Algorithm Theory
11 Dynamic Programming

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08
Outline

‣ General approach, differences to a recursive approach

‣ Basic example: Computation of the Fibonacci numbers
Method of Dynamic Programming

- **Recursive approach**
  - Solve a problem by solving several smaller analogous subproblems of the same type.
  - Then combine these solutions to generate a solution to the original problem.

- **Drawback:** Repeated computation of solutions

- **Dynamic programming**
  - Once a subproblem has been solved, store its solution in a table so that it can be retrieved later by simple table lookup
Example: Fibonacci Numbers

\[ f(0) = 0 \]
\[ f(1) = 1 \]
\[ f(n) = f(n - 1) + f(n - 2), \text{ falls } n \geq 2 \]

Remark:

\[ f(n) = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}} \quad \phi = \frac{1 + \sqrt{5}}{2} \]

Straightforward Implementation:

```plaintext
procedure fib (n : integer) : integer
if (n == 0) or (n ==1)
    then return n
else return fib(n - 1) + fib(n - 2)
```

Example: Fibonacci Numbers

Recursion tree:

\[ T(n) \geq f(n) \geq 2^n \]
Dynamic Programming

- **Approach:**
  1. Recursively define problem $P$.
  2. Determine a set $T$ consisting of all subproblems that have to be solved during the computation of a solution of $P$.
  3. Find an order $T_0, ..., T_k$ of the subproblems in $T$ such that during the computation of a solution to $T_i$ only subproblems $T_j$ with $j < i$ arise.
  4. Solve $T_0, ..., T_k$ in this order and store the solutions.
Example: Fibonacci Numbers

1. Recursive definition of the Fibonacci numbers, based on the standard definition

2. \( T = \{ f(0), \ldots, f(n-1) \} \)

3. \( T_i = f(i), \quad i = 0, \ldots, n - 1 \)

4. Computation of \( \text{fib}(i) \), for \( i \geq 2 \), only requires the results of the last two subproblems \( \text{fib}(i-1) \) and \( \text{fib}(i-2) \).
Example: Fibonacci Numbers

Computation by dynamic programming, version 1

```plaintext
procedure fib(n : integer) : integer
1  \( f_0 := 0; f_1 := 1 \)
2  for \( k := 2 \) to \( n \) do
3      \( f_k := f_{k-1} + f_{k-2} \)
4  return \( f_n \)
```
Example: Fibonacci Numbers

Computation by dynamic programming, version 2

procedure fib (n : integer) : integer

1 \( f_{\text{next-to-last}} := 0; \ f_{\text{last}} := 1 \)

2 for \( k := 2 \) to \( n \) do

3 \( f_{\text{current}} := f_{\text{last}} + \ f_{\text{next-to-last}} \)

4 \( f_{\text{next-to-last}} := f_{\text{last}} \)

5 \( f_{\text{last}} := f_{\text{current}} \)

6 if \( n \leq 1 \) then return \( n \) else return \( f_{\text{current}} \);

Linear running time, constant space requirement!
Computation of the Fibonacci Numbers using Memoization

Compute each number exactly once, store it in an array $F[0...n]$:  

```
procedure fib (n : integer) : integer
1  $F[0] := 0$;  $F[1] := 1$;
2  for $i := 2$ to $n$ do
3       $F[i] := \infty$;
4  return lookupfib(n)
```

The procedure `lookupfib` is defined as follows:

```
procedure lookupfib(k : integer) : integer
1  if $F[k] < \infty$
2      then return $F[k]$
3  else $F[k] := lookupfib(k - 1) + lookupfib(k - 2)$;
4  return $F[k]$
```
Optimal Substructure

Dynamic programming is typically applied to

*optimization problems.*

An optimal solution to the original problem contains

*optimal solutions to smaller subproblems.*
Matrix Chain Multiplications

Given: sequence (chain) \( \langle A_1, A_2, ..., A_n \rangle \) of matrices

Goal: compute the product \( A_1 \cdot A_2 \cdot .... \cdot A_n \)

Problem: Parenthesize the product in a way that minimizes the number of scalar multiplications.

