582670 Algorithms for Bioinformatics

Lecture 4: Dynamic Programming and Sequence Alignment

24.9.2015

Adapted from slides by Alexandru Tomescu, Leena Salmela and Veli Mäkinen

Sequence Similarity

Many applications of comparing and finding sequences that are similar but not identical

- Motif finding
- DNA sequencing
  - Comparing reads against other reads or reference genome(s)
- Finding similar genes/proteins (homologs)
  - Hints about function or structure
  - Estimating evolutionary distance

Measuring Sequence Similarity

- To determine how similar two sequences are, we need either a similarity measure or a distance measure
- We have seen the Hamming distance that counts mismatches

A	G	G	T	A	C	
A	C	G	T	C	C	
	1			+1		=2

- A possible similarity measure could count matches

A	G	G	T	A	C	
A	C	G	T	C	C	
1		+1	+1		+1	=4

What is wrong with Hamming distance?

- Only for sequences of the same length
- Does not allow insertions or deletions
- Example: Hamming distance is high

G	G	A	T	A	C	
A	G	G	A	T	C	
1	+1	+1	+1	+1		=5

but the sequences are clearly similar

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

Alignments with Indels

- Indel = insertion or deletion
- Example

-	G	G	A	T	A	C
A	G	G	-	T	C	C
I	M	M	D	M	S	M

- M = match
  S = Substitution
  I = Insertion
  D = Deletion

Scoring alignments

- Many possible alignments
- Which one is better?

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

S  M  M  S  D  M
```

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

D  M  M  D  M  I  M
```

- We need a score for alignments
- The best score over all possible alignments can be used as a similarity or distance between the strings

Longest Common Subsequence

- Subsequence of a string is a subset of the letters in the same order
- Example: GGTC is a subsequence of GGATAC and AGGTCC
- The matching letters in an alignment form a *common subsequence*

-	G	G	A	T	A	C	-
A	G	G	-	T	-	C	C
	G	G		T		C	
	1	+1		+1		+1	=4

- The number of matches, i.e. the length of the common subsequence, can be used as a score for an alignment
- The maximum score over all alignments, i.e, the length of the *longest common subsequence* (LCS), is a string similarity
- The LCS itself can be of interest

What is wrong with LCS?

- Which one is the better alignment?

G	C	C	A	G	G	G	
G	G	G	A	T	T	G	
1			+1			+1	= 3

G	C	C	A	G	G	-	-	-	G	
G	-	-	-	G	G	A	T	T	G	
1				+1	+1				+1	= 4

- Indels are often rare in biological sequences

Edit distance

- We can use the number of substitutions and indels as a score

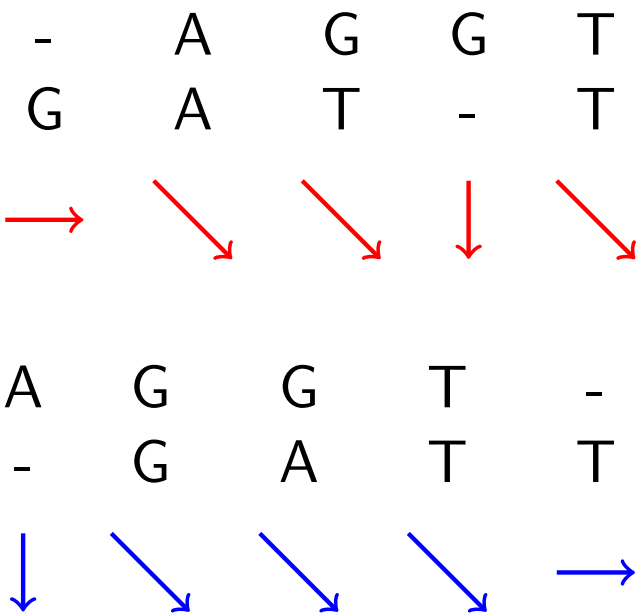
G	C	C	A	G	G	G	
G	G	G	A	T	T	G	
	1	+1		+1	+1		= 4

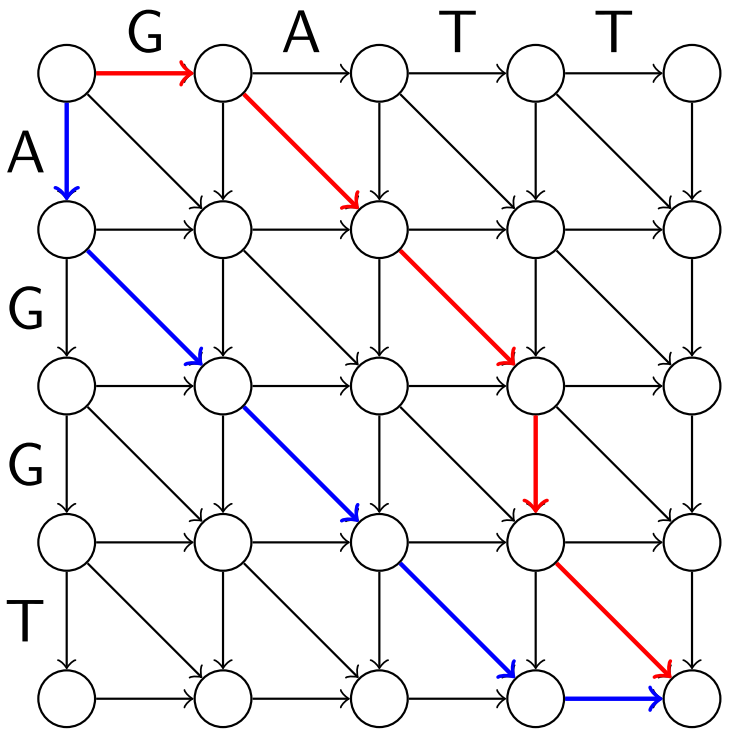
G	C	C	A	G	G	-	-	-	G
G	-	-	-	G	G	A	T	T	G
	1	+1	+1			+1	+1	+1	= 6

- The *minimum* score over all alignments is the *edit distance* of the strings
- Alternatively defined as the minimum number of edit operations (substitutions and indels) to convert one string into the other
- Also known as the Levenshtein distance

