Chapter 3
Dynamic Programming

Algorithm Theory
WS 2018/19

Fabian Kuhn

Dynamic Programming (DP)

DP $\approx$ Recursion + Memoization

Recursion: Express problem *recursively* in terms of
(a ‘small’ number of) *subproblems* (of the same kind)

Memoize: *Store* solutions for *subproblems*
reuse the stored solutions if the same subproblems
has to be solved again

Weighted interval scheduling: subproblems $W(1), W(2), W(3)$, …

runtime $=$ #subproblems $\cdot$ time per subproblem

Dynamic Programming

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

- Only the *first time* a sub-problem 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!).

- *Computing the solution*: For each sub-problem, store how the value is obtained (according to which recursive rule).

Dynamic Programming

Dynamic programming / memoization can be applied if

- Optimal solution contains optimal solutions to sub-problems (recursive structure)

- Number of sub-problems that need to be considered is small

Knapsack

- n items $1, \dots, n$, each item has weight w_i and value v_i
- Knapsack (bag) of capacity W

- Goal: pack items into knapsack such that total weight is at most W and total value is maximized:

$$\max \sum_{i \in S} v_i$$

$$\text{s.t.} \quad S \subseteq \{1, \dots, n\} \text{ and } \sum_{i \in S} w_i \leq W$$

- E.g.: jobs of length w_i and value v_i, server available for W time units, try to execute a set of jobs that maximizes the total value

Recursive Structure?

OPT(k):

- Optimal solution: $\mathcal{O}$
- If $n \notin \mathcal{O}$: $\mathrm{OPT}(n) = \mathrm{OPT}(n-1)$
- What if $n \in \mathcal{O}$?
  - Taking n gives value v_n
  - But, n also occupies space w_n in the bag (knapsack)
  - There is space for $W - w_n$ total weight left!

$$\mathrm{OPT}(n) = v_n + \underbrace{\text{optimal solution with first } n-1 \text{ items and knapsack of capacity } W - w_n}_{\text{not just } \mathrm{OPT}(n-1)}$$

A More Complicated Recursion

$\mathbf{OPT}(\boldsymbol{k}, \boldsymbol{x})$: value of optimal solution with items $1, \dots, k$
and knapsack of capacity x

main goal: $OPT(n, W)$

Recursion:

only makes sense if ≥ 0

$$OPT(k,x) = \max\{\underbrace{OPT(k-1,x)}_{\text{opt. sol. when not using item } k},\ v_k + OPT(k-1, \underbrace{x - w_k}_{\text{remaining cap.}})\}$$

Initialization

$OPT(0, x) = 0$

$OPT(k, 0) = 0$

subproblems?