Definition: A product of matrices is fully parenthesized, if it is either a single matrix or the product of two fully parenthesized matrix surrounded by parentheses.
Examples of Fully Parenthesized Matrix Products

All possible fully parenthesized matrix products of the chain \( \langle A_1, A_2, A_3, A_4 \rangle \) are:

\[
\begin{align*}
(A_1 (A_2 (A_3 A_4))) \\
(A_1 ((A_2 A_3) A_4)) \\
((A_1 A_2)(A_3 A_4)) \\
((A_1 (A_2 A_3)) A_4) \\
((((A_1 A_2) A_3) A_4))
\end{align*}
\]
Number of Different Parenthesizations

Different parenthesizations corresponds to different trees:
Number of Different Parenthesizations

\[ P(n) \text{ be the number of alternative parenthesizations of the product } A_1 \ldots A_k A_{k+1} \ldots A_n \]

\[ P(1) = 1 \]

\[ P(n) = \sum_{k=1}^{n-1} P(k) P(n - k) \quad \text{for } n \geq 2 \]

\[ P(n + 1) = \frac{1}{n + 1} \binom{2n}{n} = \frac{4^n}{n \sqrt{\pi n}} + O \left( \frac{4^n}{\sqrt{n^5}} \right) \]

\[ P(n + 1) = C_n \quad n\text{-th Catalan number} \]

Determining the optimal parenthesization by exhaustive search is not reasonable.
Multiplication of two Matrices

\[ A = (a_{ij})_{p \times q}, \quad B = (b_{ij})_{q \times r}, \quad A \times B = C = (c_{ij})_{p \times r}. \]

\[ c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj} \]

Algorithm \textit{Matrix-Mult}

Input: \((p \times q)\) matrix \(A\), \((q \times r)\) matrix \(B\)

Output: \((p \times r)\) matrix \(C = A \cdot B\)

1. \textbf{for} \(i := 1\) \textbf{to} \(p\) \textbf{do}
2. \quad \textbf{for} \(j := 1\) \textbf{to} \(r\) \textbf{do}
3. \quad \quad \(C[i, j] := 0\)
4. \quad \textbf{for} \(k := 1\) \textbf{to} \(q\) \textbf{do}
5. \quad \quad \(C[i, j] := C[i, j] + A[i, k] \cdot B[k, j]\)

Number of multiplications and additions: \(p \cdot q \cdot r\)

Using this algorithm, multiplying two \((n \times n)\) matrices requires \(n^3\) multiplications.

Remark: This can be also done using \(O(n^{2.376})\) multiplications.
Matrix Chain Multiplication: Example

- Computation of the product $A_1A_2A_3$, where
- $A_1: 10 \times 100$ matrix
- $A_2: 100 \times 5$ matrix
- $A_3: 5 \times 50$ matrix
- Parenthesization $(A_1 A_2) A_3$ requires
  - $A' = (A_1 A_2)$:
  - $A' A_3$ :
  - Sum:
Matrix Chain Multiplication: Example

- Computation of the product $A_1A_2A_3$, where
  - $A_1 : 10 \times 100$ matrix
  - $A_2 : 100 \times 5$ matrix
  - $A_3 : 5 \times 50$ matrix
- Parenthesization $(A_1 (A_2 A_3 ))$ requires
  - $A'' = (A_2 A_3)$:
  - $A_1 A''$:
  - Sum:
Structure of an Optimal Parenthesization

\[ (A_{i...j}) = ((A_{i...k})(A_{k+1...j})) \quad i \leq k < j \]

- Any optimal solution to the matrix-chain multiplication problem solutions to subproblems.

Determining an optimal recursively

- Let \( m[i,j] \) be the minimum number of operations needed to compute the product \( A_{i...j} \):
  - \( m[i,j] = 0 \) if \( i = j \)
  - \( m[i,j] = \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \} \)
  - \( s[i,j] = \) optimal splitting value \( k \)
  - the optimal parentheses of \( (A_{i...j}) \) splits the product between \( A_k \) and \( A_{k+1} \)
Recursive Matrix Chain Multiplication

Algorithm `rec-mat-chain(p, i, j)`
Input: sequence `p = ⟨p₀, p₁, ..., pₙ⟩`, where `pᵢ₋₁ × pᵢ` is the dimensions of matrix `Aᵢ`
Invariant: `rec-mat-chain(p, i, j)` returns `m[i, j]`

1. if `i = j` then return 0
2. `m[i, j] := ∞`
3. for `k := i to j – 1` do
   4. `m[i, j] := min(m[i,j], pᵢ₋₁ pₖ pⱼ +
      rec-mat-chain(p, i, k) +
      rec-mat-chain(p, k+1, j))`
5. return `m[i, j]`

Initial call: `rec-mat-chain(p, 1, n)`
Recursive Matrix Chain Multiplication — Runtime

Let $T(n)$ be the time taken by rec-mat-chain($p,1,n$).

\[
T(1) \geq 1 \\
T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k)) + 1 \\
\geq n + 2 \sum_{i=1}^{n-1} T(i) \\
\Rightarrow T(n) \geq 3^{n-1} \quad \text{(induction)}
\]

