COMP9318: Dynamic Programming and Edit Distance

Wei Wang

weiw AT cse.unsw.edu.au

School of Computer Science & Engineering
University of New South Wales

March 26, 2009

Acknowledgement

Adapted from at Prof. Pekka Kilpeläinen's slides at
http://www.cs.uku.fi/~kilpelai/BSA05/.

Inexact Matching

> *Today, the most powerful method for inferring the biological function of a gene (or the protein that it encodes) is by sequence similarity searching on protein and DNA sequence databases.*
> *— W.R. Pearson, 1995*

Inexact or approximate matching and sequence comparison are central tools in computational molecular biology. Because of

- errors in molecular data
- trying to understand mutational evolution of the sequences

exact equality is a too rigid notion of "similarity"

Inexact Matching

Inexact matching and comparison use alignments of strings to recognize their similarities Computing alignments involves

- considering subsequences (instead of contiguous *substrings*), and
- solving optimization problems (to maximize sequence similarity), often applying **dynamic programming**

The Edit Distance Problem

- The most classic inexact matching problem solved by dynamic programming: the edit distance problem
- Common way to formalize the difference or distance of two strings S_1 and S_2: number of single-character edit operations needed to transform S_1 into S_2
- An edit transcript is a sequence of operations **I** (insert the next char of S_2), **D** (delete), **R** (replace by the next char of S_2), and **M** (match) that copy the next char of S_1 to S_2.

Example: A transcript to transform "`Vintner`" to "`writers`"

```
RIMDMDMMI
V intner
wri t ers
```

Definition of the Edit Distance

Definition: Edit Distance

The edit distance between strings S_1 and S_2 is the **minimum** number of operations I, D and R in any transcript that transforms S_1 into S_2

- Also known as the Levenshtein distance
- An edit transcript with the smallest number of operations I, D and R is an *optimal transcript*

Definition: Edit Distance Problem

Compute the edit distance (and an optimal transcript) for the given strings S_1 and S_2

- **Observation**: The roles of S_1 and S_2 are symmetric, since a deletion in one string corresponds to an insertion in the other.

Computing the Edit Distance

- The edit distance between strings $S_1[1 \ldots n]$ and $S_2[1 \ldots m]$ can be computed applying dynamic programming
- Define $D(i; j)$ to be the edit distance of prefixes $S_1[1 \ldots i]$ and $S_2[1 \ldots j]$
- $D(n, m)$ **is** the edit distance of S_1 and S_2
- Dynamic programming computes $D(n, m)$ by computing $D(i, j)$ for all $i \leq n$ and $j \leq m$

Components of Dynamic Programming

Dynamic programming (of the edit distance) has three essential components:

- **Recurrence relation**
  - How is $D(i, j)$ determined from values $D(i', j')$ with smaller i' and j'?
- **Tabular computation**
  - How to memorize computed values, to avoid computing them over and over again?
- **Traceback**
  - How to find an optimal edit transcript?

The Recurrence Relation

How to determine the $D(i, j)$ values?

- **Base**

e.g., $D(i, 0)$

How to edit $S_1[1, i]$ to $S_2[1, 0]$ (i.e., empty string)?

- With i deletions!
- i.e., $D(i, 0) = i$, $\forall i (0 \leq i \leq n)$.

Similarly, $D(0, j) = j$

- insert $S_2[1], S_2[2], \ldots, S_2[j]$ into empty string

- **Inductive Case** for $i, j > 0$:

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 & //D \\ D(i, j-1) + 1 & //I \\ D(i-1, j-1) + \delta(i, j) & //M, R \end{cases}$$

where $\delta(i, j) = (S_1[i] == S_2[j])\ ?\ 0\ :\ 1$

Explaining the Recurrence Relation

A transcript that transforms $S_1[1 \ldots i]$ to $S_2[1 \ldots j]$ either

- transforms $S_1[1 \ldots i-1]$ to $S_2[1 \ldots j]$ and deletes $S_1[i] \rightarrow //D$
- transforms $S_1[1 \ldots i]$ to $S_2[1 \ldots j-1]$ and inserts $S_2[j] \rightarrow //I$
- transforms $S_1[1 \ldots i-1]$ to $S_2[1 \ldots j-1]$ and then matches or replaces $S_1[i]$ by $S_2[j] \rightarrow //M, R$

An optimal transcript is one that minimizes over the three possibilities

Note

More than one of the above cases may give the *same* minimum $\Rightarrow$ there may be many cooptimal transcripts

