Analysis and Design of Algorithms
Dynamic Programming

Lecture Notes by Dr. Wang, Rui
Fall 2008
Department of Computer Science
Ocean University of China

November 6, 2009

Introduction

- Dynamic Programming is an approach based on:
  1. Recurrence relation, a complete recurrence realtion composed of:
     - the base condition
     - the general recurrence
  2. Tabular computation
  3. Trace-back
- Often, dynamic programming solves a problem more general than proposed.
- We will learn the Dynamic Programming approach through the following problems:
  - Integer Partition
  - Sequence alignment
    - Global alignment
    - Local alignment

The Problem

- PARTITION is a decision problem defined as:

  Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$ of n nonnegative integers.

  Question: Is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} a = \sum_{a \in A - A'} a$?
- The brute-force algorithm that tries every possible subset consumes no less than 2^n time.
- The dynamic programming for PARTITION uses $O(nB)$ time, where $B = \sum_{a \in A} a$.

The Recurrence Relation

- Define $B = \sum_{a \in A} a$.
- If B odd, reply "no", and done.
- Let $t(i, j)$ denote the truth value of the statement

  "there is a subset of $\{a_1, a_2, \ldots, a_i\}$ for which the sum of the integers is exactly j",

  where $1 \leqslant i \leqslant n$ and $0 \leqslant j \leqslant B/2$.
- We are asked to compute $t(n, B/2)$.
- We derive
  - the general recurrence $t(i, j) = t(i-1, j) \vee t(i-1, j-a_i)$, with the base
    - $t(i, j) = T$ if $j = 0$,
    - $t(i, j) = T$ if $i = 1$ and $j = a_1$,
    - $t(i, j) = F$ if $0 < j \neq a_1$ and $i = 1$.

The Tabular Computation

- To compute $t(n, B/2)$ via

$$t(i,j) = \begin{cases} T & (i = 1, j = a_1) \vee j = 0, \\ F & i = 1 \wedge 0 < j \neq a_1, \\ t(i-1, j) \vee t(i-1, j-a_i) & 1 < i \leqslant n, 0 < j \leqslant B/2; \end{cases}$$

  we need to fill a table of $nB/2$ entries.
- Next is an example for instance $A = \{1, 9, 5, 3, 8\}$.

	$i \backslash t(i,j) \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$a_1 = 1$	1	T	T	F	F	F	F	F	F	F	F	F	F	F	F
$a_2 = 9$	2	T	T	F	F	F	F	F	F	F	T	T	F	F	F
$a_3 = 5$	3	T	T	F	F	F	T	T	F	F	T	T	F	F	F
$a_4 = 3$	4	T	T	F	T	T	T	T	F	T	T	T	F	F	T
$a_5 = 8$	5	T	T	F	T	T	T	T	F	T	T	T	T	T	T

- The computing proceeds row by row, from top to bottom.
- Each entry can be filled in constant time.
- The total time is $O(nB)$.

The Trace-Back

- If the problem is to search, e.g., search for the subset whose members sum to $B/2$, trace-back can be used to compute the solution.
- Trace-Back for the search version of PARTITION is illustrated in the following table.

	$i\backslash t(i,j)\backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$a_1 = 1$	1	T	T	F	F	F	F	F	F	F	F	F	F	F	F
$a_2 = 9$	2	T	T	F	F	F	F	F	F	F	T	T	F	F	F
$a_3 = 5$	3	T	T	F	F	F	T	T	F	F	T	T	F	F	F
$a_4 = 3$	4	T	T	F	T	T	T	T	F	T	T	T	F	F	T
$a_5 = 8$	5	T	T	F	T	T	T	T	F	T	T	T	T	T	T

Note that the algorithm actually solves a problem more general than PARTITION. **Question:** is $O(nB)$ a polynomial time?

Sequence Alignment 8 / 22

Problem Background

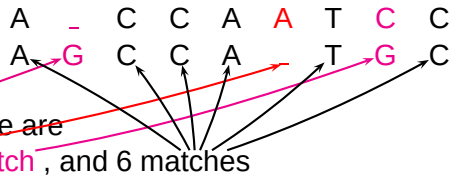
- Given two DNAs, high similarity means similar function.
- Thus, we want efficient algorithms to compare the similarity of two biological sequences.