Exponential runtime!
Matrix Chain Multiplication — Dynamic Programming

Algorithmus dyn-mat-chain

Input:  sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$  $p_{i-1} \times p_i$ dimension of matrix $A_i$

Output: $m[1,n]$

1 $n := \text{length}(p)$
2 for $i := 1$ to $n$ do $m[i, i] := 0$
3 for $l := 2$ to $n$ do /* $l = \text{length of the subproblem} */$
4   for $i := 1$ to $n - l + 1$ do /* $i$ is the left index */
5     $j := i + l - 1$ /* $j$ is is the right index*/
6     $m[i, j] := \infty$
7     for $k := i$ to $j - 1$ do
8         $m[i, j] := \min(m[i, j], p_{i-1} p_k p_j + m[i, k] + m[k + 1, j])$
9 return $m[1, n]$

Rechnernetze und Telematik
Albert-Ludwigs-Universität Freiburg
Christian Schindelhauer
Example

\[ \begin{align*}
  A_1 & \quad 30 \times 35 \\
  A_2 & \quad 35 \times 15 \\
  A_3 & \quad 15 \times 5 \\
  A_4 & \quad 5 \times 10 \\
  A_5 & \quad 10 \times 20 \\
  A_6 & \quad 20 \times 25 \\
\end{align*} \]

\[ P = (30,35,15,5,10,20,25) \]
Example

P = (30,35,15,5,10,20,25)
Example

\[ m[2, 5] = \min_{2 \leq k \leq 5} \{ m[2, k] + m[k + 1, 5] + p_1 p_k p_5 \} \]

\[ = \min \left\{ \begin{array}{l}
  m[2, 2] + m[3, 5] + p_1 p_2 p_5 \\
  m[2, 3] + m[4, 5] + p_1 p_3 p_5 \\
  m[2, 4] + m[5, 5] + p_1 p_4 p_5
\end{array} \right\} \]

\[ = \min \left\{ \begin{array}{l}
  0 + 2,500 + 35 \cdot 15 \cdot 20 \\
  2,625 + 1,000 + 35 \cdot 5 \cdot 20 \\
  4,375 + 0 + 35 \cdot 10 \cdot 20
\end{array} \right\} \]

\[ = \min \left\{ \begin{array}{l}
  13,000 \\
  7,125 \\
  11,375
\end{array} \right\} \]

\[ = 7,125 \]
Matrix Chain Multiplication and Optimal Splitting Values using Dynamic Programming

Algorithm $\text{dyn-mat-chain}(p)$

Input: sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$ $p_{i-1} \times p_i$ the dim. of matrix $A_i$

Output: $m[1,n]$ and a matrix $s[i,j]$ containing the optimal splitting values

1. $n := \text{length}(p)$
2. for $i := 1$ to $n$ do $m[i, i] := 0$
3. for $l := 2$ to $n$ do
4.     for $i := 1$ to $n - l + 1$ do
5.         $j := i + l - 1$
6.         $m[i, j] := \infty$
7.     for $k := i$ to $j - 1$ do
8.         $q := m[i, j]$
9.         $m[i, j] := \min(m[i, j], p_{i-1} p_k p_j + m[i, k] + m[k + 1, j])$
10. if $m[i, j] < q$ then $s[i, j] := k$
11. return $(m[1, n], s)$
Example of Splitting Values

```
    j
  6   1
  5  2   3
 3  4  3  4
 2
 1  3  3  5
  1 2  3  4
  2  3  4  5
     1  2  3  4  5

i
```
Computation of an Optimal Parenthesization

Algorithm Opt-Parenths

Input: chain $A$ of matrices, matrix $s$ containing the optimal splitting values, two indices $i$ and $j$

Output: an optimal parenthesization of $A_{i...j}$

1 if $i < j$
2 then $X := \text{Opt-Parenths}(A, s, i, s[i, j])$
3 \hspace{1em} $Y := \text{Opt-Parenths}(A, s, s[i, j] + 1, j)$
4 \hspace{1em} return $(X \cdot Y)$
5 else return $A_i$

Initial call: Opt–Parenths($A$, $s$, 1, $n$)
Matrix Chain Multiplications using Dynamic Programming — Top Down

„Memoization“ for increasing the efficiency of a recursive solution:

Only the first time, a subproblem is encountered, its solution is computed and then stored in a table.

Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned (without repeated computation!)
Memoized Matrix Chain Multiplication

$m[i,j]$ initialized with $\infty$

**Algorithm** \texttt{mem-mat-chain}(\(p, i, j\))

**Invariant:** \texttt{mem-mat-chain}(\(p, i, j\)) returns \(m[i, j]\);
the value is correct if \(m[i, j] < \infty\)

1. if \(i = j\) then return 0
2. if \(m[i, j] < \infty\) then return \(m[i, j]\)
3. for \(k := i\) to \(j - 1\) do
4. \hspace{1em} \(m[i, j] := \min(m[i, j], p_{i-1} p_k p_j +\)
\hspace{1.5em} \text{mem-mat-chain}(p, i, k) +
\hspace{1.5em} \text{mem-mat-chain}(p, k + 1, j))$
5. return \(m[i, j]\)
Memoized Matrix Chain Multiplication

Call:

1 \( n := \text{length}(p) - 1 \)
2 \( \text{for } i := 1 \text{ to } n \text{ do} \)
3 \( \text{for } j := 1 \text{ to } n \text{ do} \)
4 \( m[i, j] := \infty \)
5 \text{mem-mat-chain}(p, 1, n)

The computation of all entries \( m[i, j] \) using \text{mem-mat-chain} takes \( \Omega(n^3) \) time.

\( \Omega(n^2) \) entries

- each entry \( m[i, j] \) is only computed once
- each entry \( m[i, j] \) is looked up during the computation of \( m[i', j'] \)
  - if \( i' = i \) and \( j' > j \) or \( j' = j \) and \( i' < i \)

\( m[i, j] \) is looked up for at most \( 2n \) entries
Final Remarks about Matrix Chain Multiplication

1. There is an algorithm that determines an optimal parenthesization in time $O(n \log n)$
2. There is a linear time algorithm that determines a parenthesization using at most $1.155 M_{opt}$ multiplications.
Method of Dynamic Programming

- **Recursive approach**
  - Solve a problem by solving several smaller analogous subproblems of the same type.
  - Then combine these solutions to generate a solution to the original problem.

- **Drawback: Repeated computation of solutions**

- **Dynamic programming**
  - Once a subproblem has been solved, store its solution in a table so that it can be retrieved later by simple table lookup
Two Different Approaches

- **Bottom-up:**
  + the table is maintained in an efficient way, time saving
  + subproblems are solved in a special, optimized order, space saving
  - extensive rewriting of the original problem code is necessary
  - possibly, unnecessary subproblems are solved

- **Top-down (memoization)**
  + only slight modifications in the original program code are necessary
  + only those subproblems definitely required are solved
  - separate table management is time consuming
  - table size is often suboptimal
Optimal Substructure

Dynamic programming is typically applied to

Optimization problems

An optimal solution to the original problems contains optimal solutions to smaller subproblems
Construction of Optimal Binary Search Tree

\((-\infty, k_1) \quad k_1 \quad (k_1, k_2) \quad k_2 \quad (k_2, k_3) \quad k_3 \quad (k_3, k_4) \quad k_4 \quad (k_4, \infty)\)

\begin{align*}
4 & \quad 1 & \quad 0 & \quad 3 & \quad 0 & \quad 3 & \quad 0 & \quad 3 & \quad 10
\end{align*}

weighted path length:
\[3 \cdot 1 + 2 \cdot (1 + 3) + 3 \cdot 3 + 2 \cdot (4 + 10)\]
Construction of Optimal Binary Search Trees

**Give:** set of keys $S$

$$S = \{k_1, \ldots, k_n\} \quad -\infty = k_0 < k_1 < \ldots < k_n < k_{n+1} = \infty$$

$a_i$: (absolute) frequency of request to key $k_i$

$b_j$: (absolute) frequency of request to $x \in (k_j, k_{j+1})$

Weighted path length $P(T)$ of a binary search tree $T$ for $S$:

$$P(T) = \sum_{i=1}^{n} (\text{depth}(k_i) + 1)a_i + \sum_{j=0}^{n} \text{depth}(k_j, k_{j+1})b_j$$

**Goal:** Binary search tree with minimum weighted path length $P$ for $S$
Construction of Optimal Binary Search Trees

P(T_1) = 21

P(T_2) = 27
Construction of Optimal Binary Search Trees

An optimal binary search tree is a binary search tree with minimum weighted path length.
Construction of Optimal Binary Search Tree

\[ P(T) = P(T_l) + W(T_l) + P(T_r) + W(T_r) + a_{root} \]

\[ = P(T_l) + P(T_r) + W(T) \]

\[ W(T) := \text{total weight of all nodes in } T \]

If \( T \) is a tree with minimum weighted path length \( S \), then subtree \( T_l \) and \( T_r \) are trees with minimum weighted path length for subsets of \( S \).
Construction of Optimal Binary Search Trees

