Edit distance between two strings

RIMRDMDMMI
v intner
wr t ers

Editing operations: Insertion (I), Deletion (D), Replacement (R).
Nonoperation: Match (M)

Edit transcript, of two strings S1 and S2, is a string over the alphabet I, D, R, M that describes a transformation of S1 to S2.

Edit distance between two strings is defined as the minimum number of edit operations needed to transform the first string into the second.

So, here is another optimization problem! How to solve?
Brute-force approach

- The idea: enumerate all possible edit transcripts and find the one that contains minimal number of edit operations. For example,

```
    vintner  vintner  vintner
    writers  writers  writers
```

- The Correctness: We have shown a few different edit transcripts that transform “vintner” to “writers”. As easily seen, at the worst-case, the length of an edit transcript of strings $S_1$ and $S_2$ can be $|s_1| + |s_2|$, namely, delete string $S_1$ and insert string $s_2$. So we only need to enumerate all strings over the alphabet I, D, R, M, of length at most $|s_1| + |s_2|$. There are at most $4^{|s_1| + |s_2|}$ strings. (A tighter upper bound is $4^\max(|s_1|, |s_2|)$)

- The Performance: extremely slow.
- Can we do better?

Dynamic Programming

Dynamic programming is an algorithm design paradigm for solving optimization problems

It is applicable when a problem is divisible, and an optimal solution to the problem is also optimal at subproblems.

Dynamic programming approach has three essential components:
- Recurrence relation
- Tabular computation
- Traceback.
Dynamic Programming: The recurrence relation

Definition For two strings $S_1$ and $S_2$, $D(i,j)$ is defined to be the edit distance of substrings $S_1[1..i]$ and $S_2[1..j]$.

If $S_1$ has $n$ letters and $S_2$ has $m$ letters, then the edit distance between $S_1$ and $S_2$ is the value $D(n,m)$.

In general, there exist the following recursive relationship.

$$D(i,0) = i$$
$$D(0,j) = j$$
$$D(i,j) = \min[D(i-1,j) + 1, D(i,j-1) + 1, D(i-1,j-1) + t(i,j)]$$

where $t(i,j) = 1$ if $S_1(i) \neq S_2(j)$
$= 0$ if $S_1(i) = S_2(j)$.

Correctness of the general recurrence

Lemma 1 The value of $D(i,j)$ must be $D(i,j-1) + 1$, $D(i-1,j) + 1$, or $D(i-1,j-1) + t(i,j)$. There are no other possibilities.

If $S_1$ has $n$ letters and $S_2$ has $m$ letters, then the edit distance between $S_1$ and $S_2$ is the value $D(n,m)$.

In general, there exist the following recursive relationship.

Lemma 2 $D(i,j) \leq \min[D(i-1,j) + 1, D(i,j-1), D(i-1,j-1) + t(i,j)]$
Dynamic Programming: Tabular computing

Top-down recursive approach

\[
\text{compEditDistance}(m, n) \\
1. \text{if } (n = 0 \text{ and } m > 0) \text{ then return } m \\
2. \text{ else if } (m = 0 \text{ and } n > 0) \text{ then return } n \\
3. \text{ if } (S1[m] = S2[n]) \\
4. \text{ then } \text{inc} \leftarrow 0 \\
5. \text{ else } \text{inc} \leftarrow 1 \\
6. \text{tmp0} \leftarrow \text{compEditDistance}(m-1, n-1) + \text{inc} \\
7. \text{tmp1} \leftarrow \text{compEditDistance}(m, n-1) + 1 \\
8. \text{tmp2} \leftarrow \text{compEditDistance}(m-1, n) + 1 \\
9. \text{ if } (\text{tmp1} > \text{tmp2}) \\
10. \text{ then } \text{Min} \leftarrow \text{tmp2} \\
11. \text{ if } (\text{Min} > \text{tmp0}) \\
12. \text{ then } \text{Min} \leftarrow \text{tmp0} \\
13. \text{return } \text{Min}
\]

Number of recursive calls in computing \(D(n, m)\) is \(O(3^{\max(n,m)})\), and subproblems are called many different times.
Dynamic Programming: Tabular computing

Bottom-up approach

\[ S_{i,j} = \min \{ \begin{array}{c} S_{i-1,j-1} + t(i,j) + 1 \\ S_{i-1,j} + 1 \\ S_{i,j-1} + 1 \end{array} \} \]

- Diagonal step indicates a match or replacement, at \( S_{i,i} \) and \( S_{j,j} \).
- Horizontal step indicates an insertion in \( S_2 \) or deletion in \( S_1 \).
- Vertical step indicates an insertion in \( S_1 \) or deletion in \( S_2 \).

