Lecture 14
Dynamic Programming and Approximate String Matching

Brute-force approach

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

RIMIMR M M M I M M M M I M M M I M M M I M M M M I M M M M I
v in t n e r v in t n e r v in t n e r v in t n e r v in t n e r v in t n e r
w r i t e r s w r i t e r s w r i t e r s w r i t e r s w r i t e r s w r i t e r s

- The Correctness: We have shown a few different edit transcripts that transform "winner" to "writer". As easily seen, at the worst-case, the length of a edit transcript of strings S1 and S2, can be |S1| + |S2|, namely, delete string S1 and insert string S2. So we only need to enumerate all strings over the alphabet I, D, R, M, of length at most |S1| + |S2|. There are at most 4^(m+n) strings. (A tighter upper bound is 4 max(|S1|,|S2|))

- 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_i and S_j, D(i,j) is defined to be the edit distance of substrings S_i[1..i] and S_j[1..j].

If S_i has n letters and S_j has m letters, then the edit distance between S_i and S_j 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_i(i) = S_j(i)
0 if S_i(i) ≠ S_j(i)

Correctness of the general recurrence

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

If S_i has n letters and S_j has m letters, then the edit distance between S_i and S_j is the value D(n,m).

In general, there exist the following recursive relationship.

D(i,j) ≤ min [D(i-1,j) + 1, D(i, j-1) + 1, D(i-1, j-1) + t(i,j)]
Top-down recursive approach

```plaintext
cmpEditDistance(m, n)
1. if (n = 0 and m > 0) then return m
2. else if (m = 0 and n > 0) then return n
3. if (S1[i] = S2[j])
   then inc = 0
4. else inc = 1
5. inc = inc + 1
6. tmp = cmpEditDistance(m+1, n+1) + inc
7. tmp1 = cmpEditDistance(m+1, n) + 1
8. tmp2 = cmpEditDistance(m, n+1) + 1
9. if (tmp > tmp1)
10. then Min = tmp1
11. if (Min > tmp2)
12. then Min = tmp2
13. return Min
```

Bottom-up approach

```plaintext
D[i][j] = D[i-1][j-1] + 1
D[i][j] = D[i-1][j] + 1
D[i][j] = D[i][j-1] + 1

• Diagonal step indicates a match or replacement, at S1[i] and S2[j].
• Horizontal step indicates an insertion in S1 or deletion in S2.
• Vertical step indicates an insertion in S2 or deletion in S1.
```

Dynamic Programming: Tabular computing

```
D[i][j] = D[i-1][j-1] + 1
D[i][j] = D[i-1][j] + 1
D[i][j] = D[i][j-1] + 1

• Diagonal step indicates a match or replacement, at S1[i] and S2[j].
• Horizontal step indicates an insertion in S1 or deletion in S2.
• Vertical step indicates an insertion in S2 or deletion in S1.
```

Dynamic Programming: Traceback

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

• First row and first column are initialized.
• To fill a cell, only need to know its left, upper, and left-upper neighbors.
• Each cell is computed only once.
• When a cell is filled, also keep pointers to its predecessors.
• Entire table can be filled in one row at a time.
• Doesn’t look familiar? It’s induction.
```

Application in bioinformatics

```
Evolutionary distance between amino acids
```

```
writers  wri-t-ers  wri-t-ers
vintner- v-intner- v-intner-
```

```
Evolutionary distance between amino acids
```

```
writers  wri-t-ers  wri-t-ers
vintner- v-intner- v-intner-
```

Running time = O(n+m).

```
Running time = O(nm).
```

```
Running time = O(nm).
```

```
Running time = O(nm).
```

```
Running time = O(nm).
```
Application in bioinformatics

Identify homologous genes based on sequence similarity

```latex
\begin{align*}
\text{LCS\_Length}(S1, S2) & \\
\text{return } \text{subproblem}(0,0) \\
\text{Subproblem}(i,j) & \\
\text{if } (S1[i] = \text{'}0\text{'} \text{ or } S2[j] = \text{'}0\text{'}) & \\
\text{return } 0 \\
\text{else} & \\
\text{if } (S1[i] = S2[j]) & \\
\text{return } 1 + \text{Subproblem}(i+1, j+1) \\
\text{else} & \\
\text{return } \max(\text{Subproblem}(i+1, j), \text{Subproblem}(i, j+1))
\end{align*}
```

Dynamics Programming: Memoization

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

LCS\_Length($S_1$, $S_2$)

1. return subproblem(0,0)

Subproblem($i,j$)

1. if ($S_1[i] = \text{'}0\text{'}$ or $S_2[j] = \text{'}0\text{'}$) 
2. then return 0
3. else
4. if ($S_1[i] = S_2[j]$) 
5. then return 1 + LCS\_Length($i+1$, $j+1$)
6. else
7. return max(LCS\_Length($i+1$, $j$), LCS\_Length($i$, $j+1$))

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

```latex
\begin{align*}
\text{Memoized\_LCS\_Length($S_1$, $S_2$)} & \\
\text{M = length($S_1$)} \\
\text{for } i = 0 \text{ to } M \text{ \{initialize tabular computing\} } \\
\text{for } j = 0 \text{ to } N \text{ \{/ to denote unfilled cell in the table\} } \\
\text{return subproblem(0,0)} \\
\text{Subproblem($i,j$)} & \\
\text{if L[$i$,$j$] $\geq$ 0 then \{the subproblem already solved\} } \\
\text{return L[$i$,$j$] \{loop up the result and return\} } \\
\text{if ($S_1[i] = \text{'}0\text{'}$ or $S_2[j] = \text{'}0\text{'}$) } \\
\text{return 0} \\
\text{else} & \\
\text{if ($S_1[i] = S_2[j]$)} \\
\text{L[$i$,$j$] = 1 + Subproblem($i+1$, $j+1$)} \\
\text{else} & \\
\text{L[$i$,$j$] = max(Subproblem($i+1$, $j$), Subproblem($i$, $j+1$))} \\
\text{return L[$i$,$j$]} 
\end{align*}
```

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