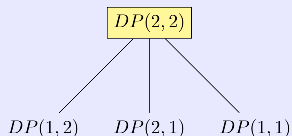
Tabular Computation of Edit Distance

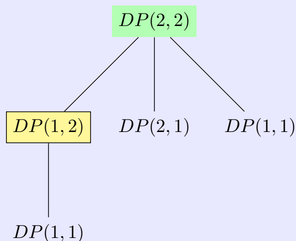
- It is **easy** to implement recurrences for $D(i,j)$ as a recursive procedure
  - as long as the programming language supports recursion
  - but this implementation will be extremely slow and not scalable at all.
- The problem is that a recursive procedure for $D(i,j)$ computes the same values *exponentially* many times
- But there are only $(n+1) \times (m+1)$ different $D(i,j)$ values $(0 \leq i \leq n, 0 \leq j \leq m)$
  - Compute them in a suitable order (*bottom-up*), and store them in an array so that each value is computed only once

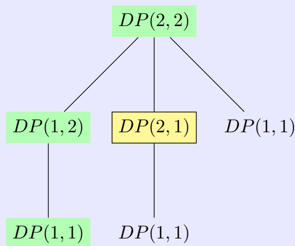
Naive Recursive Implementation

We know that $D(i,0) = i$, $(0 \le i \le n)$ and $D(0,j) = j$, $(0 \le j \le n)$.

		a	b
	0	1	2
c	1		
b	2		







Legend

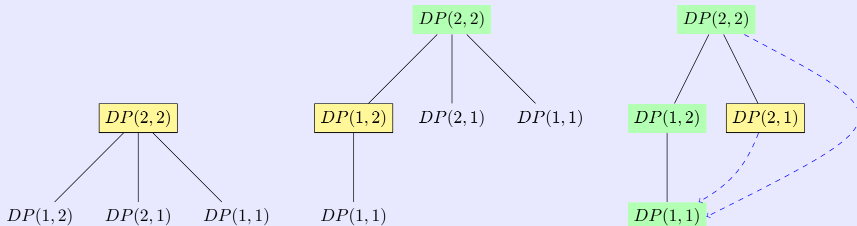
vistied nodes

current node

Memorization

A simple yet effective improvement is to use the memorization technique:

- Before calculating $D(x, y)$ recursively, first check if it has already been computed and thus stored in a buffer B.
- After calculating the current $D(x, y)$ value, store it in the buffer B.

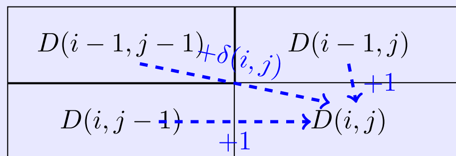


Bottom-up Computation

1. We know that $D(i,0) = i$, $(0 \leq i \leq n)$ and $D(0,j) = j$, $(0 \leq j \leq n)$.

		w	r	i	t	e	r	s
	0	1	2	3	4	5	6	7
V	1							
i	2							
n	3							
t	4							
n	5							
e	6							
r	7							

2. Then compute the remaining cells:



Bottom-up Computation

3. Find an appropriate "schedule" to compute the values in the cells (such that no recursion is needed)
   - Use either row-wise or column-wise order

		w	r	i	t	e	r	s
	0	1	2	3	4	5	6	7
V	1	1	2	3	4	5	6	7
i	2	2	2	**???**				
n	3							
t	4							
n	5							
e	6							
r	7							

for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to m **do**
        Compute $D(i, j)$ according to the recurrence
    end for
end for

Complexity of the Tabulation

- Each of the $n \cdot m$ cells is filled in constant time.
- Therefore, the edit distance $D(n, m)$ of strings $S_1[1 \dots n]$ and $S_2[1 \dots m]$ can be computed in $O(n \cdot m)$ time.
- The naive dynamic programming algorithm needs $O(n \cdot m)$ space, but that can be optimized to $O(\min(m, n))$ space.
  - Observation: only need to keep the *current* and the *previous* row/column.

Real Examples

Algorithm	S_1, S_2	**Time (Sec)**
naive	'Sundayxx', 'Saturdayxx'	14.0
memorization	'Sundayxx' $\times$ 30, 'Saturdayxx' $\times$ 30	11.0
dynamic programming	'Sundayxx' $\times$ 30, 'Saturdayxx' $\times$ 30	0.7

A Tricky Example

1. edit_distance("abcd", "cdab") = ?
2. edit_distance("ef, "fe") = ?
3. edit_distance("abcdef", "cdabfe") = ?