Alignment Graph

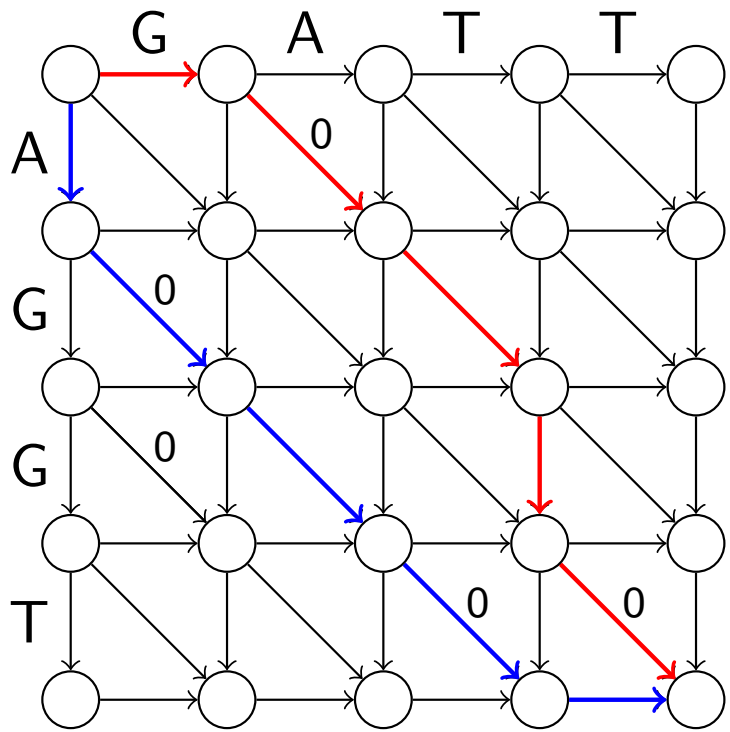
- Each alignment corresponds to a path in a rectangular graph





Edit distance as a shortest path problem

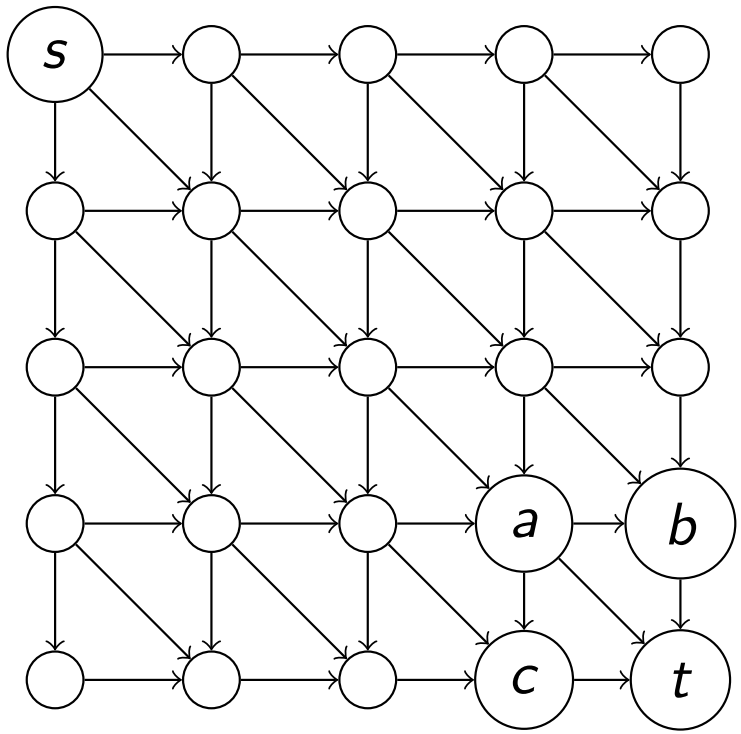
- Assign weights to the edges according to the scoring scheme
- For edit distance
  - Match edges have weight 0
  - Other edges have weight 1
  - (Only 0s shown here)
- The length/weight of a path is the sum of edge weights
  - The same as the score of the alignment
  - Example: both marked paths have length 3



- Edit distance is the length of the shortest path

Finding shortest path

- We want to compute the shortest path from the source node s to the sink node t
- Every path from s to t must go through one of the neighboring nodes a, b and c
- Basic idea: find the shortest path from s to each of the nodes a, b and c, add the weights of the edges from a, b and c to t and take the minimum.



Finding shortest path

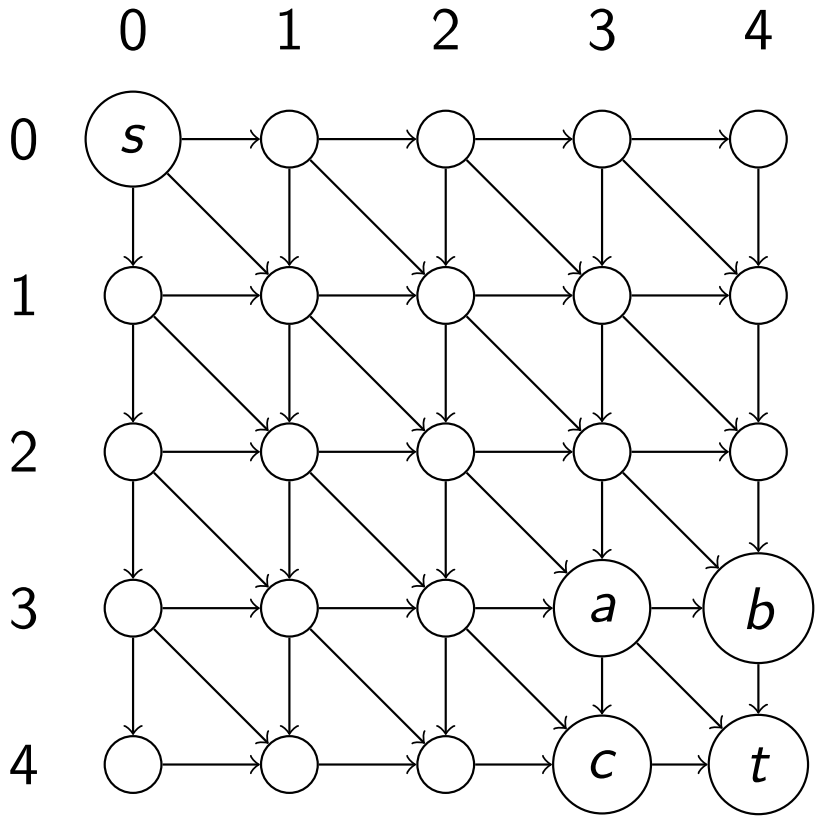
- Define
  - $w(u, v)$ = weight of the edge from node u to node v
  - $d(u, v)$ = length of shortest path from u to v
- We compute $d(s, t)$ using the formula

$$d(s,t) = \min \begin{cases} d(s,a) + w(a,t) \\ d(s,b) + w(b,t) \\ d(s,c) + w(c,t) \end{cases}$$

