Advanced Algorithmics (4AP)
Dynamic programming

Jaak Vilo
2009 Spring

Dynamic programming

- Avoid calculating repeating subproblems
- fib(1)=fib(0)=1;
- fib(n) = fib(n-1)+fib(n-2)

- Although natural to encode (and a useful task for novice programmers to learn about recursion) recursively, this is inefficient.

Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.
- We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.
Matrix multiplication

\[ \begin{align*}
\text{for } i &= 1 \ldots n \\
\text{for } j &= 1 \ldots k \\
\text{let } c_{ij} &= \sum_{x=1}^{m} a_{ix} b_{xj} \\
\end{align*} \]

\[ O(nmk) \]

\[
\begin{array}{ccc}
\text{A} & \times & \text{B} \\
\hline
\text{m} & \text{m} & \text{n} \\
\text{n} & \text{k} & \text{k} \\
\end{array}
\]

\[ \text{C} \]

**6.5 Chain matrix multiplication**

The **matrix-chain multiplication problem** can be stated as follows: given a chain \( <A_1, A_2, \ldots, A_n> \) of \( n \) matrices:

- matrix \( A_i \) has dimension \( p_{i-1} \times p_i \)
- fully parenthesize the product \( A_1 A_2 \ldots A_n \) in a way that minimizes the number of scalar multiplications.
\[ A_1 A_2 A_3 A_4 \]

- \( \{A_1 (A_2 (A_3 A_4)) \}, \)
- \( \{ (A_1 (A_2 A_3)) A_4 \}, \)
- \( \{ (A_1 A_2) (A_3 A_4) \}, \)
- \( \{ (A_1 A_2) (A_3) A_4 \}, \)
- \( \{ ((A_1 A_2) A_3) A_4 \}. \)

- Denote the number of alternative parenthesizations of a sequence of \( n \) matrices by \( P(n) \).
- Since we can split a sequence of \( n \) matrices between the \( k \)th and \( (k + 1) \)st matrices for any \( k = 1, 2, \ldots, n - 1 \) and then parenthesize the two resulting subsequences independently, we obtain the recurrence
  \[
P(n) = \frac{1}{n + 1} \binom{2n}{n} = \Omega\left( \frac{4^n}{n^{3/2}} \right).
  \]

Let’s crack the problem

\[ A_{i..j} = A_i A_{i+1} \cdots A_j \]

- Optimal parenthesis of \( A_1 A_2 \cdots A_n \) splits at some \( k, k+1 \).
- Optimal = \( A_{1..k} A_{k+1..n} \)
- \( T(A_{1..k}) = T(A_{1..k}) + T(A_{k+1..n}) + T(A_{1..k} A_{k+1..n}) \)
- \( T(A_{1..k}) \) must be optimal for \( A_1 A_2 \cdots A_k \)

Recursion

- \( m[i, j] \) - minimum number of scalar multiplications needed to compute the matrix \( A_{i..j} \)
- \( m[i, j] = 0 \)
- \( \text{cost}(A_{i..k} \cdot A_{k+1..j}) = p_{i..k} p_{k+1..j} \)
- \( m[i, j] = m[i, k] + m[k+1, j] + p_{i..k} p_{k+1..j} \).

This recursive equation assumes that we know the value of \( k \), which we don’t. There are only \( j - i \) possible values for \( k \), however, namely \( k = i + 1, \ldots, j - 1 \).

Since the optimal parenthesis must use one of these values for \( k \), we need only check them all to find the best. Thus, our recursive definition for the minimum cost of parenthesizing the product \( A_{i..k} A_{k+1..j} \) becomes

\[
m[i, j] = \min_{k = i+1}^{j-1} \left( m[i, k] + m[k+1, j] + p_{i..k} p_{k+1..j} \right)
  \quad \text{if } i < j.
\] (16.2)

To help us keep track of how to construct an optimal solution, let us define \( s[i, j] \) to be a value of \( k \) at which we can split the product \( A_{i..k} A_{k+1..j} \) to obtain an optimal parenthesisation. That is, \( s[i, j] \) equals a value \( k \) such that \( m[i, j] = m[i, k] + m[k+1, j] + p_{i..k} p_{k+1..j} \).
Recursion