Let

- $T(i, j)$: optimal binary search tree for $(k_i, k_{i+1})$ $k_{i+1}$ ... $k_j$ $(k_j, k_{j+1})$,
- $W(i, j)$: weight of $T(i, j)$, i.e. $W(i, j) = b_i + a_{i+1} + ... + a_j + b_j$,
- $P(i, j)$: weighted path length of $T(i, j)$.
Construction of Optimal Binary Search Trees

\[ T(i, j) = \]

\[ \begin{align*}
T(i, l - 1) & \\
T(l, j) & \\
\end{align*} \]

request frequency:

\[ b_i a_{i+1} a_{i-1} b_{i-1} a_i b_j a_{i+1} a_j b_j \]
Construction of Optimal Binary Search Trees

\[ W(i, i) = b_i, \text{ for } 0 \leq i \leq n \]

\[ W(i, j) = W(i, j - 1) + a_j + b_j, \text{ for } 0 \leq i < j \leq n \]

\[ P(i, i) = 0, \text{ for } 0 \leq i \leq n \]

\[ P(i, j) = W(i, j) + \min_{i < l \leq j} \{ P(i, l - 1) + P(l, j) \}, \text{ for } 0 \leq i < j \leq n (*) \]

\[ r(i, j) = \text{the index } l \text{ for which the minimum is achieved in (*)} \]
Construction of Optimal Binary Search Trees

Base cases

Case 1: $h = j - i = 0$

$$T(i, i) = (k_i, k_{i+1})$$

$$W(i, i) = b_i$$

$$P(i, i) = 0, \ r(i, i) \ text{ not defined}$$
Construction of Optimal Binary Search Trees

Case 2: \( h = j - i = 1 \)

\[
\begin{align*}
W(i, i+1) &= b_i + a_{i+1} + b_{i+1} = W(i, i) + a_{i+1} + W(i+1, i+1) \\
P(i, i+1) &= W(i, i + 1) \\
r(i, i+1) &= i + 1
\end{align*}
\]
Computing the Minimum Weighted Path Length using Dynamic Programming

Case 3: $h = j - i > 1$

for $h = 2$ to $n$ do

for $i = 0$ to $(n - h)$ do

&{ j = i + h; \\
  determine (largest) $l$, $i < l \leq j$, s.t. $P(i, l - 1) + P(l, j)$ is minimal \\
  $P(i, j) = P(i, l - 1) + P(l, j) + W(i, j)$; \\
  $r(i, j) = l$; \\
  }
Construction of Optimal Binary Search Trees

Define:

\[ P(i, j) := \text{minimum weighted path length for sum of} \ \{ b_i a_{i+1} b_{i+1} \ldots a_j b_j \} \]

\[ W(i, j) := \]

Then:

\[ W(i, j) = \begin{cases} b_i & \text{if } i = j \\ W(i, j - 1) + a_j + W(j, j) & \text{otherwise} \end{cases} \]

\[ P(i, j) = \begin{cases} 0 & \text{if } i = j \\ W(i, j) + \min_{i < \ell \leq j} \{ P(i, \ell - 1) + P(\ell, j) \} & \text{otherwise} \end{cases} \]

→ Computing the solution \( P(0, n) \) takes time \( O(n^3) \) and requires \( O(n^2) \) space
Construction of Optimal Binary Search Trees

Theorem
An optimal binary search tree for \( n \) keys and \( n+1 \) intervals with known request frequencies can be constructed in \( O(n^3) \) time.
Method of Dynamic Programming

- Recursive approach
  - Solve a problem by solving several smaller analogous subproblems of the same type.
  - Then combine these solutions to generate a solution to the original problem.
- Drawback: Repeated computation of solutions
- Dynamic programming
  - Once a subproblem has been solved, store its solution in a table so that it can be retrieved later by simple table lookup
Dynamic Programming

- Algorithm design technique, often applied to optimization problems
- Generally suitable for recursive approaches, when solution to subproblems are required repeatedly
- Approach
  - maintain a table of subproblem solutions
- Advantage
  - improved running time
  - often polynomial instead of exponential
String Matching Problems

Edit Distance

For two given strings $A$ and $B$, compute the edit distance $D(A,B)$ as well as a minimum sequence of edit operations that transforms $A$ into $B$.

```
mathematican
multiplication
```
String Matching Problems

Approximate String Matching

For a given text $T$, a pattern $P$ and a distance $d$, find all substrings $P'$ of $T$ with $D(P,P') \leq d$