arbitrary weights $\approx 2^n$ NP-hard

integer weights : $\underline{\underline{n \cdot W}}$

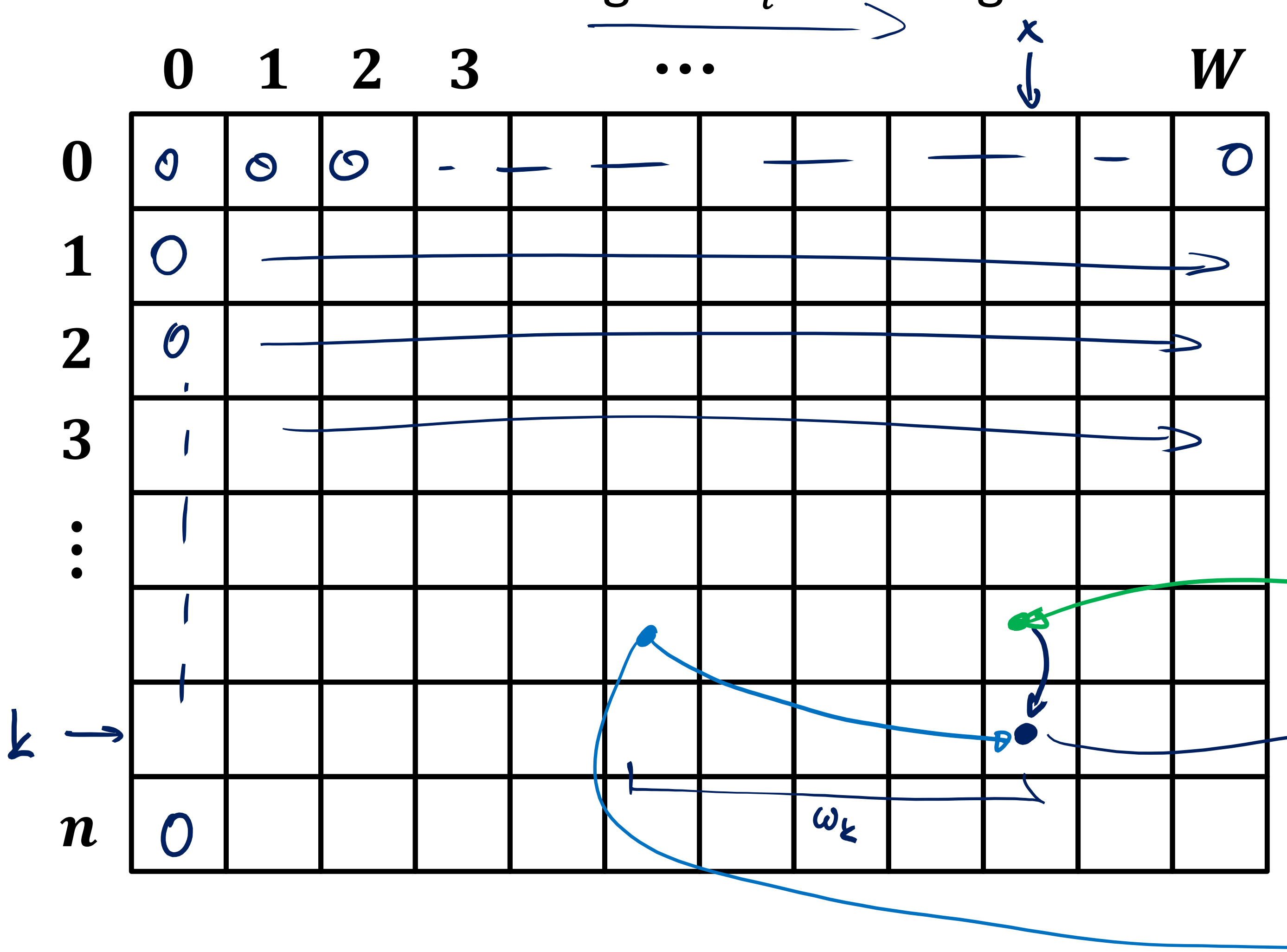
assume that weights are integers

$\Rightarrow$ time: $O(n \cdot W)$

Dynamic Programming Algorithm

Set up table for all possible $\mathrm{OPT}(k, x)$-values

- Assume that all weights w_i are integers!



Row i, column j:

$\boldsymbol{OPT(i, j)}$

$\mathrm{OPT}(k, x) = \max\{\mathrm{OPT}(k-1, x),\; v_k + \mathrm{OPT}(k-1, x - w_k)\}$

Example

- 8 items: (3,2), (2,4), (4,1), (5,6), (3,3), (4,3), (5, 4), (6,6)
  Knapsack capacity: 12

  weight value

- $OPT(k, x) = \max\{OPT(k-1, x), OPT(k-1, x-w_k) + v_k\}$

	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	2	2	2	2	2	2	2	2	2	2
2	0	4	4	4	6	6	6					
3												
4												
5												
6												
7												
8												?

7,5

Running Time of Knapsack Algorithm

- **Size of table:** $O(n \cdot W)$
- Time per table entry: $O(1)$ → **overall time: $O(nW)$**
- Computing solution (set of items to pick):
  Follow $\leq n$ arrows → $O(n)$ time (after filling table)
- Note: Time depends on W → can be exponential in n…
- And it is problematic if weights are not integers.

still possible if weights are rational

<u>another special case</u>: values are integers $\quad OPT(k, y)$

total value exactly $= y$

- general case : NP-hard

String Matching Problems

Edit distance:

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

- **Example:** mathematician → multiplication:

$$\texttt{m}\ \texttt{u}\ \underbrace{\ }_{\texttt{l}}\ \texttt{t}\ \texttt{i}\ \texttt{p}\ \texttt{l}\ \underbrace{\ }_{\texttt{ic}}\ \texttt{a}\ \texttt{t}\ \texttt{i}\ \texttt{o}\ \cancel{\texttt{i}}\ \cancel{\texttt{a}}\ \texttt{n}$$

10 ops.