- Checks all possibilities...
- But – there is only a few subproblems – choose $i, j$ s.t. $1 \leq i \leq j \leq n$ - $O(n^2)$
- A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of the applicability of dynamic programming.

```
// foreach length from 2 to n
// find minimal using subsolutions
```

```
 MATRIX-CHAIN-ORDER($p$)
  1 n = length($p$) - 1
  2 for $i = 1$ to $n$
  3 do $m[i][i] = 0$
  4 for $j = 2$ to $n$
  5 do for $i = 1$ to $n - j + 1$
  6 do $j = i + j - 1$
  7 $m[i][j] = \infty$
  8 for $k = i$ to $j - 1$
  9 do $q = m[i][k] + m[k+1][j] + p[0] \cdot p[k] \cdot p[j]$  
  10 if $q < m[i][j]$
  11 then $m[i][j] = q$
  12 $s[i][j] = k$
  13 return $m$ and $s$
```

```
Example
```

```
matrix dimensions:  
A1 30 X 35  
A2 35 X 15  
A3 15 X 5  
A4 5 X 10  
A5 10 X 20  
A6 20 X 25
```

- A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm. The loops are nested three deep, and each loop index ($l$, $i$, and $k$) takes on at most $n$ values.
- Time $O(n^3)$ => $O(n^3)$
- Space $\Theta(n^2)$

- Step 4 of the dynamic-programming paradigm is to construct an optimal solution from computed information.
- Use the table $s[1 .. n, 1 .. n]$ to determine the best way to multiply the matrices.
Multiply using S table

\[
\text{MATRIX-CHAIN-MULTIPLY}(A, s, i, j) \\
1 \text{ if } j > i \\
2 \quad \text{then } X = \text{MATRIX-CHAIN-MULTIPLY}(A, s, i, s[i, j]) \\
3 \quad Y = \text{MATRIX-CHAIN-MULTIPLY}(A, s, s[i, j]+1, j) \\
4 \quad \text{return } \text{MATRIX-MULTIPLY}(X, Y) \\
5 \text{else return } A_i \\
(\langle A_i A_2 A_j \rangle)(\langle A_4 A_5 A_6 \rangle)
\]

Elements of dynamic programming

- **Optimal substructure** within an optimal solution
- Overlapping subproblems
- Memoization

- A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and then stored in the table. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned. (tabulated)

  - This approach presupposes that the set of all possible subproblem parameters is known and that the relation between table positions and subproblems is established. Another approach is to memoize by using hashing with the subproblem parameters as keys.

Overlapping subproblems

\[
\begin{array}{ccccccc}
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
\end{array}
\]

Longest Common Subsequence (LCS)

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & & & & & \\
1 & & & & & \\
2 & & & & & \\
3 & & & & & \\
4 & & & & & \\
5 & & & & & \\
6 & & & & & \\
7 & & & & & \\
\end{array}
\]

Optimal triangulation

The problem is to find a triangulation that minimizes the sum of the weights of the triangles in the triangulation.

Two ways of triangulating a convex polygon. Every triangulation of this 7-sided polygon has 7 - 3 = 4 chords and divides the polygon into 7 - 2 = 5 triangles.
Parse trees. (a) The parse tree for the parenthesized product 
\((A_1(A_2A_3))(A_4(A_5A_6)))\) and for the triangulation of the 7-sided polygon
(b) The triangulation of the polygon with the parse tree overlaid. Each
matrix \(A_i\) corresponds to the side \(v_{i-1}v_i\) for \(i = 1, 2, \ldots, 6\).

Optimal triangulation

\[
q[i,j] = \begin{cases} 
  0 & \text{if } i = j, \\
  \min \{ q[i-1,j], q[i,j-1], q[i,j-2] + w(v_{i-1}, v_i) \} & \text{if } i < j.
\end{cases} \quad (16.7)
\]

Similarity

- How can we measure the similarity of two strings?
- When are the two things “almost” the same?