- The values $d(s, a)$, $d(s, b)$ $d(s, c)$ are computed similarly from their neighbors

Naming nodes by coordinates

- Take advantage of the grid structure by naming the nodes by a pair of coordinates
- $s = (0, 0)$
- $t = (m, n)$ (here $m = n = 4$)
- $a = (m - 1, n - 1)$
- $b = (m - 1, n)$
- $c = (m, n - 1)$



Shortest path length

- Let $D(i,j) = d(s,(i,j))$ be the shortest path length from s to (i,j)
- We compute $D(i,j)$ using the formula

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + w((i-1,j-1),(i,j)) \\ D(i-1,j) + w((i-1,j),(i,j)) \\ D(i,j-1) + w((i,j-1),(i,j)) \end{cases}$$

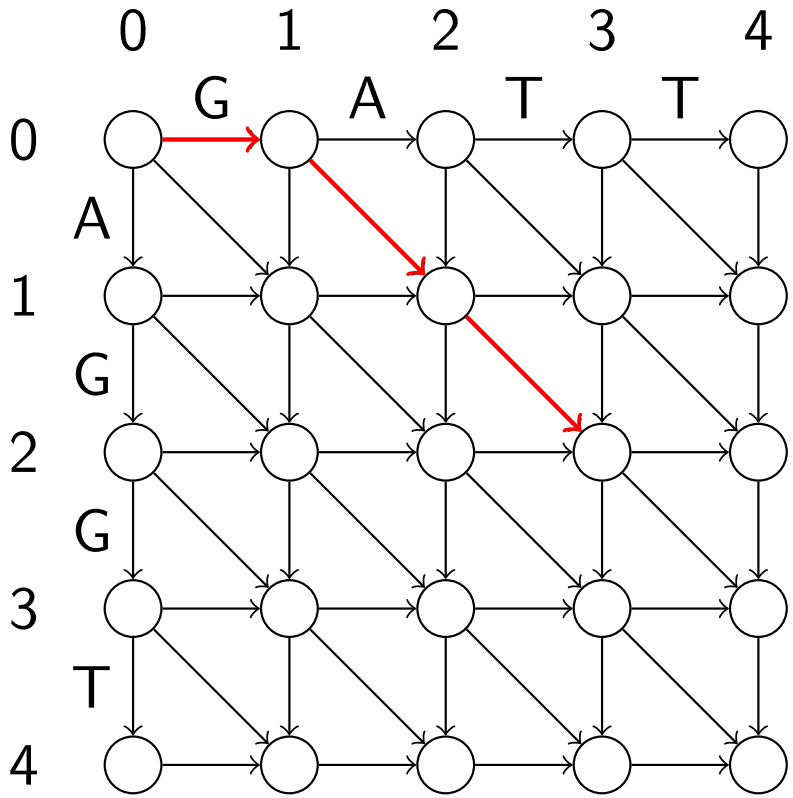
  when $i > 0$ and $j > 0$
- Separate formulas for zero coordinates

$$\begin{aligned} D(i,0) &= D(i-1,0) + w((i-1,0),(i,0)) \\ D(0,j) &= D(0,j) + w((0,j-1),(0,j)) \\ D(0,0) &= 0 \end{aligned}$$

Subpath is prefix alignment

- A (sub)path from s to (i, j) corresponds to an alignment of string prefixes of lengths i and j
- Example: alignment for path to $(2, 3)$

-	A	G
G	A	T



Edit distance computation

- Consider strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$
- We want to compute the edit distance $d_L(A, B)$

- $D(i, j)$ is the length optimal path to node (i, j)
- This is the same as the score of the optimal alignment between $a_1 \ldots a_i$ and $b_1 \ldots b_j$, i.e.,

$$D(i, j) = d_L(a_1 \ldots a_i, b_1 \ldots b_j)$$

and

$$D(m, n) = d_L(A, B)$$

- We can compute the edit distance using the equations we saw earlier
  - Edge weights $w(\cdot, \cdot)$ are either 0 (match) or 1

Edit distance equations

- We want to compute $D(m, n)$
- When both $i > 0$ and $j > 0$, use

$$D(i,j) = \min \begin{cases} D(i-1, j-1) + (\text{ if } a_i = b_j \text{ then } 0 \text{ else } 1) \\ D(i-1, j) + 1 \\ D(i, j-1) + 1 \end{cases}$$

- Otherwise use

$$D(i,0) = D(i-1,0) + 1 = i$$
$$D(0,j) = D(0,j-1) + 1 = j$$
$$D(0,0) = 0$$

Recursive computation

- The main formula:

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + (\text{ if } a_i = b_j \text{ then 0 else 1}) \\ D(i-1,j) + 1 \\ D(i,j-1) + 1 \end{cases}$$

- A natural way to implement this is using recursion to solve the subproblems $D(i-1,j-1)$, $D(i-1,j)$ and $D(i,j-1)$
- This is very inefficient because each recursive call generates three new calls
- Some values $D(i,j)$ are computed many times during the recursive computation
- A possible solution is to store each $D(i,j)$ value when it is computed for the first time and later use the stored value instead of using recursion
  - This is known as memoization

Dynamic Programming

- The idea of dynamic programming is similar to memoization:
  store solutions to subproblems
- The difference is how the computation is organized:
  subproblems are computed and stored *before* they are needed for the first time
- Requires finding an appropriate order for computing the subproblems
- Often leads to simple and efficient code

Example: Computing Fibonacci numbers

- Fibonacci numbers are defined by the equation:

$$F(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

- Computing $F(6)$ by recursion

$$\begin{aligned} F(6) &= F(5) + F(4) \\ &= (F(4) + F(3)) + (F(3) + F(2)) \\ &= (F(3) + F(2)) + (F(2) + F(1)) + (F(2) + F(1)) + 1 \\ &= (F(2) + F(1)) + 1 + 1 + 1 + 1 + 1 + 1 \\ &= 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\ &= 8 \end{aligned}$$

- Each recursive call generates two new calls until the calls reach the bottom
- The call $F(3)$ is made three separate times, the call $F(2)$ five times