Running time = \( O(nm) \)
Dynamic Programming: Traceback

\[
\begin{array}{cccccccc}
& w & r & i & t & e & r & s \\
v & 0 & 2 & 3 & 4 & 5 & 6 & 7 \\
i & 1 & 2 & 2 & 3 & 3 & 4 & 5 \\
n & 2 & 2 & 2 & 3 & 4 & 5 & 6 \\
t & 3 & 3 & 3 & 4 & 5 & 6 & 6 \\
e & 4 & 4 & 4 & 5 & 6 & 6 & 6 \\
r & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\
\end{array}
\]

writt-ers  writ-ers  writ-ers
vintner-  vintner-  -vintner-

Running time = \(O(n+m)\).

Application in bioinformatics

Evolutionary distance between amino acids

|     | A | C | D | E | F | G | H | I | K | L | M | N | P | Q | R | S | T | V | W | Y |
| **C** |  0 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| **G** |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **H** |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| **I** |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| **K** |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| **L** |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| **M** |  6 |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 |
| **N** |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 |
| **P** |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 |
| **Q** |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 |
| **R** | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 |
| **S** | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 | 10 |
| **T** | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 |
| **V** | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |
| **W** | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |
| **Y** | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |

CSC320, F03, Lect4, Liao
Application in bioinformatics

Identify homologous genes based on sequence similarity

```bash
>gi|7495582|gi|8464528 acetate kinase - Helicobacter pylori (strain 26695)

Score = 35.8 bits (81), expect = 0.10
Identities = 21/51 (41%), Positives = 29/51 (56%), Gaps = 2/51 (3%)

Query: 1 VLVQGQGSSLKFAIDAMGQELSWGARCY-GLREDKIEHEEKENQE 49
Subject: 3 ILVLNG0003FFKLFSEHMKFLHASEIES10GEKIFKHEMVQDE 53
```

Dynamics Programming: Memoization

To compute the length of the longest common subsequence of two strings

\[
\text{LCS\_Length}(\text{S1}, \text{S2})
\]

1. return subproblem(0,0)

Subproblem(i,j)

1. if \((\text{S1}[i] = \text{S2}[j])\) then
2. \quad return \(1 + \text{LCS\_Length}(i+1, j+1)\)
3. else
4. \quad if \((\text{S1}[i] = \text{S2}[j])\) then
5. \quad \quad return \(1 + \text{LCS\_Length}(i+1, j+1)\)
6. else
7. \quad return \(\max(\text{LCS\_Length}(i+1, j), \text{LCS\_Length}(i, j+1))\)
Dynamics Programming: Memoization

Memoization technique stores the results of subproblems and looks them up when needed.

Memoized_LCS_Length(S1, S2)
1.  m = length(S1)
2.  n = length(S2)
3.  for i = 0 to m  // initialize tabular computing
4.     for j = 0 to n
5.         L[i][j] = -1  // -1 to denote unfilled cell in the table
6.  return subproblem(0,0)

Subproblem(i,j)
1.  if L[i][j] >= 0 then  // this subproblem already solved
2.      return L[i][j]  // loop up the result and return
3.  if (S1[i] = '0' or S2[j] = '0')
4.      then L[i][j] = 0
5.  else
6.      if (S1[i] == S2[j]) then
7.         L[i][j] = 1 + Subproblem(i+1,j+1)
8.      else
9.         L[i][j] = max{ Subproblem(i+1,j), Subproblem(i,j+1) }
10.  return L[i][j]

Key learning

- Precondition to use dynamic programming:
  - Problem can be divided to subproblems
  - An optimal solution to the problem must also be optimal at each subproblem. (recurrence relations)

- Bottom-up or top-down
  - Top-down leads to a recursive algorithm: solve a large problem assuming we know solutions for smaller problems. Memoization.

- Comparison to greedy algorithms.
  - Dynamic programming is rigorous, i.e., it exhausts all possible options and finds the optimal ones.
  - Greedy algorithms use approximation, do not guarantee to find the optimal solutions.