Algorithms Theory

14 – Dynamic Programming (4)

Edit distance
Approximate string matching
Sequence alignment

P.D. Dr. Alexander Souza

Winter term 11/12

Dynamic programming

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

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 program 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

String matching problems

Edit distance

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

m	a	-	t	h	e	m	-	-	a	t	i	c	i	a	n
m	u	l	t	i	p	l	i	c	a	t	i	o	-	-	n

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.

```
G A G C A -  C T T G G A T T C T C G G
 -  -  -  C A C G T G G -  -  -  -  -   -  -  -   -
```

Edit distance

Given: Two strings $A = a_1a_2 .... a_m$ and $B = b_1b_2 ... b_n$.

Goal: Determine the 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 string *B.*
2. Delete character from string *A.*
3. Insert a character from string *B* into string *A*.

m	a	-	t	h	e	m	-	-	a	t	i	c	i	a	n
m	u	l	t	i	p	l	i	c	a	t	i	o	-	-	n

Edit distance

Unit-cost model:

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

$$a=\varepsilon,\ b=\varepsilon \ \text{ possible}$$

We assume the triangle inequality holds for *c*:

$$c(a,c) \leq c(a,b) + c(b,c)$$

→ each character is changed at most once

Edit distance

Trace as representation of the sequence of edit operations:

```
A =      b  a  a  c  a  a  b  c
         |  |    /  /     |   /
B =   a  b  a  c  b  c  a  c
```

or using indents:

```
A =  -  b  a  a  c  a  -  a  b  c
        |  |     |  |     |     |
B =  a  b  a  -  c  b  c  a  -  c
```

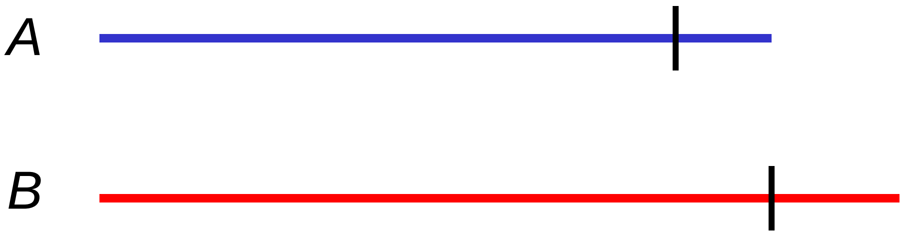
Edit distance (costs) : 5

Splitting an optimal trace yields two optimal subtraces
→ dynamic programming is suitable

Computation of the edit distance

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

$$D_{i,j} = D(A_i, B_j)$$



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: $D_{m,n} = D_{m-1,n} + 1$

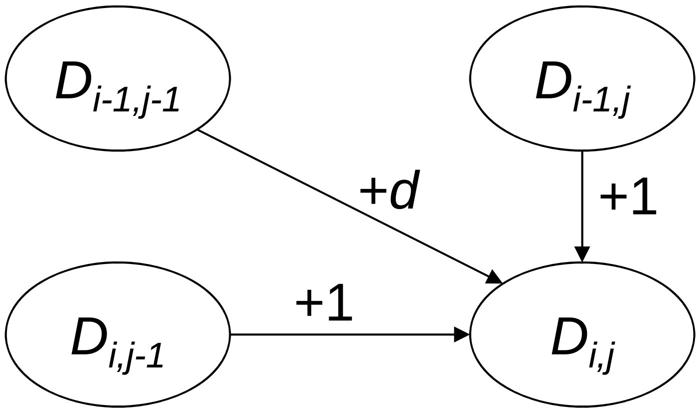
3. b_n is inserted: $D_{m,n} = D_{m,n-1} + 1$

Computation of the edit distance

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

$$D_{m,n} = \min \begin{cases} D_{m-1,n-1} & + & c(a_m, b_n) \\ D_{m-1,n} & + & 1 \\ D_{m,n-1} & + & 1 \end{cases}$$

→ computation of all $D_{i,j}$ required, $0 \leq i \leq m$, $0 \leq j \leq n$.