Example: Computing Fibonacci numbers

- Fibonacci numbers are defined by the equation:

$$F(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

- Computing $F(6)$ by dynamic programming

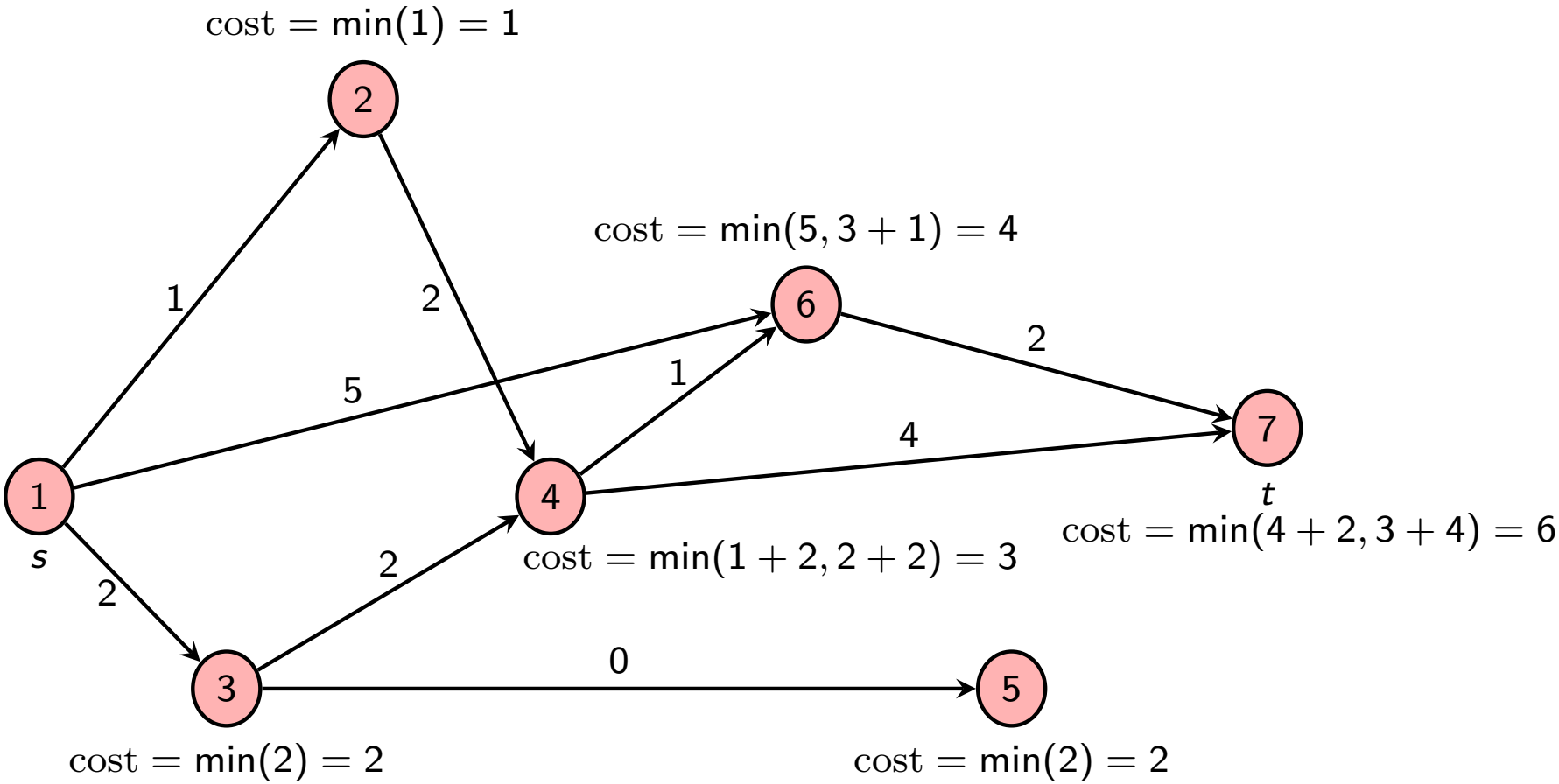
$$\begin{aligned} F[1] &= F[2] = 1 \\ F[3] &= F[2] + F[1] = 1 + 1 = 2 \\ F[4] &= F[3] + F[2] = 2 + 1 = 3 \\ F[5] &= F[4] + F[3] = 3 + 2 = 5 \\ F[6] &= F[5] + F[4] = 5 + 3 = 8 \end{aligned}$$

- The square brackets indicate that the values are stored in an array
  - In fact, only the two previous values need to be kept
- Here the ordering of the subproblems is simple

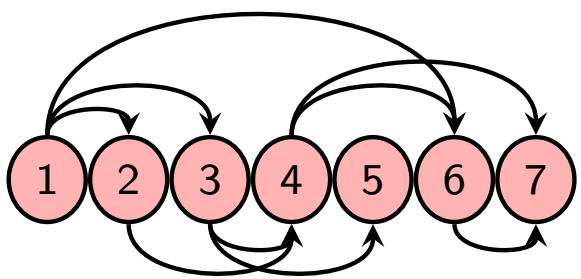
Example: Shortest path in a DAG

DAG=directed acyclic graph

Shortest path from s to t?