Applications of Sequence Comparison

- in Computational Biology:
  - Inferring the biological function of gene.
    - When a gene looks similar to some gene with known function, we can conjecture that both genes have similar function.
  - Finding the evolution distance between two species.
    - Evolution modifies the DNA of species. By measuring the similarity of their genome, we know their evolution distance.
  - Helping genome assembly.
    - Based on the overlapping information of a huge amount of short DNA pieces, Human genome project reconstructs the whole genome. The overlapping information is done by sequence comparison.
  - Finding common subsequences in two genome.
  - Finding repeats within a genome.
  - ...many many other applications.
- In other areas: spelling correction, textual database retrieval, plagiarism detection, etc.

Alignment

- Given two strings, how can we measure their similarity?
- An alignment is a notion to measure similarity by transforming (inserting, deleting, and replacing the characters) the first string into the second.
- Take ACCAATCC and AGCCATGC for example, by inserting a space (-) after the the first 'A' and deleting the last 'A', we get



- In the above alignment, there are
  1 insert , 1 delete , 1 mismatch , and 6 matches

Similarity Functions

- To evaluate the goodness of the alignment, we need a similarity function specifying each individual match/mismatch/insert/delete contribute to the overall similarity ?
- E.g., match: 2, mismatch, insert, delete: -1.

δ	_	A	C	G	T
_	0	-1	-1	-1	-1
A	-1	2	-1	-1	-1
C	-1	-1	2	-1	-1
G	-1	-1	-1	2	-1
A	-1	-1	-1	-1	2

where, $\delta(A, A) = 2$, , $\delta(C, G) = -1$

Outline

- Global alignment:
  - Needleman-Wunsch algorithm.
  - Hirschbergs linear-space algorithm.
- Local alignment:
  - Smith-Waterman algorithm.

Global Alignment

- The following alignment has similarity score 9.

A	_	C	C	A	A	T	C	C
A	G	C	C	A	_	T	G	C

- The above alignment has the maximum score. Such an alignment is called an optimal alignment.
- The string alignment problem is to find the alignment with the maximum similarity score.
- String alignment problem is also called global alignment problem.

Involving the whole string

Needleman-Wunsch's Algorithm for Global Alignment

- Consider two strings $S[1..n]$ and $T[1..m]$.
- Define $V(i,j)$ be the score of the optimal alignment between the substrings $S[1..i]$, and $T[1..j]$.
- We are interested in $V(n,m)$.
- The Basis:
  - $V(0,0) = 0$.
  - $V(0,j) = V(0,j-1) + \delta(\_, T[j])$. Insert j times
  - $V(i,0) = V(i-1,0) + \delta(S[i], \_)$. Delete i times.

The Recurrence

For $i > 0$ and $j > 0$ we infer that.

- **Align $S[i]$ with $T[j]$:** then $S[i]$ and $T[j]$ either match or mismatch, and

$$V(i,\ j) = V(i-1,\ j-1) + \delta\left(S[i],\ T[j]\right)$$

$$\begin{array}{cccccc} x & x & x & \ldots & x & x \\ & & & & & | \\ y & y & y & \ldots & y & y \end{array}$$

- **delete $S[i]$:** then

$$V(i,\ j) = V(i-1,\ j) + \delta\left(S[i],\ \_\right)$$

$$\begin{array}{ccccccc} x & x & x & \ldots & x & x \\ & & & & & | \\ y & y & y & \ldots & y & \_ \end{array}$$

- **insert after $S[i]$:** then

$$V(i,\ j) = V(i,\ j-1) + \delta\left(\_,\ T[j]\right)$$

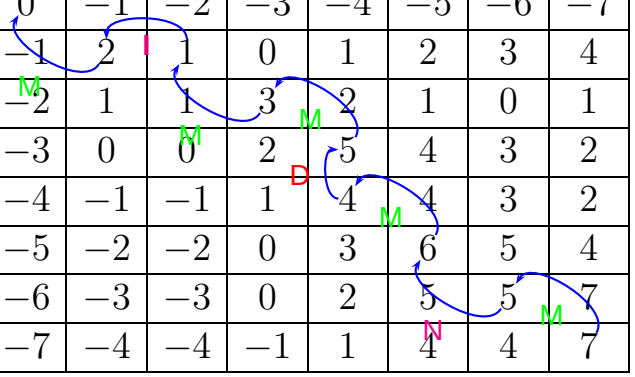
$$\begin{array}{ccccccc} x & x & x & \ldots & x & \_ \\ & & & & & | \\ y & y & y & \ldots & y & y \end{array}$$