Recurrence relation for the edit distance

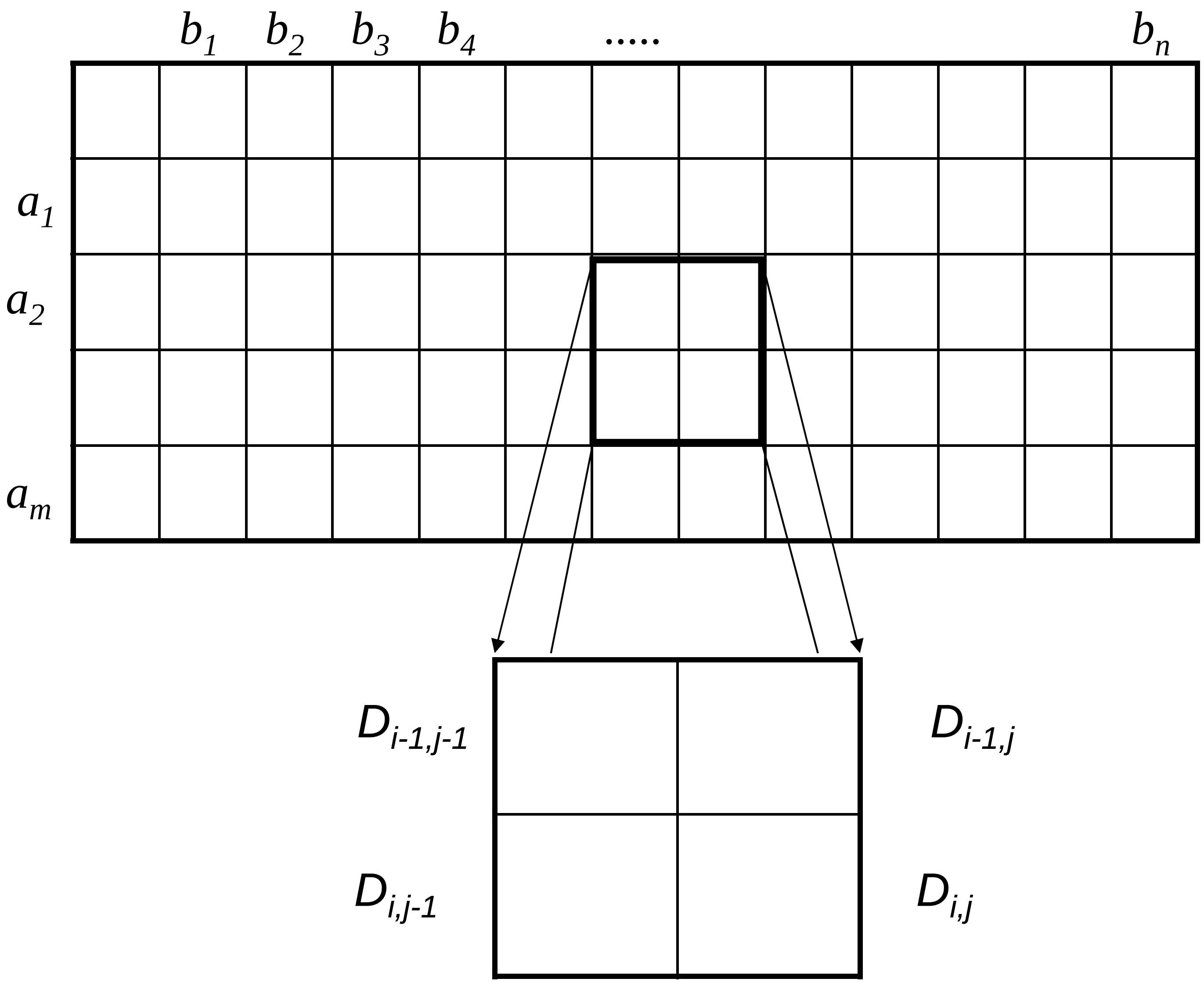
Base cases:

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

Recurrence relation:

$$D_{i,j} = \min \begin{Bmatrix} D_{i-1,j-1} & + & c(a_i, b_j) \\ D_{i-1,j} & + & 1 \\ D_{i,j-1} & + & 1 \end{Bmatrix}$$