Edit Distance

Given: Two strings $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$

Goal: Determine the minimum number $D(A, B)$ of edit operations required to transform A into B

Edit operations:

a) Replace a character from string A by a character from B

b) Delete a character from string A

c) Insert a character from string B into A

```
    ε
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
```

alignment

Edit Distance – Cost Model

- Cost for **replacing** character a by b: $\boldsymbol{c(a,b) \geq 0}$

- Capture insert, delete by allowing $a = \varepsilon$ or $b = \varepsilon$:
  - Cost for **deleting** character a: $\boldsymbol{c(a,\varepsilon)}$
  - Cost for **inserting** character b: $\boldsymbol{c(\varepsilon,b)}$

- **Triangle inequality**:

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

→ each character is changed at most once!

- **Unit cost model**: $c(a,b) = \begin{cases} 1, & \text{if } a \neq b \\ 0, & \text{if } a = b \end{cases}$

Recursive Structure

- Optimal "alignment" of strings (unit cost model)

  `bbcadfagikccm` and `abbagflrgikacc`:

  k

  ```
  - b b c a g f a - g i k - c c m
  a b b - a d f l r g i k a c c -
  ```

  ℓ

- Consists of optimal "alignments" of sub-strings, e.g.:

  ```
  -bbcagfa         -gik-ccm
             and
  abb-adfl         rgikacc-
  ```

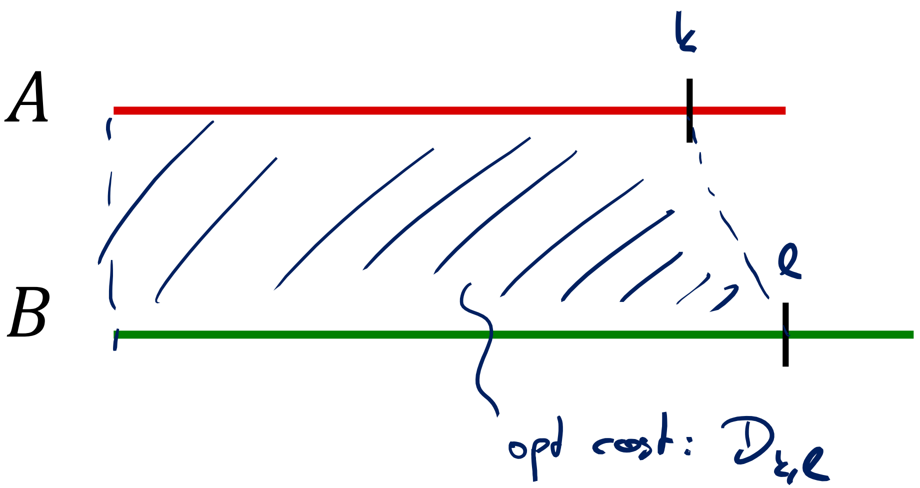
- Edit distance between $A_{1,m} = a_1 \dots a_m$ and $B_{1,n} = b_1 \dots b_n$:

$$D(A,B) = \min_{k,\ell}\left\{\underline{D\left(A_{1,k}, B_{1,\ell}\right)} + \underline{D\left(A_{k+1,m}, B_{\ell+1,n}\right)}\right\}$$

Computation of the Edit Distance

Let $A_k := a_1 \dots a_k$, $B_\ell := b_1 \dots b_\ell$, and

$$D_{k,\ell} := D(A_k, B_\ell)$$

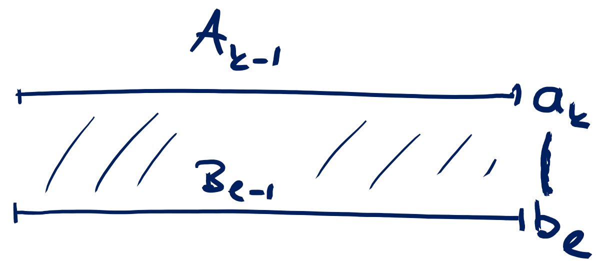


Computation of the Edit Distance

Three ways of ending an "alignment" between A_k and B_ℓ:

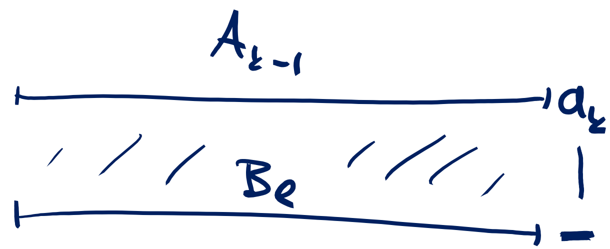
1. a_k is replaced by b_ℓ:

   $D_{k,\ell} = D_{k-1,\ell-1} + c(a_k, b_\ell)$



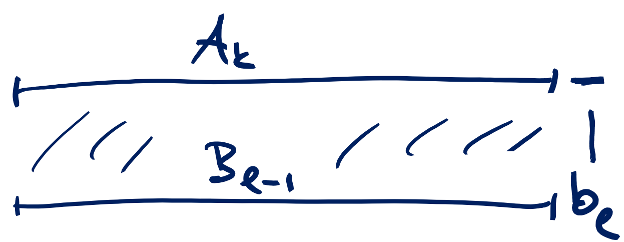
2. a_k is deleted:

   $D_{k,\ell} = D_{k-1,\ell} + c(a_k, \varepsilon)$



3. b_ℓ is inserted:

   $D_{k,\ell} = D_{k,\ell-1} + c(\varepsilon, b_\ell)$

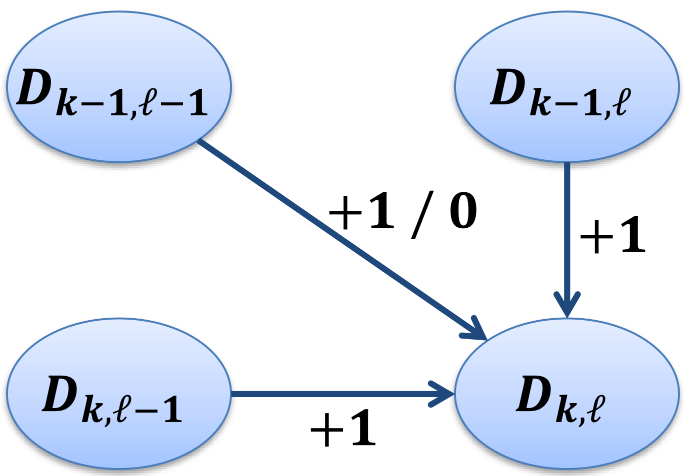


Computing the Edit Distance

- Recurrence relation (for $k, \ell \geq 1$)

$$D_{k,\ell} = \min \begin{Bmatrix} D_{k-1,\ell-1} & + c(a_k, b_\ell) \\ D_{k-1,\ell} & + c(a_k, \varepsilon) \\ D_{k,\ell-1} & + c(\varepsilon, b_\ell) \end{Bmatrix} = \underbrace{\min \begin{Bmatrix} D_{k-1,\ell-1} & + 1 \,/\, 0 \\ D_{k-1,\ell} & + 1 \\ D_{k,\ell-1} & + 1 \end{Bmatrix}}_{\textbf{unit cost model}}$$

- Need to compute $D_{i,j}$ for all $0 \leq i \leq k$, $0 \leq j \leq \ell$:



Recurrence Relation for the Edit Distance

Base cases:

$$\boldsymbol{D_{0,0} = D(\varepsilon, \varepsilon) = \underline{0}}$$
$$\boldsymbol{D_{0,j} = D(\varepsilon, B_j) = D_{0,j-1} + c(\varepsilon, b_j)}$$
$$\boldsymbol{D_{i,0} = D(A_i, \varepsilon) = D_{i-1,0} + c(a_i, \varepsilon)}$$

unit cost:

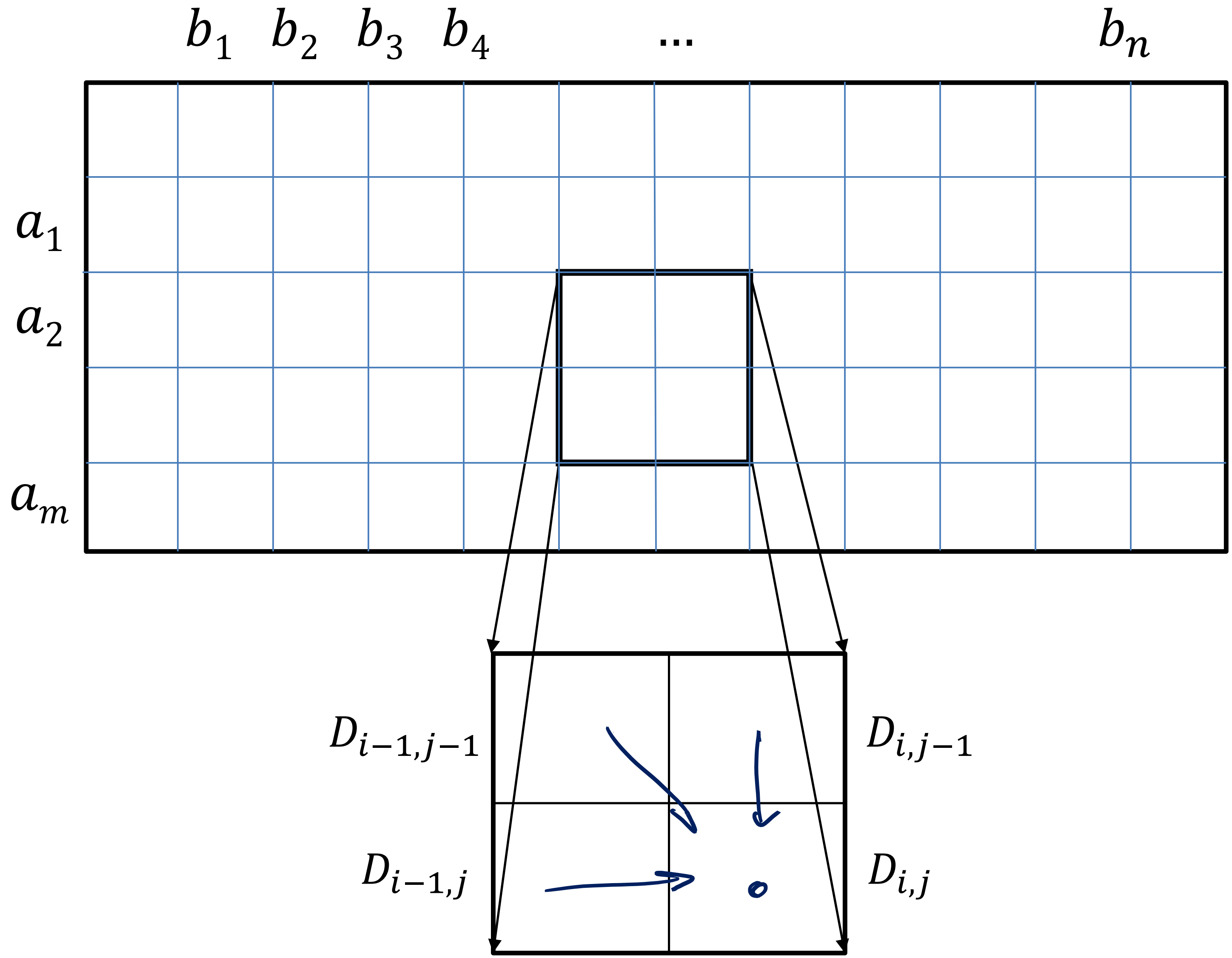
$D_{0,j} = j$

$D_{i,0} = i$

Recurrence relation:

$$\boldsymbol{D_{i,j} = \min \begin{Bmatrix} D_{k-1,\ell-1} + c(a_k, b_\ell) \\ D_{k-1,\ell} + c(a_k, \varepsilon) \\ D_{k,\ell-1} + c(\varepsilon, b_\ell) \end{Bmatrix}}$$

Order of solving the subproblems



Algorithm for Computing the Edit Distance

Algorithm *Edit-Distance*

Input: 2 strings $A = a_1 \dots a_m$ and $B = b_1 \dots b_n$

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

$D[0,0] := 0;$

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**

$$D[i, j] := \min \left\{ \begin{array}{ll} D[i-1, j] & +1 \\ D[i, j-1] & +1 \\ D[i-1, j-1] & + c\left(a_i, b_j\right) \end{array} \right\};$$