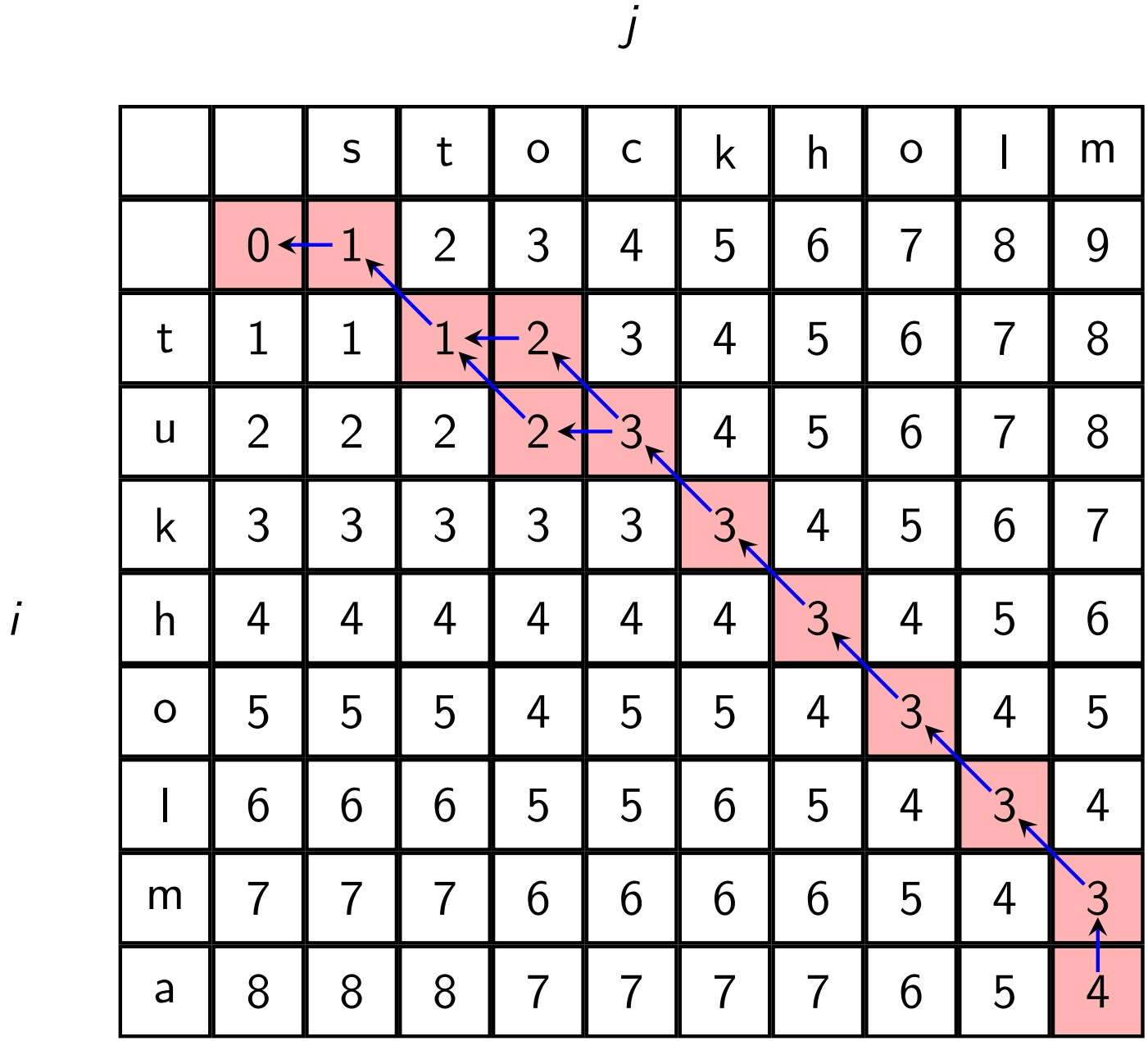
Topological sort to order subproblems:



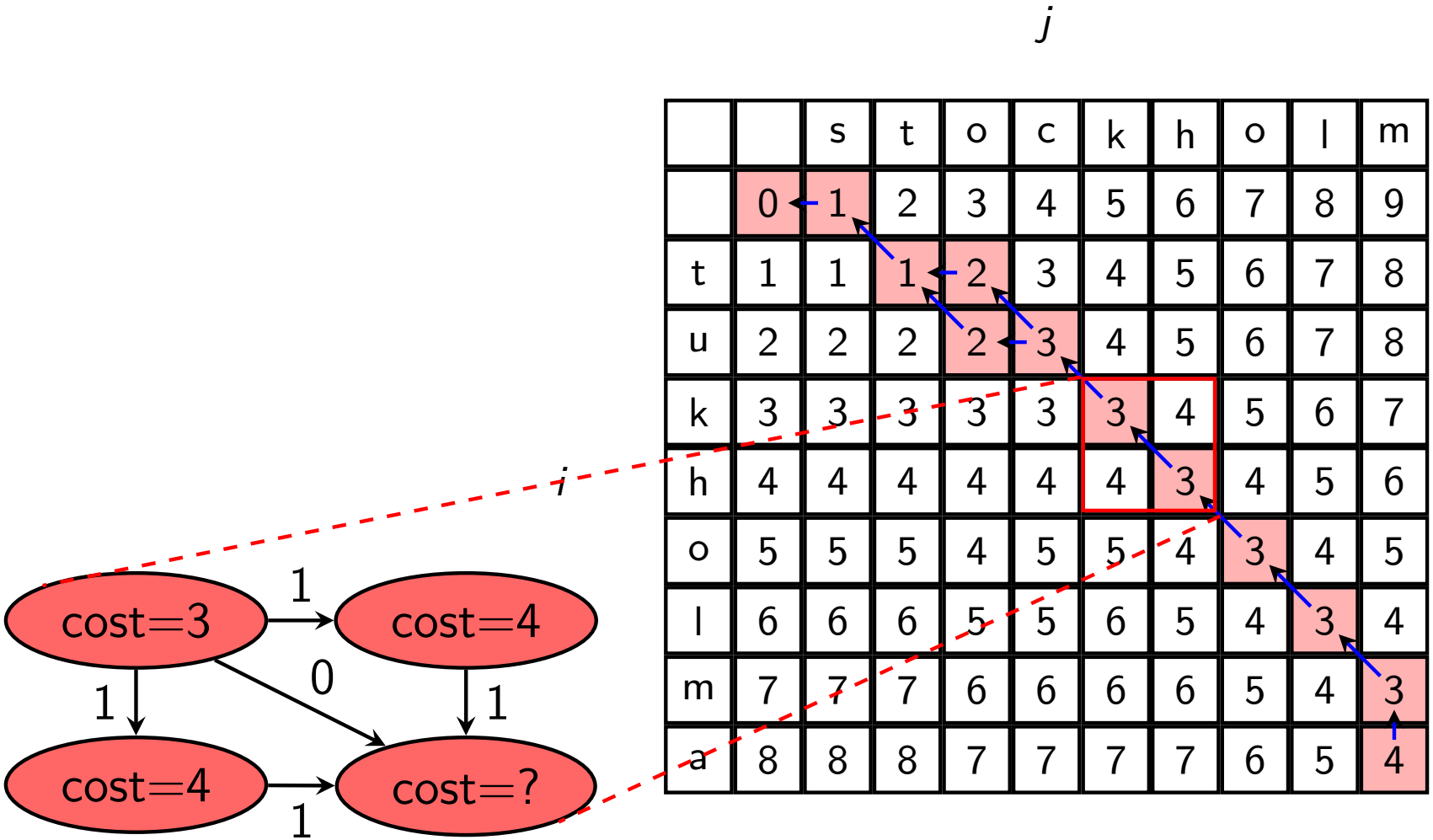
Edit distance by dynamic programming

- Compute all the values $D(i, j)$
- Start from small values of i and j and proceed to bigger values
  - For example, column-by-column or row-by-row
- Store computed values in a two-dimensional array $D[0..m, 0..n]$
  - D is often called the edit distance matrix
- No recursive calls; All subproblem values are obtained from the matrix