Order of solving the subproblems



Algorithm for computing the edit distance

Algorithm *Edit-distance*

Input: two strings $A = a_1 \ldots. a_m$ and $B = b_1 \ldots b_n$

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

```
D[0,0] := 0
for i := 1 to m do D[i,0] = i
for j := 1 to n do D[0,j] = j
for i := 1 to m do
    for j := 1 to n do
          D[i,j] := min( D[i - 1,j] + 1,
                         D[i,j - 1] + 1,
                         D[i –1, j – 1] + c(a_i ,b_j ))
```

Example

		a	b	a	c
	0	1	2	3	4
b	1				
a	2				
a	3				
c	4				

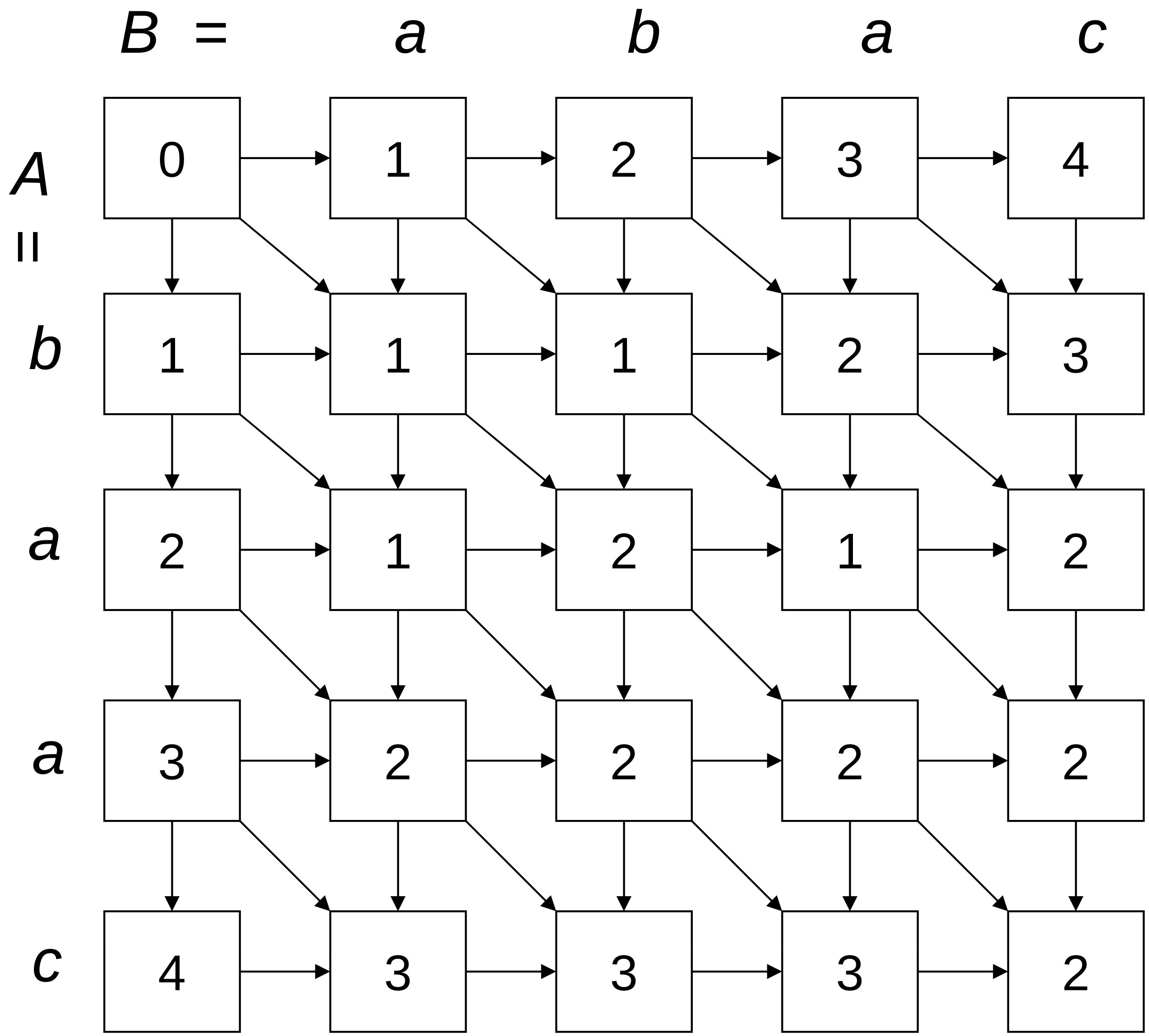
Computing the edit operations

Algorithm *Edit-operations* (i,j)
Input: matrix D (already computed)
Output: sequence of edit operations