Edit distance (Levenshtein distance)

- Smallest nr of edit operations to convert one
  string into the other

<table>
<thead>
<tr>
<th>IND USTRY</th>
<th>IND USTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEREST</td>
<td>INTEREST</td>
</tr>
</tbody>
</table>

- Definition The edit distance \(D(A,B)\) between strings \(A\) and \(B\) is
  the minimal number of edit operations to change \(A\) into \(B\). Allowed
  edit operations are deletion of a single letter, insertion of a letter, or replacing one letter with another.
- Let \(A = a_1 a_2 \ldots a_m\) and \(B = b_1 b_2 \ldots b_n\).
- E1: Deletion \(a_i \rightarrow \epsilon\)
- E2: Insertion \(\epsilon \rightarrow b_i\)
- E3: Substitution \(a_i \rightarrow b_j\) (if \(a_i \neq b_j\))
- Other possible variants:
- E4: Transposition \(a_ia_{i+1} \rightarrow b_ib_{i+1}\) and \(a_ia_{i+1} \rightarrow b_{i+1}a_i\)
  [e.g. lecture \(\rightarrow\) letcure]
How can we calculate this?

\[
D(\alpha, \beta) = \min\left\{ \begin{align*}
1. & \ D(\alpha, \beta) \quad \text{if } a = b \\
2. & \ D(\alpha, \beta) + 1 \quad \text{if } a \neq b \\
3. & \ D(\alpha, \beta) + 1 \\
4. & \ D(\alpha, \beta) + 1
\end{align*} \right. \]

How can we calculate this efficiently?

\[
D(S,T) = \min\left\{ \begin{align*}
1. & \ D(S[1..n-1], T[1..m-1]) + (S[n]=T[m])? 0 : 1 \\
2. & \ D(S[1..n], T[1..m-1]) +1 \\
3. & \ D(S[1..n-1], T[1..m]) +1
\end{align*} \right. \]

\[
d(i,j) = \min\left\{ \begin{align*}
1. & \ d(i-1,j-1) + (S[i]=T[j])? 0 : 1 \\
2. & \ d(i, j-1) +1 \\
3. & \ d(i-1, j) +1
\end{align*} \right. \]

Define:

\[d(i,j) = D(S[1..i], T[1..j])\]

Recursion?

\[\text{sub fib(int x)}\]

\[
\text{if (x<3) return 1; else return fib(x-1)+fib(x-2)};\]

Recursion?

Algorithm Edit distance \(D(A,B)\) using Dynamic Programming (DP)

Input: \(A=a_1a_2...a_n\) \(B=b_1b_2...b_m\)

Output: Value \(d_{mn}\) in matrix \(d_{ij}\), \(O(n)\) \(O(m)\).

for \(i=0\) to \(m\) do \(d_{0i} = i\); 
for \(j=0\) to \(n\) do \(d_{0j} = j\); 
for \(i=1\) to \(n\) do 
  for \(j=1\) to \(m\) do 
    \(d_{ij} = \min\{ d_{i-1,j-1} + (\text{if } a_==b_j \text{ then } 0 \text{ else } 1), \)
    \(d_{i-1,j} + 1, \ d_{i,j-1} + 1 \}
return \(d_{mn}\)
Dynamic Programming

Edit distance is a metric

- It can be shown, that \( D(A,B) \) is a metric
  - \( D(A,B) \geq 0 \), \( D(A,B) = 0 \) iff \( A=B \)
  - \( D(A,B) = D(B,A) \)
  - \( D(A,C) \leq D(A,B) + D(B,C) \)

Alignment

Path of edit operations

- Optimal solution can be calculated afterwards
  - Quite typical in dynamic programming
    \[ d(i-1, j-1) \]
    \[ d(i-1, j) \]
    \[ d(i, j-1) \]
    \[ d(i, j) \]
  - Memorize sets \( \text{pred}[i,j] \) depending from where the \( d_{ij} \) was reached.
Three possible minimizing paths

- Add into pred\([i,j]\]
  - \((i-1,j-1)\) if \(d_{ij} = d_{i-1,j-1} + (\text{if } a_i==b_j \text{ then } 0 \text{ else } 1)\)
  - \((i-1,j)\) if \(d_{ij} = d_{i-1,j} + 1\)
  - \((i,j-1)\) if \(d_{ij} = d_{i,j-1} + 1\)