Edit distance matrix: example



Edit distance matrix as a graph



		s	t	o	c	k	h	o	l	m
	0	1	2	3	4	5	6	7	8	9
t	1	1	1	2	3	4	5	6	7	8
u	2	2	2	2	3	4	5	6	7	8
k	3	3	3	3	3	3	4	5	6	7
h	4	4	4	4	4	4	3	4	5	6
o	5	5	5	4	5	5	4	3	4	5
l	6	6	6	5	5	6	5	4	3	4
m	7	7	7	6	6	6	6	5	4	3
a	8	8	8	7	7	7	7	6	5	4

$$\text{cost} = \min(3 + 0, 4 + 1, 4 + 1) = 3$$

Finding optimal alignments

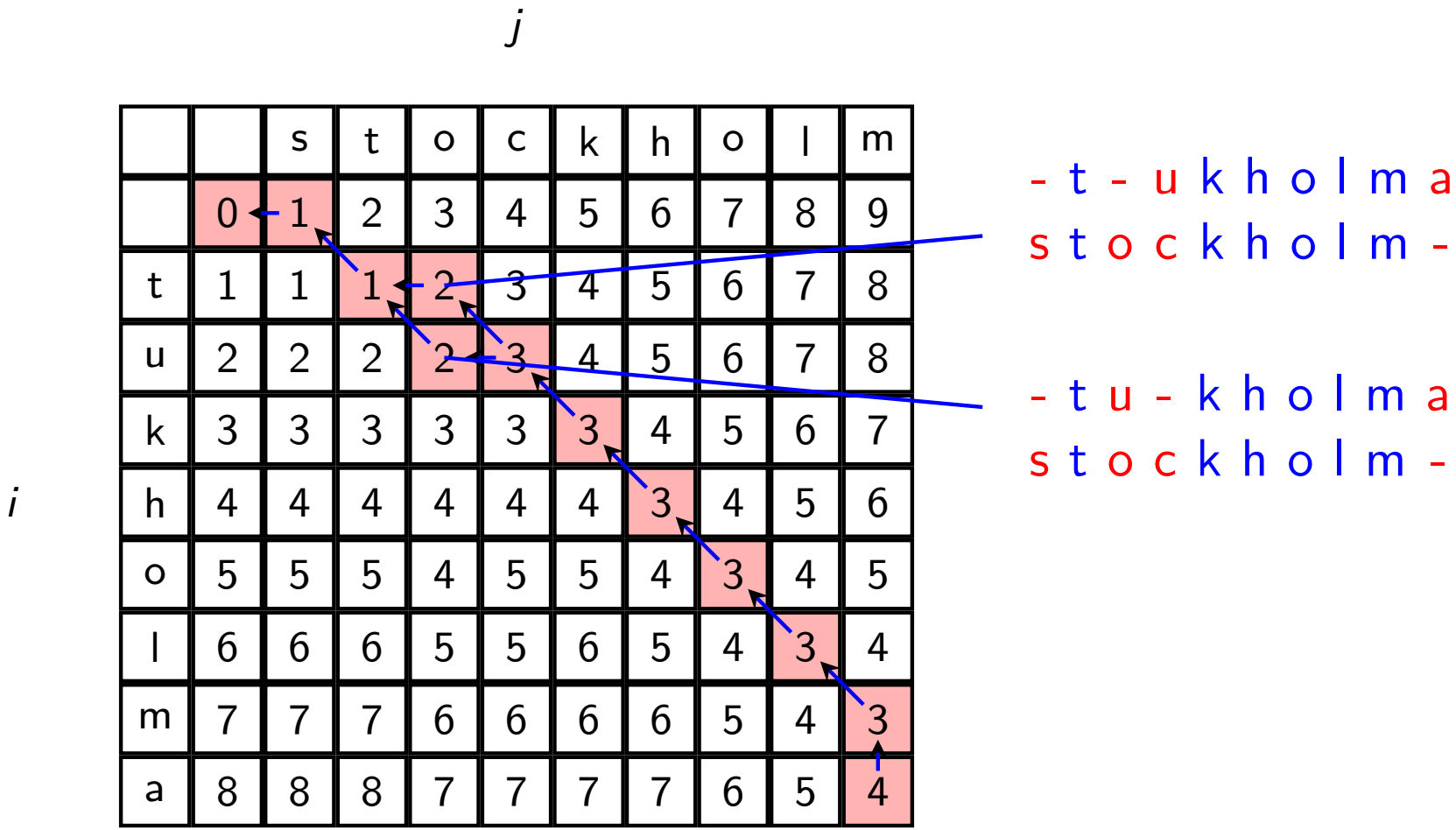
One alignment:

- Store pointer to each cell telling from which cell the minimum was obtained.
- Follow the pointers from (m, n) to $(0, 0)$.
- Reverse the list.

All alignments:

- Backtrack from (m, n) to $(0, 0)$ by checking at each cell (i, j) on the path whether the value $D[i, j]$ could have been obtained from cell $(i, j-1)$, $(i-1, j-1)$, or $(i-1, j)$.
- Explore all directions.
  - All three directions possible.
  - Exponential number of optimal paths in the worst case.

Optimal alignments: example



Variations and generalizations

- The problem we have been looking at is known as the global alignment problem
- There are other alignment problems:
  - Approximate string matching or fitting alignment
  - Overlap alignment
  - Local alignment

Global alignment

- Input: Two strings A and B
- Output: Best alignment between A and B

- Example applications
  - Compare two motifs
  - Compare two genes/proteins

Fitting alignment or approximate string matching

- Input: Two strings P and T
- Output: The substring S of T with the best alignment between P and S

- Example applications
  - Compare a read against a reference genome
  - Compare a gene against a genome to find similar genes

Overlap alignment

- Input: Two strings A and B
- Output: The suffix S of A and the prefix P of B with the best alignment between S and P

- Example application
  - Find overlaps between reads

Local alignment

- Input: Two strings S and T
- Output: The substring A of S and the substring B of T with the best alignment between A and B

- Example application
  - Find partial similarities between genes/proteins (e.g., conserved regions)