Example

		a	b	c	c	a
	0	1	2	3	4	5
b	1	1	1	2	3	4
a	2					
b	3					
d	4					
a	5					

Edit Operations

		a	b	c	c	a
	0	1	2	3	4	5
b	1	1	1	2	3	4
a	2	1	2	2	3	3
b	3	2	1	2	3	4
d	4	3	2	2	3	4
a	5	4	3	3	3	3

Computing the Edit Operations

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

```
if i = 0 and j = 0 then return empty list

if i ≠ 0 and D[i, j] = D[i − 1, j] + 1 then
    return Edit-Operations(i − 1, j) ∘ „delete a_i“

else if j ≠ 0 and D[i, j] = D[i, j − 1] + 1 then
    return Edit-Operations(i, j − 1) ∘ „insert b_j“

else // D[i, j] = D[i − 1, j − 1] + c(a_i, b_j)
    if a_i = b_i then return Edit-Operations(i − 1, j − 1)
    else return Edit-Operations(i − 1, j − 1) ∘ „replace a_i by b_j“
```

Initial call: *Edit-Operations*(*m*,*n*)

Edit Operations

		a	b	c	c	a
	0	1	2	3	4	5
b	1	1	1	2	3	4
a	2	1	2	2	3	3
b	3	2	1	2	3	4
d	4	3	2	2	3	4
a	5	4	3	3	3	3

b a b d – a
| | | | | |
– a b c c a
x x x

Edit Distance: Summary

- Edit distance between two strings of length m and n can be computed in $O(mn)$ time.

- Obtain the edit operations:
  - for each cell, store which rule(s) apply to fill the cell
  - track path backwards from cell (m, n)
  - can also be used to get all optimal "alignments"

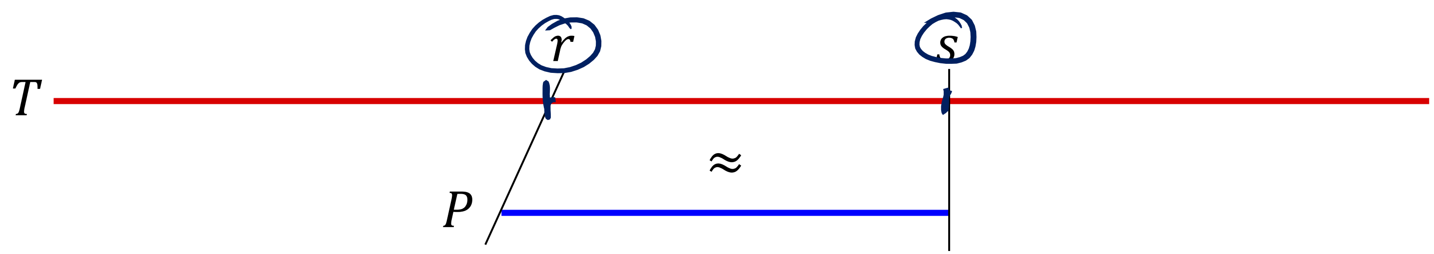
- Unit cost model:
  - interesting special case
  - each edit operation costs 1

Approximate String Matching

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

Goal: Find an interval $[r, s]$, $1 \leq r \leq s \leq n$ such that the sub-string $T_{r,s} \coloneqq t_r \dots t_s$ is the one with highest similarity to the pattern P:

$$\arg\min_{1 \leq r \leq s \leq n} D\left(T_{r,s}, P\right)$$



Approximate String Matching

Naive Solution:

for all $\underline{1 \leq r \leq s \leq n}$ **do** $\mathcal{O}(n^2)$ comb.

compute $\underline{D(T_{r,s}, P)}$ <u>cost:</u> $\mathcal{O}((s-r) \cdot m) = \mathcal{O}(n \cdot m)$

