith-Waterman algorithm
  - Write a function which accepts the following arguments:
    - 2 sequences
    - a similarity measure between sequence elements (either a function which takes two residue types as arguments, or a two-dimensional matrix)
    - a gap penalty function, which takes the gap length as an argument

Homework

- Implement the Smith-Waterman algorithm
  - Your function should return the common subsequence with the highest score
  - For grading purposes, your function should also print out the score matrix
  - *E-mail both your code and program output to conrad@cgl.ucsf.edu*

Homework

- Input data
  - Apply your code to the example from the Smith & Waterman paper
    - **CAGCCUCGCUUAG** *vs.* **AAUGCCAUUGACGG**
    - $S(A_i, B_j) = 1$ if $A_i = B_j$; $-1/3$ otherwise
    - $W_k = 1.0 + (1/3) * k$
    - Also try **GCCCUGCUUAG** *vs.* **UGCCGCUGACGG**
  - Your alignment and score matrix should match those published in the paper