Approximate string matching

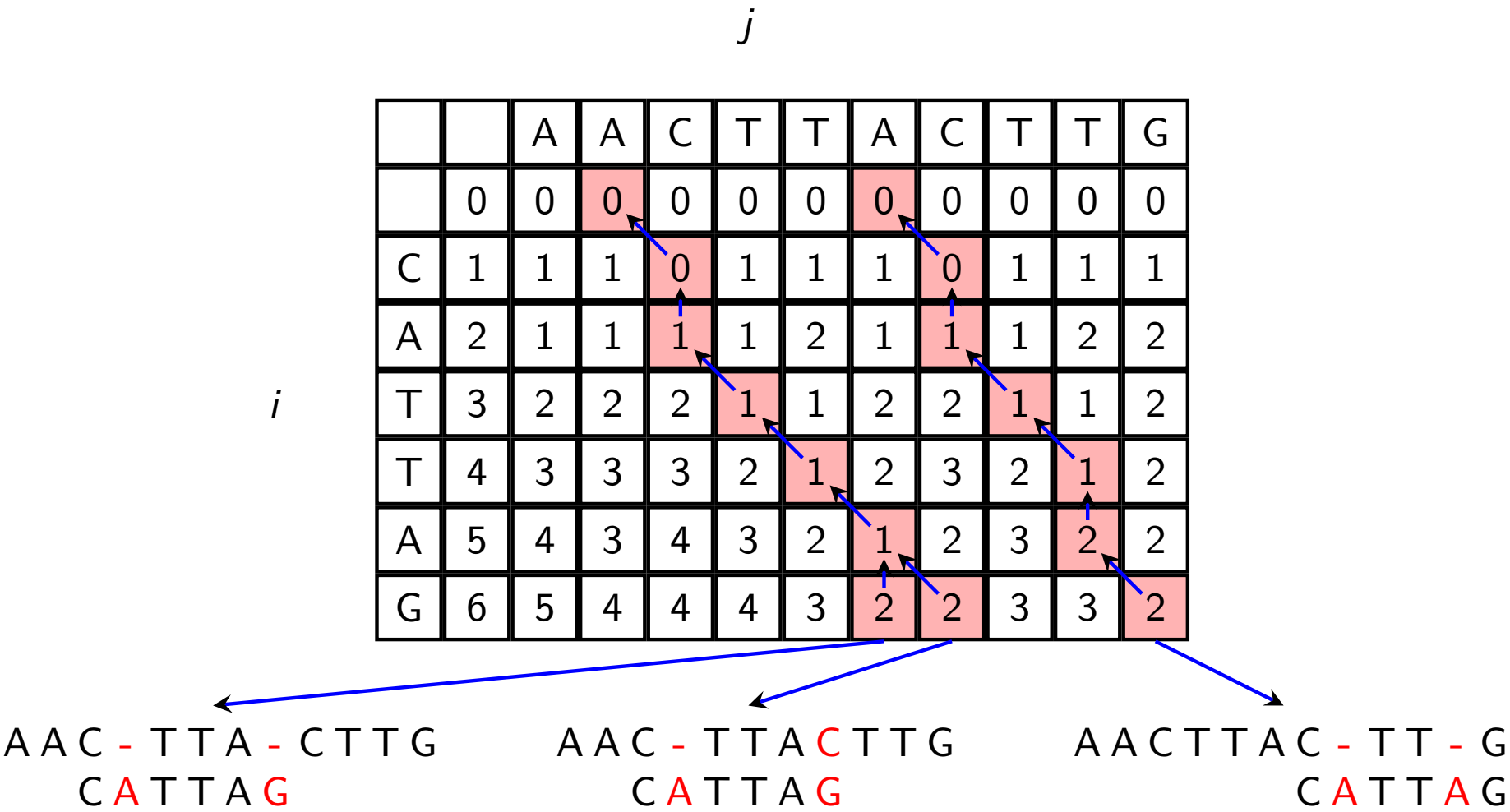
- k-errors problem
  - Input: Two strings $P[1..m]$ and $T[1..n]$ and an integer k
  - Output: Substrings S of T such that $d_L(P, S) \leq k$

- Can be solved using a "zero first row trick"
  - Compute $D[0..m, 0..n]$ as when computing the edit distance except

$$D[0, j] = 0 \text{ for all } j$$

  - $D[i, j]$ then equals the minimum number of edits to convert $P[1, i]$ into *some suffix of* $T[1, j]$.
  - If $D[m, j] = k' \leq k$ then $d_L(P, S) = k'$ for some substring S of T ending at position j in T

Approximate string matching: example



		A	A	C	T	T	A	C	T	T	G
	0	0	0	0	0	0	0	0	0	0	0
C	1	1	1	0	1	1	1	0	1	1	1
A	2	1	1	1	1	2	1	1	1	2	2
T	3	2	2	2	1	1	2	2	1	1	2
T	4	3	3	3	2	1	2	3	2	1	2
A	5	4	3	4	3	2	1	2	3	2	2
G	6	5	4	4	4	3	2	2	3	3	2

Problem with edit distance

- Edit distance is good scoring system for global alignment and approximate string matching but not for local alignment or overlap alignment
- Edit distance favors short overlaps

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

   1           +1        = 2
```

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

                                  = 0
```

General scoring scheme

- $\delta(a, b)$ is the score for changing symbol a into b
  - If $a = b$ this the score of a match
- $\delta(a, -)$ is the score of deleting a
- $\delta(-, b)$ is the score of inserting b

- Typically
  - $\delta(a, b) = 1$ if $a = b$
  - $\delta(a, b) = -\mu$ if $a \neq b$
  - $\delta(a, -) = \delta(-, b) = -\sigma$
- Similarity measure
  - Best alignment has maximal score
  - Find longest path in alignment graph

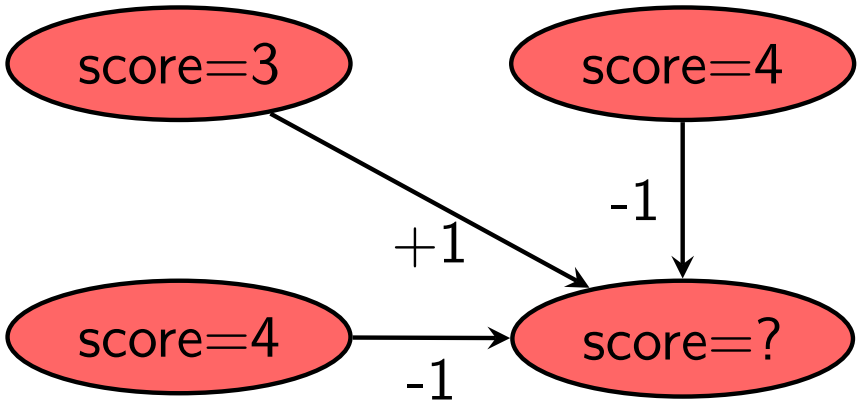
Scoring scheme for local and overlap alignment