if $i = 0$ **and** $j = 0$ **then return**
if $i \neq 0$ **and** $D[i,j] = D[i-1, j] + 1$
  then *Edit-operations* ($i - 1, j$)
    „delete $a[i]$“
else if $j \neq 0$ **and** $D[i,j] = D[i, j-1] + 1$
  then *Edit-operations* ($i, j - 1$)
    „insert $b[j]$“
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 the edit operations



Trace graph of the edit operations

Trace graph:
Representation of all possible traces of operations that transform *A* into *B.* Directed edges from vertex (i, j) to vertices $(i + 1, j)$, $(i, j + 1)$ and $(i + 1, j + 1)$.
Edge weights represent the edit costs.

Along an optimal path, costs increase monotonically.

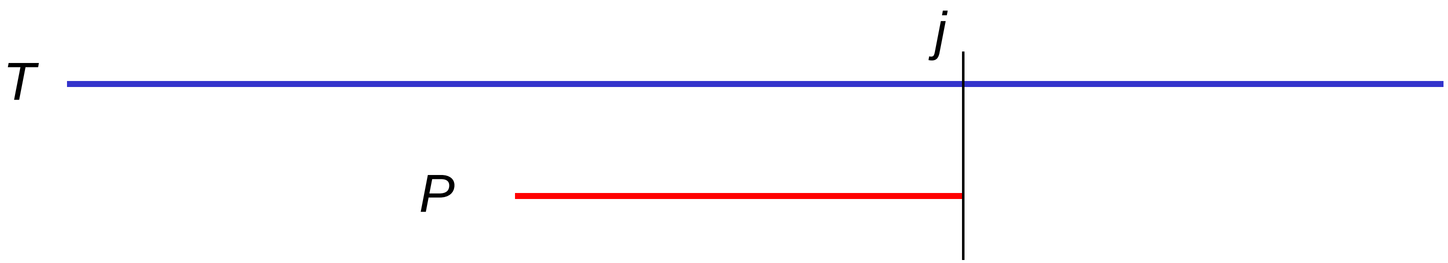
Each path from the upper left corner to the lower right corner with monotonically increasing costs represents an optimal trace.

Approximate string matching

Given: Two strings $T = t_1 t_2 \ldots t_n$ (text) and $P = p_1 p_2 \ldots p_m$ (pattern).

Goal: Find an interval $[j', j]$, $1 \leq j', j \leq n$, such that the substring $T_{j', j} = t_{j'} \ldots t_j$ of T 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:

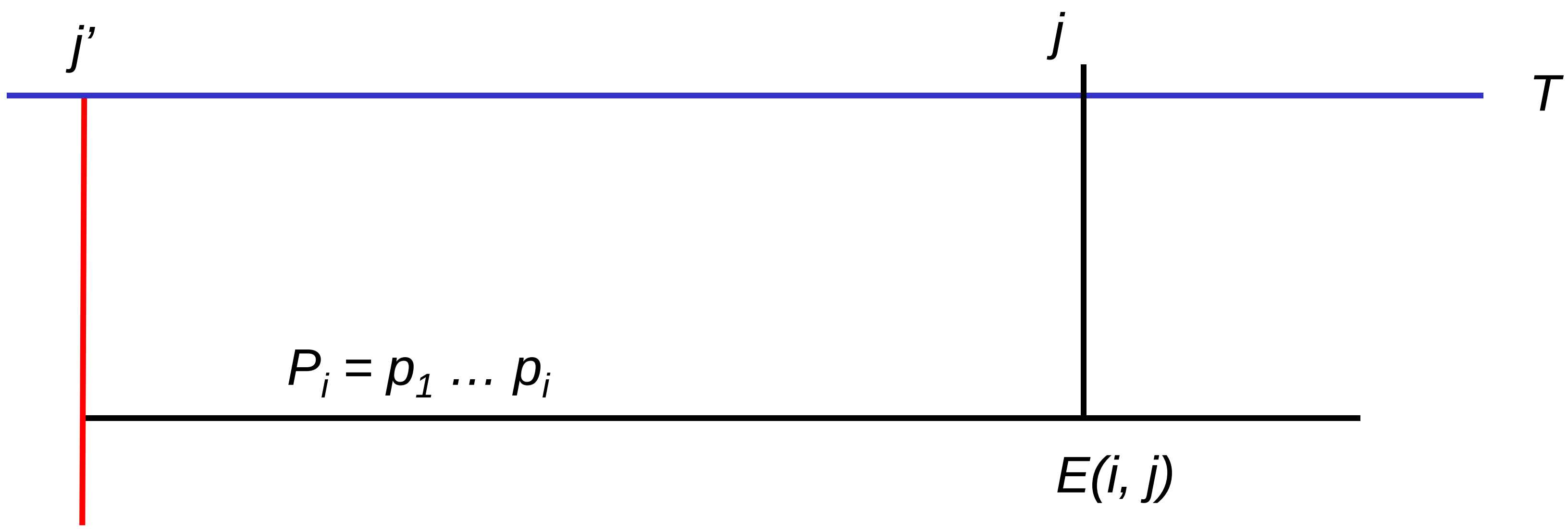
for all $1 \leq j', j \leq n$ **do**

compute $D(P, T_{j', j})$

choose the minimum

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*.

Approximate 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 $1 \leq j \leq n$ let:

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

Optimal trace:

```
P_i   = b  a  a  c  a  a  b  c
        |  |    /  /     |    /
T_j',j = b  a  c  b  c  a  c
```

Approximate string matching

Recurrence relation:

$$E_{i,j} = \min \begin{Bmatrix} E_{i-1,j-1} + c(p_i, t_j), \\ E_{i-1,j} + 1, \\ E_{i,j-1} + 1 \end{Bmatrix}$$

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 cases:

$$E_{0,0} = E(\varepsilon, \varepsilon) = 0$$

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

whereas

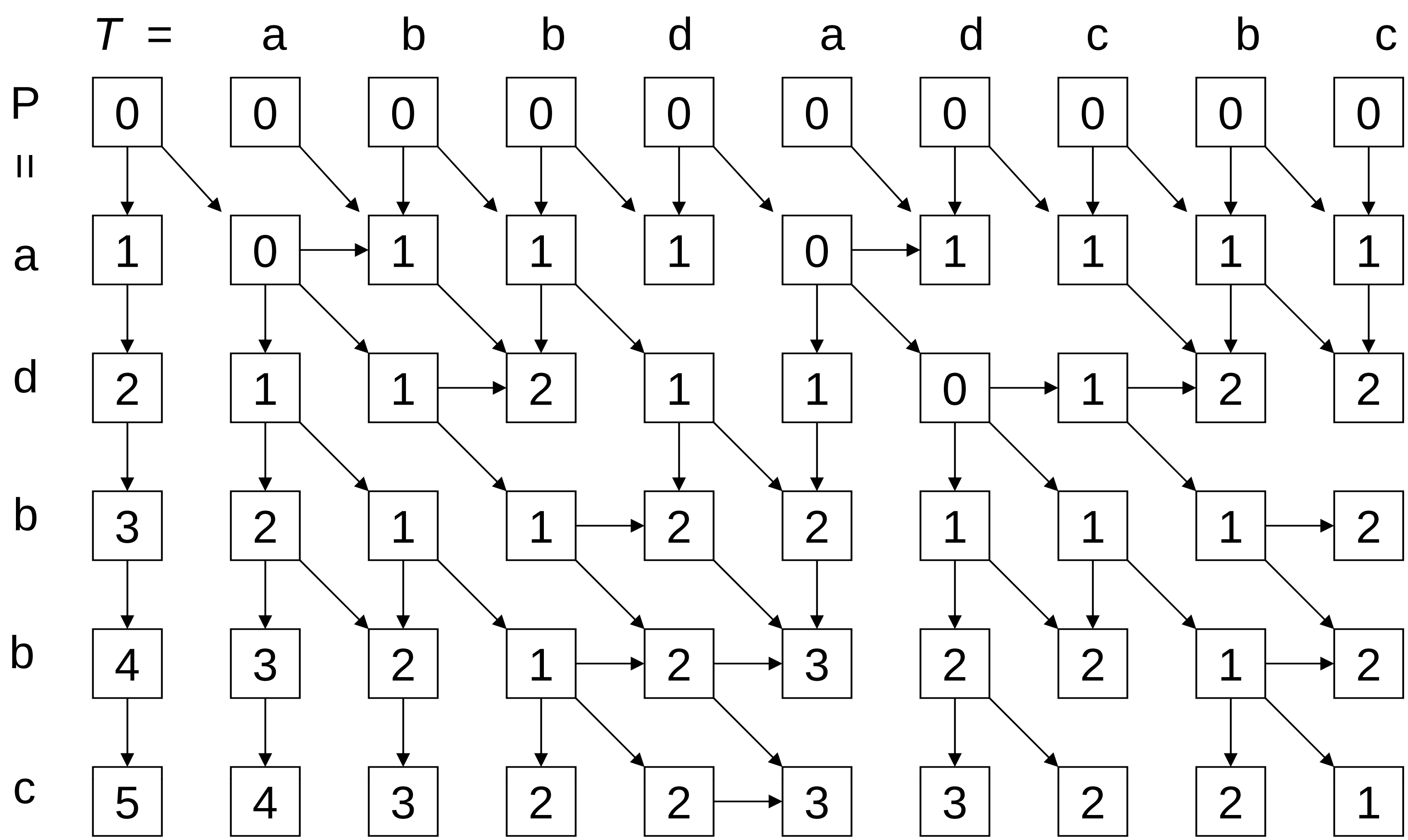
$$E_{0,j} = E(\varepsilon, T_j) = 0$$

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 matching

Dependency graph:



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 similarity to P_i, ending at position j and satisfying

$$D(P_i, T_{j', j}) = E_{i, j}$$

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.

```
G A - C G G A T T A G
G A T C G G A A T A G
```

Similarity of strings

Measuring the similarity of two characters:

example value	setting	in general
+ 1	for a match	} *s*(a,b)
- 1	for a mismatch	
- 2	for spaces	- c

Measuring the similarity of two sequences :

$$S(A,B) = \sum_{pairs\ (a_i, b_i)} \text{similarity of } (a_i, b_i)$$

Goal: Find an alignment that maximizes the 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* :
   Gain: *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 relation:

$$S_{m,n} = \max \left(S_{m-1,n-1} + s(a_m, b_n), \; S_{m-1,n} - c, \; S_{m,n-1} - c \right)$$

Base cases:

$$S_{0,0} = S(\varepsilon, \varepsilon) = 0$$
$$S_{0,j} = S(\varepsilon, B_j) = -jc$$
$$S_{i,0} = S(A_i, \varepsilon) = -ic$$

Most similar substrings

Given: Two strings $A = a_1 \dots a_m$ and $B = b_1 \dots b_n$.

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

Most similar substrings

Method:

for all $1 \le i \le m$, $1 \le j \le n$ **do**

compute i' and j' so that $S(A_{i',i}, B_{j',j})$ is maximized

For $0 \le i \le m$ and $0 \le j \le n$ let:

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

Optimal trace:

```
A_{i´,i} =  b  a  a  c  a  -  a  b  c
            |  |     |  ┃     |     |
B_{j´,j} =  b  a  -  c  b  c  a  -  c
```

Most similar substrings

Recurrence relation:

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

Base cases:

$$H_{0,0} = H(\varepsilon, \varepsilon) = 0$$
$$H_{i,0} = H(A_i, \varepsilon) = 0$$
$$H_{0,j} = H(\varepsilon, B_j) = 0$$