Multiple paths possible

- All paths are correct
- There can be many (how many?) paths

Space can be reduced

```
\begin{array}{cccccc}
  & a & b & a & c & b & c \\
A & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
   & 1 & 1 & 1 & 2 & 3 & 4 & 5 \\
   & 2 & 1 & 2 & 1 & 2 & 3 & 4 \\
   & 3 & 2 & 2 & 3 & 2 & 3 & 3 \\
   & 4 & 3 & 3 & 3 & 2 & 3 & 3 \\
   & 5 & 4 & 3 & 4 & 3 & 2 & 3 \\
B & \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
                             b & 1 & 1 & 1 & 2 & 3 & 4 & 5 \\
                             a & 2 & 1 & 2 & 1 & 2 & 3 & 4 \\
                             a & 3 & 2 & 2 & 3 & 2 & 3 & 3 \\
                             c & 4 & 3 & 3 & 3 & 2 & 3 & 3 \\
                             b & 5 & 4 & 3 & 4 & 3 & 2 & 3 \\
                             C[m] & \begin{bmatrix} 3 & \end{bmatrix} & \end{array}
```

\(C[0]=2\)

What are the other questions in edit distance calculations?

- Space complexity
- Time complexity
- Other ways to look at the algorithm(s)
- Applications
- More complex notions of similarity
- …

Calculation of \(D(A,B)\) in space \(\Theta(m)\)

```
Input: A=a_1a_2...a_m, B=b_1b_2...b_n (choose m<=n)
Output: \(d_{m,n}=D(A,B)\)
for \(i=0\) to \(m\) do \(C[i]=i\);
for \(j=1\) to \(n\) do
  \(C=\{0\}; C[0]=j;\)
  for \(i=1\) to \(m\) do
    \(d=\text{min}(C + (\text{if } a_i==b_j \text{ then } 0 \text{ else } 1), C[i-1] + 1, C[j] + 1)\)
    \(C[C[i]] = d;\)
  end
end
write \(C[m]\)
```

Time complexity is \(\Theta(mn)\) since \(C[0..m]\) is filled \(n\) times
Shortest path in the graph

Observations?

• Shortest path is close to the diagonal
  – If a short distance path exists

• Values along any diagonal can only increase
  (by at most 1)

Diagonal

Property of any diagonal: The values of matrix $d_{ij}$ can on any specific diagonal either increase by 1 or stay the same

Diagonal lemma

Lemma: For each $d_{ij}$, 1≤ i, j ≤ m holds: $d_{ij} = d_{i-1,j-1}$ or $d_{ij} = d_{i-1,j} + 1$. (notice that $d_{0,0}$ and $d_{m-1,m-1}$ are on the same diagonal)

Proof: Since $d_{ij}$ is an integer, show:

• $d_{ij} \leq d_{i-1,j-1} + 1$
• $d_{ij} \geq d_{i-1,j}$

From the definition of edit distance 1. holds since $d_{ij} \leq d_{i-1,j} + 1$

Induction on i+j:

– Basis is trivial when $i=0$ or $j=0$ (if we agree that $d_{-1,j} = d_{i-1,0}$)
– Induction step there are 3 possibilities –
  • On minimization the $d_{ij}$ is calculated from entry $d_{i-1,j}$ so $d_{ij} = d_{i-1,j}$
  • On minimization the $d_{ij}$ is calculated from entry $d_{i,j-1}$ so $d_{ij} = d_{i,j-1} + 1$
  • On minimization the $d_{ij}$ is calculated from entry $d_{i-1,j-1}$ Analogous to 2.
– Hence, $d_{i+1,j} \leq d_{ij}$
Transform the matrix into $f_{kp}$

- For each diagonal only show the position (row index) where the value is increased by 1.
- Also, one can restrict the matrix ($d_{ij}$) to only this part where $d_{ij} \leq d_{mn}$ since only those $d_{ij}$ can be on the shortest path.
- We'll use the matrix ($f_{kp}$) that represents the diagonals of $d_{ij}$
  - $f_{kp}$ is a row index $i$ from $d_{ij}$, such that on diagonal $k$ the value $p$ reaches row $i$ ($d_{ij}=p$ and $j-i=k$).
  - Initialization: $f_{kp,-1}$ and $f_{kp,-m}$ when $p \leq |k|-1$; $d_{mn}=p$, such that $f_{m,n}=m$

Calculating matrix ($f_{kp}$) by columns

- Assume the column $p-1$ has been calculated in ($f_{kp}$), and we want to calculate $f_{kp'}$ (the region of $d_{ij}=p$)
- On diagonal $k$ values $p$ reach at least the row $t=\max(f_{kp-1+1}, f_{k+1,p-1+1})$ if the diagonal $k$ reaches so far.
- If on row $t+1$ additionally $a_i = b_i$ on the same diagonal, then $d_{ij}$ cannot increase, and value $p$ reaches row $t+1$.
- Repeat previous step until $a_i \neq b_i$ on diagonal $k$.