choose the minimum

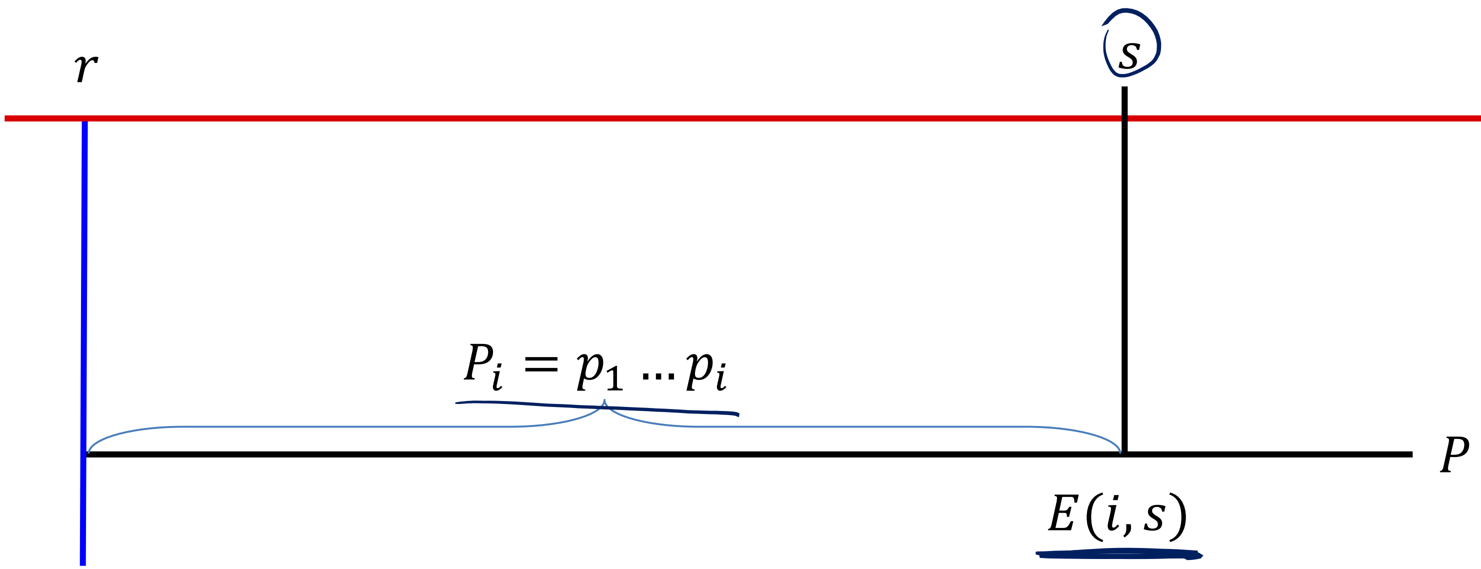
$\rightarrow$ <u>overall:</u> $\mathcal{O}(n^3 \cdot m)$

<u>unit cost:</u> can be improved to $\mathcal{O}(n \cdot m^3)$

Approximate String Matching

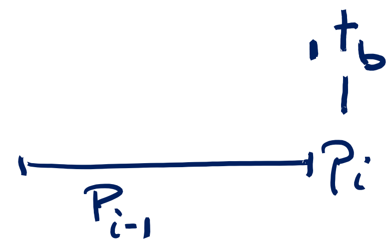
A related problem:

- For each position s in the text and each position i in the pattern compute the minimum edit distance $E(i,s)$ between $P_i = p_1 \dots p_i$ and any substring $T_{r,s}$ of T that ends at position s.



Approximate String Matching

Three ways of ending optimal alignment between T_b and P_i:



1. t_b is replaced by p_i:

$$E_{b,i} = E_{b-1,i-1} + c(t_b, p_i)$$

2. t_b is deleted:

$$E_{b,i} = E_{b-1,i} + c(t_b, \varepsilon)$$

3. p_i is inserted:

$$E_{b,i} = E_{b,i-1} + c(\varepsilon, p_i)$$

Approximate String Matching

Recurrence relation (unit cost model):

$$E_{b,i} = \min \begin{Bmatrix} E_{b-1,i-1} & +1\ /\ 0 \\ E_{b-1,i} & +1 \\ E_{b,i-1} & +1 \end{Bmatrix}$$

Base cases:

$$E_{0,0} = 0$$
$$E_{0,i} = i$$
$$\underline{E_{i,0} = 0}$$

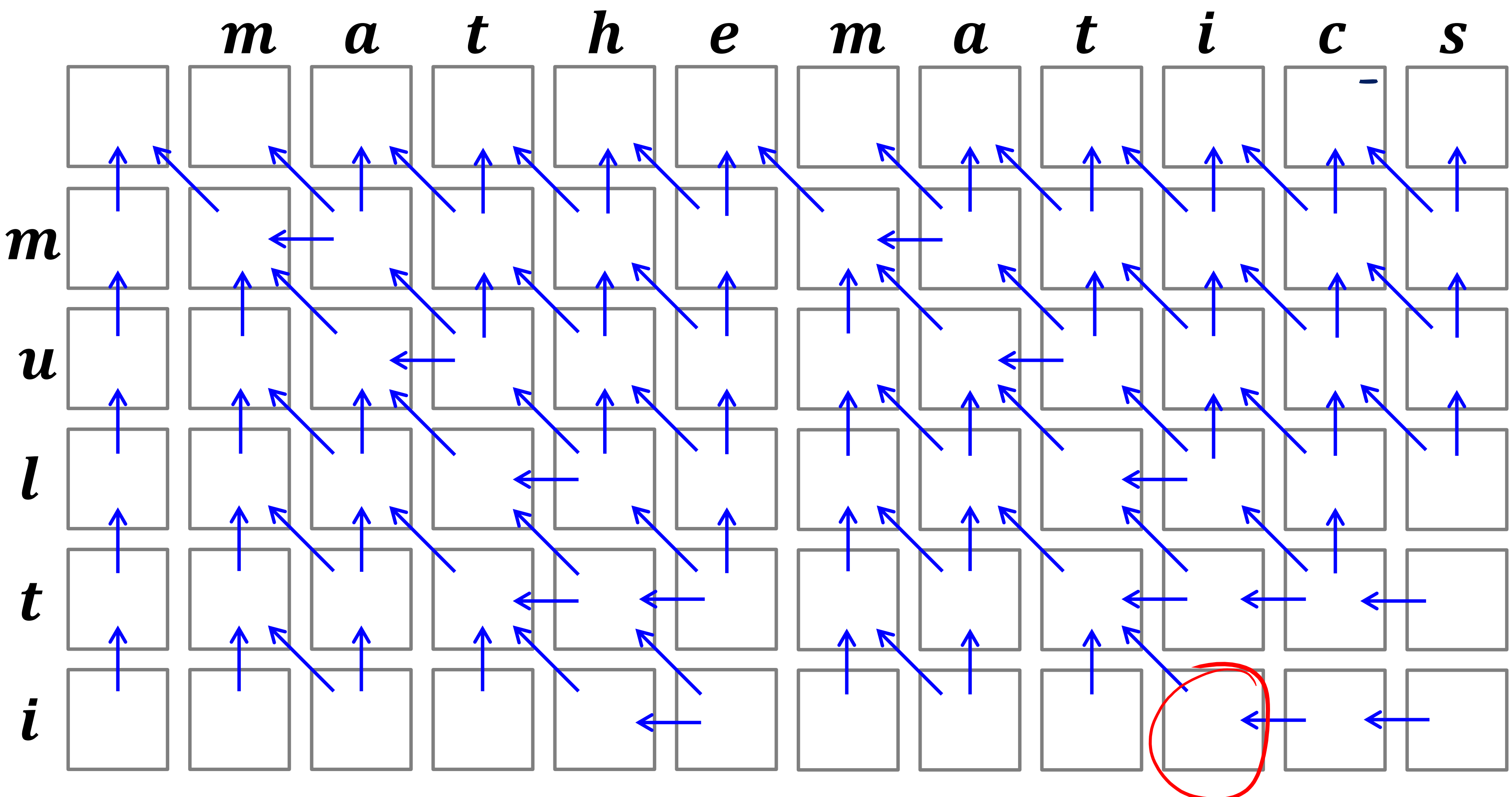
r i

$T_{r,i}$

ε

Example

$E(b,i)$



mathe|ma-ti|cs
multi

Approximate String Matching

- Optimal matching consists of optimal sub-matchings

- Optimal matching can be computed in $O(mn)$ time

- Get matching(s):
  - Start from minimum entry/entries in bottom row
  - Follow path(s) to top row

- Algorithm to compute $E(b, i)$ identical to edit distance algorithm, except for the initialization of $E(b, 0)$

Related Problems in Bioinformatics

Sequence Alignment:

Find optimal alignment of two given DNA, RNA, or amino acid sequences.

```
G A – C G G A T T A G
| |
G A T C G G A A T – G
```

Global vs. Local Alignment:

- *Global alignment*: find optimal alignment of 2 sequences
- *Local alignment*: find optimal alignment of sequence 1 (patter) with sub-sequence of sequence 2 (text)