The optimal value achieved by choose the maximum among the three cases. So we conclude that for $i, j > 0$

$$V(i,j) = \max \begin{cases} V(i-1,\ j-1) + \delta\left(S[i],\ T[j]\right) & match/mismatch, \\ V(i-1,\ j) + \delta\left(S[i],\ \_\right) & delete, \\ V(i,\ j-1) + \delta\left(\_,\ T[j]\right) & insert. \end{cases}$$

Example

	_	A	G	C	A	T	G	C
_	0	−1	−2	−3	−4	−5	−6	−7
A	−1	2	1	0	1	2	3	4
C	−2	1	1	3	2	1	0	1
A	−3	0	0	2	5	4	3	2
A	−4	−1	−1	1	4	4	3	2
T	−5	−2	−2	0	3	6	5	4
C	−6	−3	−3	0	2	5	5	7
C	−7	−4	−4	−1	1	4	4	7



The Trace-Back gives us that A_CAATCC / AGCA_TGC is optimal.

Analysis

- We need to fill in all entries in the table, which is of size $n \times m$.
- Each entry can be computed in $O(1)$ time.
- Time complexity$= O(mn)$.
- Space complexity$= O(mn)$.
  - Note that this requires a lot of space, say, $m = n = 100$K.
  - When we compare two very long sequences, space may be the limiting factor.
  - Can we solve the string alignment problem in linear space?
  - Yes, there is one in $O(m + n)$ space (Hirschberg's Algorithm, omitted).
- **Consider:** How to compute Longest Common Subsequence?

Local Alignment

- **The problem:** Give two strings $S[1..n]$ and $T[1..m]$, among all substrings of S and T, find a substring A of S and a B of T such that their global alignment has the highest score.
- The Brute-Force algorithm tries every possible pair of substrings and returns the pair (A, B) with highest score.
- The time needed by Brute-Force:
  - There are n^2 choices of A and m^2 choices of B.
  - The global alignment of A and B can be computed in $O(mn)$ time.
  - In total, time complexity$= O(m^3n^3)$.
- **Can we do better?**

Some Terminologies

- X is a suffix of $S[1..n]$ if $X = S[k..n]$ for some $k > 0$.
- X is a prefix of $S[1..n]$ if $X = S[1..k]$ for some $k \leqslant n$.
- E.g., consider $S[1..7] = ACCGATT$,
  - ACC is a prefix of S,
  - GATT is a suffix of S,
  - empty string is both prefix and suffix of S, and
  - so is the string S itself.

Smith-Waterman's Algorithm

- Let $V(i, j)$ be the maximum score of the global alignment of A and B over all suffixes A of $S[1..i]$ and all suffixes B of $T[1..j]$.
- Then, score of local alignment of S and T is $\max_{i,j} V(i, j)$, and the recurrence

$$V(i,j) = \max \begin{cases} 0 & i = 0 \text{ or } j = 0, \\ V(i-1,\ j-1) + \delta(S[i],\ T[j]) & i > 0,\ j > 0,\ match/mismatch, \\ V(i-1,\ j) + \delta(S[i],\ \_) & i > 0,\ j > 0,\ delete, \\ V(i,\ j-1) + \delta(\_,\ T[j]) & i > 0,\ j > 0,\ insert. \end{cases}$$

- E.g., consider $S[1..7] = ACCGATT$,
- Using dynamic programming to solve the problem,
  - we need to fill in a table of $O(mn)$ entries,
  - each entry can be computed in $O(1)$ time,
  - finally, find the the entry with maximum value. and
  - So, the time complexity is $O(mn)$.

Example

Take S =CTCATGC, T =ACAATCG for an example, assume that the score for match is 2, for insert, delete, and mismatch -1. Then Smith-Waterman's algorithm will fill a table

	_	C	T	C	A	T	G	C
_	0	0	0	0	0	0	0	0
A	0	0	0	0	2	1	0	0
C	0	2	1	2	1	1	0	2
A	0	0	1	1	4	3	2	1
A	0	0	0	0	3	3	2	1
T	0	0	2	1	2	5	4	3
C	0	2	1	4	3	4	4	6
G	0	1	1	3	3	3	6	5

- $V(i,j) = 6$.
- The Trace-Back tells us the optimal solution is $A = S[2..7]$ =CAATCG and $B = T[3..6]$ =CATG.
- The optimal global alignment for these two substrings A and B is CAATCG / C_AT_G,
- which achieves score 6.