Algorithm A(): calculate $f_{kp}$

A(k,p)
1. $t=\max(f_{kp-1+1}, f_{k+1,p-1+1})$
2. while $a_{t+k} = b_{t+k}$ do $t=t+1$
3. $f_{kp} = \text{if } t>m \text{ or } t+k>n \text{ then undefined else } t$

Example

- In original matrix ($d_{ij}$) there are 42 values, and in diagonal matrix ($f_{kp}$) 22 values (ext.)
- Notes: The values in ($f_{kp}$) are not comfortably organized. With appropriate index transformations it is possible to give the ($f_{kp}$) more regular shape.

```plaintext
Example (compare to prev. wc)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```

Example (compare to prev. wc)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Algorithm A(): calculate $f_{kp}$

A(k,p)
1. $t=\max(f_{kp-1+1}, f_{k+1,p-1+1})$
2. while $a_{t+k} = b_{t+k}$ do $t=t+1$
3. $f_{kp} = \text{if } t>m \text{ or } t+k>n \text{ then undefined else } t$

Example

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

$A_{1,2} ... t=\max(3,2,3) = 2$

$A_{3,2} \text{ and } A_{2,3}$

$A_{0,2} = 5$

$A_{1,2} = 1$
Algorithm: Diagonal method by columns

\[ p = -1 \]
\[ \text{while } f_{m,p} \neq m \]
\[ p = p + 1 \]
\[ \text{for } k = -p \text{ to } p \text{ do } \]
\[ // f_{kp} = A(k,p) \]
\[ t = \max\{f_{k-1,p+1}, f_{k-1,p}, f_{k+1,p-1}+1\} \]
\[ \text{while } a_{i+1} = b_{i+1} \text{ do } t = t + 1 \]
\[ f_{kp} = \text{if } t > m \text{ or } t + k > n \text{ then undefined else } t \]

- \( p \) can only occur on diagonals \(-p \leq k \leq p\).
- Method can be improved since \( k \) is often such that \( f_{kp} \) is undefined.
- We can decrease values of \( k \):
  - \(-m \leq k \leq n\) (diagonal numbers)
  - Let \( m \leq n \) and \( d_{ij} \) on diagonal \( k \).
    - if \(-m \leq k \leq 0\) then \( |k| \leq d_{ij} \leq m \)
    - if \( 1 \leq k \leq n \) then \( k \leq d_{ij} \leq k+m \)
    - Hence, \(-m \leq k \leq m\) if \( p \leq m \) and \( p-m \leq k \leq p \) if \( p \geq m \)

Some notes

- In applications small \( D(A,B) \) are most interesting.
- Can modify the algorithm by providing maximum \( t \).
- Hence, \( O(tm) \) - the smaller the \( t \), the faster the algorithm.
- Space can be reduced by keeping only previous column.
- How to output the shortest path?
  - Relatively simple, in time \( O(s) \), outputs a single path.

Extensions to basic edit distance

- New operations

- Variable costs

- ...

Transposition \((ab \rightarrow ba)\)

- **E4: Transposition**
  \[ a_i a_{i+1} \rightarrow b_j b_{j+1} \text{, s.t. } a_j = b_{j+1} \text{ and } a_{i+1} = b_j \]
  - (e.g.: lecture \(\rightarrow\) letcure)

\[ d(i,j) = \begin{cases} 
1. d(i-1,j-1) + (S[n]=T[m])? 0 : 1 \\
2. d(i,j-1) + 1 \\
3. d(i-1,j) + 1 \\
4. d(i-2,j-2) + (\text{if } S[i-1,i] = T[j,j-1] \text{ then } 1 \text{ else } \infty) 
\end{cases} \]
The edit distance algorithm can be changed easily.

Space efficiency can also be achieved using 2 last columns, hence still O(m)

Diagonal method can be modified, since the diagonal lemma holds.

Algorithms can be modified in a relatively straightforward manner.

• The edit distance algorithm can be changed easily
• Space efficiency can also be achieved using 2 last columns, hence still O(m)
• Diagonal method can be modified, since the diagonal lemma holds.
• Algorithms can be modified in a relatively straightforward manner.

Longest common subsequences

• Definition. String C=c₁c₂...cᵣ is a subsequence (alamjada) of A=a₁a₂...aₘ if by removing from A null or more characters, one can get C.

• String C=c₁c₂...cᵣ is the longest common subsequence, LCS (pikim ühine alamjada) of A=a₁a₂...aₘ and B=b₁b₂...bₙ, if C is the longest string that is both the subsequence of A and B.

• C=LCS(A,B)
• The length of LCS(A,B), |C|, can be used as the similarity measure for A and B.

• LCS(A,B) can be calculated similarly to edit distance |LCS(A,B)|=(|A|+|B|−D′(A,B))/2

• Let D′(A,B) the edit distance where the only allowed operations are insertion and deletion (no replace).

• Theorem

  a) |LCS(A,B)|= (|A|+|B|−D′(A,B))/2
  b) Let's have two sets D′(A,B) with optimal nr of changes:

     1. aᵢ→ε, aᵢ₊₁→ε, ..., aᵢ₊ₚ→ε deletions from A and
     2. ε→bⱼ₁, ε→bⱼ₂, ..., ε→bⱼᵣ insertions into B.

     Then LCS(A,B)=C can be constructed such that, C is A after deletions of 1. and C is B after deletion of all insertions 2. (insertions in 2. are reversely deletions from B).

• Example. LCS(england, inglismaa)=ngla.

• Unix command `diff` compares files row by row and searches the deviations from the LCS of the two files.

  |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

  • |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2
  • Let D′(A,B) the edit distance where the only allowed operations are insertion and deletion (no replace).
  • Theorem

    a) |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2
    b) Let's have two sets D′(A,B) with optimal nr of changes:

      1. aᵢ→ε, aᵢ₊₁→ε, ..., aᵢ₊ₚ→ε deletions from A and
      2. ε→bⱼ₁, ε→bⱼ₂, ..., ε→bⱼᵣ insertions into B.

      Then LCS(A,B)=C can be constructed such that, C is A after deletions of 1. and C is B after deletion of all insertions 2. (insertions in 2. are reversely deletions from B).

    • Proof b)

      a) According to construction, C is uniquely defined
      b) |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

      • According to construction, C is uniquely defined
      • |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

      • According to b) |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

      • According to construction, C is uniquely defined
      • |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

      • According to b) |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

      • According to construction, C is uniquely defined
      • |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

      • According to construction, C is uniquely defined
      • |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

      • According to construction, C is uniquely defined
      • |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

      • According to construction, C is uniquely defined
      • |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

      • According to construction, C is uniquely defined
      • |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2

      • According to construction, C is uniquely defined
      • |LCS(A,B)| = ( |A| + |B| - D′(A,B) )/2
Generalized edit distance

- Use more operations E1...En, and to provide different costs to each.

**Definition.** Let x, y ∈ Σ*. Then every x → y is an edit operation. Edit operation replaces x by y.
  - If A=uv then after the operation, A=uyv
- We note by w(x → y) the cost or weight of the operation.
- Cost may depend on x and/or y. But we assume w(x → y) ≥ 0.

Applications of generalized edit distance

- Historic documents, names
- Human language and dialects
- Transliteration rules from one alphabet to another
  e.g. Tõugu => Tyugu (via Russian)
- ...

Examples

- If operations can only be applied in parallel, i.e. the part already changed cannot be modified again, then we can use the dynamic programming.
- Otherwise it is an algorithmically unsolvable problem, since question - can A be transformed into B using operations of G, is unsolvable.
- The diagonal method in general may not be applicable.
- But, since each diversion from diagonal, the cost slightly increases, one can stay within the narrow region around the diagonal.
näituseks – näiteks
Ahwrika - Aafrika
weikese - väikese
materjaali - materjali

tuseks -> teks
a -> aa, hw -> f
w -> v, e -> ä
aa -> a

“kavalam” otsimine
Dush, dušš, dushš?
Gorbatšov, Gorbatšov, Горбачов,
Gorbachev
režim, rezhiim, riim

How?

• Apply Aho-Corasick to match for all possible edit operations
• Use minimum over all possible such operations and costs
• Implementation: Reina Käärik

Possible problems/tasks

• Manually create sensible lists of operations
  – For English, Russian, etc...
  – Old language,
• Improve the speed of the algorithm (testing)

• Train for automatic extraction of edit operations and respective costs from examples of matching words…