Sequence Alignment

Find optimal alignments of DNA sequences

\begin{align*}
G & A & G & C & A & - & C & T & T & G & G & A & T & T & C & T & C & G & G \\
\end{align*}
Edit Distance

**Given:** Strings $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$

**Goal:** Minimum number $D(A,B)$ of edit operations required to transform $A$ into $B$.

**Edit operations:**
1. Replace a character from string $A$ by a character from $B$
2. Delete a character from string $A$
3. Insert a character from string $B$ into string $A$.

\[
\text{mathematician} \quad \text{multiplication}
\]
Edit Distance

Unit cost model:
for a, b being characters or empty words, i.e. \( \varepsilon \)

\[
c(a, b) = \begin{cases} 
1 & \text{if } a \neq b \\
0 & \text{if } a = b 
\end{cases}
\]

We want to have a metric. Hence it should satisfy the triangle inequality:

\[
c(a, c) \leq c(a, b) + c(b, c)
\]

→ for strings only one letter is changed at a time
→ each change increases the cost by one unit
**Edit Distance**

*Trace* as representation of the sequence of edit operations:

\[
A = \begin{array}{cccccccc}
    b & a & a & c & a & a & b & c \\
\end{array} \\
\]

\[
B = \begin{array}{cccccccc}
    a & b & a & c & b & c & a & c \\
\end{array} \\
\]

or using *indents*

\[
A = \begin{array}{cccccccc}
    - & b & a & a & c & a & - & a & b & c \\
\end{array} \\
\]

\[
B = \begin{array}{cccccccc}
    a & b & a & - & c & b & c & a & - & c \\
\end{array} \\
\]

Edit distance (costs) : 5

Splitting an optimal trace yields two optimal subtraces

→ dynamic programming is suitable
Computation of the Edit Distance

Let $A_i = a_1...a_i$ and $B_j = b_1....b_j$

$$D_{ij} = D(A_i, B_j)$$
Computation of the Edit Distance

Three ways of ending a trace

1. $a_m$ is replaced by $b_n$:
   \[ D_{m,n} = D_{m-1,n-1} + c(a_m, b_n) \]

2. $a_m$ is deleted: $D_{m,n} = D_{m-1,n} + 1$

3. $b_n$ is inserted: $D_{m,n} = D_{m,n-1} + 1$
Computation of the Edit Distance

Recurrence relation \((m,n \geq 1)\):

\[
D_{m,n} = \min \left\{ \begin{array}{c}
D_{m-1,n-1} + c(a_m, b_n) \\
D_{m-1,n} + 1 \\
D_{m,n-1} + 1
\end{array} \right\}
\]

→ computation of all \(D_{ij}\) necessary, \(0 \leq i \leq m, 0 \leq j \leq n\).
Recurrences for the Edit Distance

Base case:

\[ D_{0,0} = D(\varepsilon, \varepsilon) = 0 \]
\[ D_{0,j} = D(\varepsilon, B_j) = j \]
\[ D_{i,0} = D(A_i, \varepsilon) = i \]

Recurrence equation:

\[
D_{i,j} = \min \left\{ \begin{array}{c}
D_{i-1,j-1} + c(a_i, b_j) \\
D_{i-1,j} + 1 \\
D_{i,j-1} + 1
\end{array} \right\}
\]
Order of Solving the Subproblems

\[ b_1 \quad b_2 \quad b_3 \quad b_4 \quad \ldots \quad b_n \]

\[
\begin{array}{cccc}
D_{i-1,j-1} & +c(a_i,b_j) & \downarrow & D_{i,j} \\
D_{i,j-1} & +1 & \downarrow & D_{i,j} \\
\end{array}
\]

60

\[ \text{min} \]
Algorithm for Computing the Edit Distance

Algorithm Edit-Distance

Input: Strings $A = a_1 \ldots a_m$ and $B = b_1 \ldots b_n$

Output: Matrix $D = (D_{ij})$

1 $D[0,0] := 0$
2 for $i := 1$ to $m$ do $D[i,0] = i$
3 for $j := 1$ to $n$ do $D[0,j] = j$
4 for $i := 1$ to $m$ do
5   for $j := 1$ to $n$ do
6       $D[i,j] := \min(D[i-1,j] + 1,$
7       $D[i,j-1] + 1,$
8       $D[i-1,j-1] + c(a_i,b_j))$
### Example

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Computing the Edit Operations

Algorithm Edit-operations \((i,j)\)

Input: matrix \(D\) (already computed)

Output: sequence of edit operations

1. if \(i = 0\) and \(j = 0\) then return
2. if \(i \neq 0\) and \(D[i,j] = D[i-1, j] + 1\)

   then Edit-operations \((i-1, j)\)