- Positive score for good things and negative score for bad things is required in local and overlap alignment
- For example $\mu = \sigma = 1$

C	G	G	A	G	-	T		
	T	G	A	G	C	T	A	
-1	+1	+1	+1	-1	+1			$= +2$

C	G	G	A	G	T						
					T	G	A	G	C	T	A
					1						$= +1$

Global alignment



$$\text{score} = \max(3 + 1, 4 - 1, 4 - 1) = 4$$

$$S[i,j] = \max \begin{cases} S[i-1,j-1] + \delta(a_i, b_j) \\ S[i-1,j] + \delta(a_i, -) \\ S[i,j-1] + \delta(-, b_j) \end{cases}$$

Global alignment: Example

$\delta(a_i, b_j) = 1$, if $a_i = b_j$

$\delta(a_i, b_j) = -1$, otherwise

$\delta(a_i, -) = \delta(-, b_j) = -1$

j

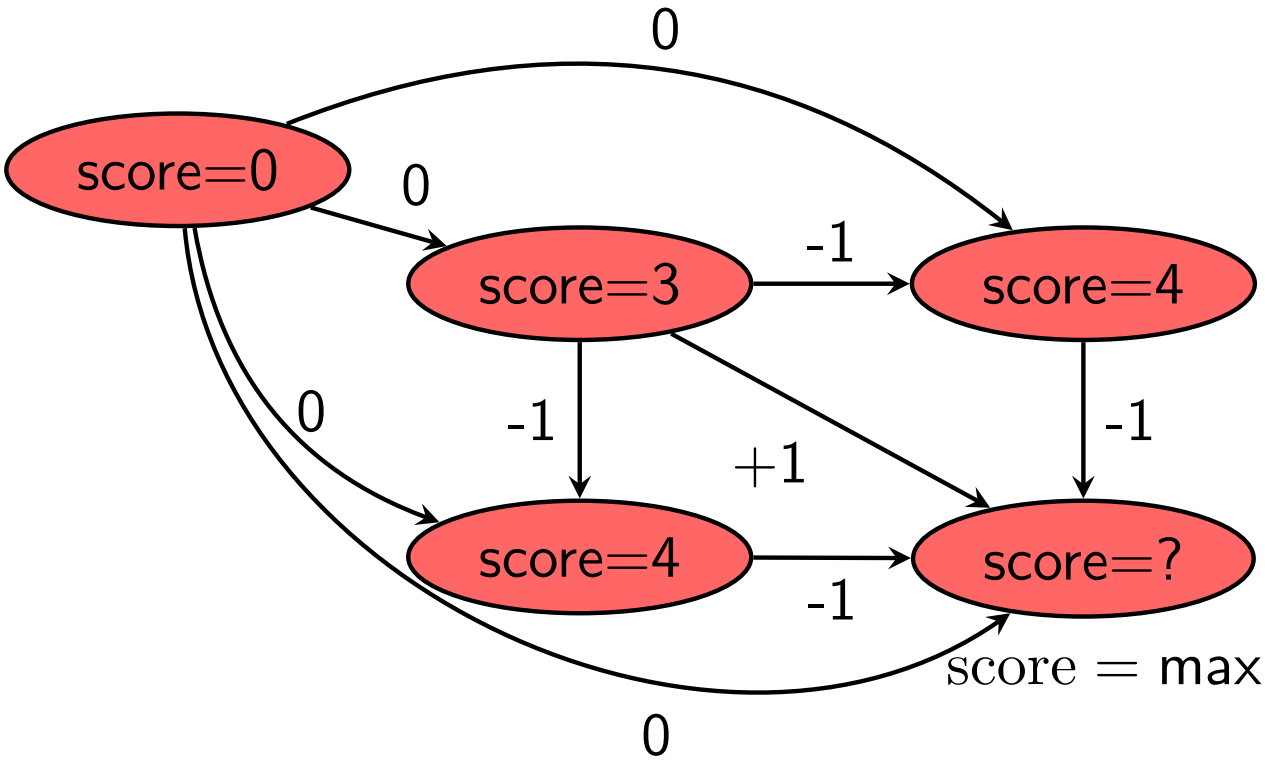
		A	A	C	T	T	A	C	T	T	G
	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10
C	-1	-1	-2	-1	-2	-3	-4	-5	-6	-7	-8
A	-2	0	0	-1	-2	-3	-2	-3	-4	-5	-6
T	-3	-1	-1	-1	0	-1	-2	-3	-2	-3	-4
T	-4	-2	-2	-2	0	+1	0	-1	-2	-1	-2
A	-5	-3	-1	-2	-1	0	+2	+1	0	-1	-2
G	-6	-4	-2	-2	-2	-1	+1	+1	0	-1	0

i

Local alignment

- Heaviest/longest path beginning anywhere and ending anywhere
- Beginning anywhere
  - Add 0 weight edge from source node $(0, 0)$ to every other node
  - Consider all paths starting from $(0, 0)$
  - No negative heaviest path scores because we can always choose the zero weight path from the source as the maximum
- Ending anywhere
  - Find maximum score over all nodes

Local alignment



$$\text{score} = \max(0, 3+1, 4-1, 4-1) = 4$$

$$S[i,j] = \max \begin{cases} 0 \\ S[i-1,j-1] + \delta(a_i, b_j) \\ S[i-1,j] + \delta(a_i, -) \\ S[i,j-1] + \delta(-, b_j) \end{cases}$$

Local alignment: Example

$\delta(a_i, b_j) = 1$, if $a_i = b_j$

$\delta(a_i, b_j) = -1$, otherwise

$\delta(a_i, -) = \delta(-, b_j) = -1$

j

i

		A	A	C	T	T	A	C	T	T	G
	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	1	0	0	0	1	0	0	0
A	0	1	1	0	0	0	1	0	0	0	0
T	0	0	0	0	1	1	0	0	1	1	0
T	0	0	0	0	1	2	1	0	1	2	1
A	0	1	1	0	0	1	3	2	1	1	1
G	0	0	0	0	0	0	2	2	1	0	2