   „delete \(a[i]\)“

3. else if \(j \neq 0\) and \(D[i,j] = D[i, j-1] + 1\)

   then Edit-operations \((i, j-1)\)

   „insert \(b[j]\)“

4. else /* \(D[i,j] = D[i-1, j-1] + c(a[i], b[j])\) */

   Edit-operations \((i-1, j-1)\)

   „replace \(a[i]\) by \(b[j]\)“

Initial call: Edit-operations \((m,n)\)
Trace Graph of Edit Operations

\[ A = \begin{array}{ccc}
0 & 1 & 2 \\
1 & 2 & 2 \\
2 & 2 & 2 \\
3 & 3 & 2 \\
4 & 3 & 2 \\
\end{array} \]

\[ B = \begin{array}{ccc}
a & b & a & c \\
0 & 1 & 2 & 3 & 4 \\
1 & 1 & 2 & 1 & 2 & 3 \\
2 & 1 & 2 & 1 & 2 \\
3 & 2 & 2 & 2 & 2 \\
4 & 3 & 3 & 2 & 2 \\
\end{array} \]
Trace Graph of the Edit Operations

- **Trace Graph:**
  - Representation of all possible traces of operations that transform A into B. Direct edges from vertex \((i,j)\) to vertices \((i+1), (i,j+1)\) and \((i+1,j+1)\).

- **Edge weights represent the edit costs.**

- **Along an optimal path, costs increase monotonically**

- **Each path from upper left corner to the lower right corner with monotonically increasing costs represents an optimal trace**
Approximate String Matching

Given: pattern string $P = p_1p_2 \ldots p_m$ and text string $T = t_1t_2 \ldots t_n$

Goal: Find an interval $[j', j]$, $1 \leq j', j \leq n$, such that the substring $T_{j', j} = t_{j'} \ldots t_j$ is the one with the highest similarity to the pattern $P$.

Thus, for all other intervals $[k', k]$, $1 \leq k', k \leq n$:

$$D(P, T_{j', j}) \leq D(P, T_{k', k})$$
Approximate String Matching

Naive approach:

for all $1 \leq j', j \leq n$ do
compute $D(P,T_{j', j})$

choose the minimum

Running Time $O(n^3m)$
Approximate String Matching

Consider a related problem:

For each position $j$ in the text and each position $i$ in the pattern compute the minimum edit distance between $P_i$ and any substring $T_{j'..j}$ of $T$ that ends at position $j$. 
Approximative String Matching

Method:
for all $1 \leq j \leq n$ do
  determine $j'$, so that $D(P,T_{j'},j)$ is minimized

For $1 \leq i \leq m$ and $0 \leq j \leq n$ let:

$$E_{i,j} = \min_{1 \leq j' \leq j+1} D(P_i,T_{j'},j)$$

Optimal trace:

$P_i = \text{b a a c a a b c}$

$T_{j',j} = \text{b a c b c a c}$
Approximative String Matching

Recurrence equation:

\[
E_{i,j} = \min \left\{ \begin{array}{l}
E_{i-1,j-1} + c(p_i, t_j), \\
E_{i-1,j} + 1, \\
E_{i,j-1} + 1
\end{array} \right\}
\]

Remarks:

The index \( j' \) may differ for \( E_{i-1,j-1}, E_{i-1,j} \) and \( E_{i,j-1} \).

A subtrace of an optimal trace is an optimal subtrace.
Approximate String Matching

Base case:

\[ E_{0,0} = E(\varepsilon, \varepsilon) = 0 \]
\[ E_{i,0} = E(P_i, \varepsilon) = i \]

whereas

\[ E_{0,j} = E(\varepsilon, T_j) = 0 \]

Observation:

An optimal sequence of edit operations that transforms P into 
\( T_{j',j} \) does not start with an insertion of character \( t_{j'} \).
Approximate String Matchings

Dependency Graph

\[ T = \begin{array}{ccccccccccc}
  a & b & b & d & a & d & c & b & a \\
\end{array} \]

\[ P = \begin{array}{ccccccccccc}
  0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \]

\[ \Pi \]

\[ a \]
\[ \begin{array}{c}
  1 \\
\end{array} \]

\[ d \]
\[ \begin{array}{c}
  2 \\
\end{array} \]

\[ b \]
\[ \begin{array}{c}
  3 \\
\end{array} \]

\[ b \]
\[ \begin{array}{c}
  4 \\
\end{array} \]

\[ c \]
\[ \begin{array}{c}
  5 \\
\end{array} \]
Approximate String Matching

**Theorem**

If there is a path from $E_{0, j-1}$ to $E_{i, j}$, in the dependency graph, then $T_{j', j}$ is a substring of $T$ that has the highest to $P_{j'}$, ending at position $j$ and satisfying

$$D(P_{j'}, T_{j', j}) = E_{i, j}$$
Similarity of Strings

Sequence Alignment:

For two given DNA sequences, insert spaces (or dashes) such that after placing the resulting strings one above the other, the number of matching characters is maximized.

\[
\begin{align*}
G & A & - & C & G & G & A & T & T & A & G \\
G & A & T & C & G & G & A & A & T & A & G
\end{align*}
\]
Similarity of Strings

Similarity measure for characters

<table>
<thead>
<tr>
<th>example</th>
<th>setting</th>
<th>in general</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ 1</td>
<td>for a match</td>
<td>} s(a,b)</td>
</tr>
<tr>
<td>- 1</td>
<td>for a mismatch</td>
<td></td>
</tr>
<tr>
<td>- 2</td>
<td>for spaces</td>
<td>- c</td>
</tr>
</tbody>
</table>

Measuring the similarity of two sequences

\[ S(A, B) = \sum_{\text{characters } a_i, b_i} \text{similarity of}(a_i, b_i) \]

Goal: Find alignment optimizing similarity
Similarity of Strings

Similarity $S(A,B)$ of two strings $A$ and $B$

**Operations:**
1. Replacement of a character $a$ by some character $b$: $s(a,b)$
2. Deletion of a character from $A$, insertion of a character from $B$, Loss: $-c$

**Goal:**
Find a sequence of operations that transforms $A$ into $B$ such that the total gain is maximized.
Similarity of Strings

\[ S_{i,j} = S(A_i, B_j), \ 0 \leq i \leq m, \ 0 \leq j \leq n \]

Recurrence equation:

\[ S_{m,n} = \max \ (S_{m-1,n-1} + s(a_m, b_n), \ S_{m-1,n} - c, \ S_{m,n-1} - c) \]

Initial condition:

\[ S_{0,0} = S(\epsilon, \epsilon) = 0 \]
\[ S_{0,j} = S(\epsilon, B_j) = -jc \]
\[ S_{i,0} = S(A_i, \epsilon) = -ic \]
Most Similar Substring

**Given:** Two strings $A = a_1 \ldots a_m$ and $B = b_1 \ldots b_n$

**Goal:** Find two intervals $[i', i] \subseteq [1, m]$ and $[j', j] \subseteq [1, n]$ with

$$S(A_{i', i}, B_{j', j}) \geq S(A_{k', k}, B_{l', l}),$$

for all $[k', k] \subseteq [1, m]$ and $[l', l] \subseteq [1, n]$.

**Naive Approach:**

for all $[i', i] \subseteq [1, m]$ and $[j', j] \subseteq [1, n]$ do

compute $S(A_{i', i}, B_{j', j})$

**Running time:** $O(m^2 n^2)$
Most Similar Substrings

Method:

for all $1 \leq i \leq m$, $1 \leq j \leq n$ do

Compute $i'$ und $j'$, such that $S(A_{i',i}, B_{j',j})$ is maximal

For $0 \leq i \leq m$ und $0 \leq j \leq n$ let:

$$H_{i,j} = \max_{1 \leq i' \leq i+1, \, 1 \leq j' \leq j+1} S(A_{i',i}, B_{j',j})$$

Optimal trace

$$A_{i',i} = \begin{array}{ccccccccc}
  b & a & a & c & a & - & a & b & c \\
\end{array}$$

$$B_{j',j} = \begin{array}{ccccccccc}
  b & a & - & c & b & c & a & - & c \\
\end{array}$$
Most Similar Substring

Recurrence relation:

\[ H_{i,j} = \max \begin{cases} 
H_{i-1,j-1} + s(a_i, b_j) \\
H_{i-1,j} - c \\
H_{i,j-1} - c \\
0 
\end{cases} \]

Base cases:

\[
\begin{align*}
H_{0,0} &= H(\varepsilon, \varepsilon) = 0 \\
H_{i,0} &= H(A_i, \varepsilon) = 0 \\
H_{0,j} &= H(\varepsilon, B_j) = 0 
\end{align*}
\]

in our example:
- \( s(a,a) = +1 \)
- \( s(a,b) = -1 \) for \( a \neq b \)
- \( c = -2 \) (inserting/deleting)
Most similar substring

Dependency Graph

\[ B = \begin{array}{cccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 2 & 0
\end{array} \]
Algorithm Theory
11 Dynamic Programming

Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Rechnernetze und Telematik
Wintersemester 2007/08