Dynamic Programming: Edit Distance

Outline

1. DNA Sequence Comparison and CF
2. Change Problem
3. Manhattan Tourist Problem
4. Longest Paths in Graphs
5. Sequence Alignment
6. Edit Distance

Section 5: Sequence Alignment

Back to Biology: Sequence Alignment

- Recall that our original problem was to fit a similarity score on two DNA sequences:

- To do this we will use what is called an **alignment matrix**.

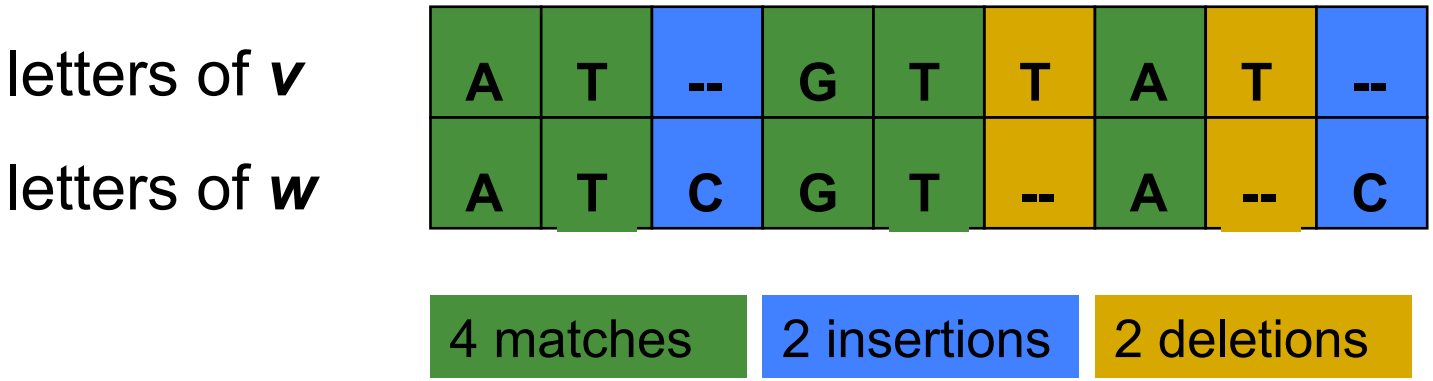
Alignment Matrix: Example

- Given 2 DNA sequences **v** and **w** of length m and n:

v: ATCTGAT $m=7$

w: TGCATA $n=6$

- Example of Alignment : 2 * k matrix ($k > m, n$)



Common Subsequence

- Given two sequences

  $\boldsymbol{v} = v_1 v_2 \ldots v_m$ and $\boldsymbol{w} = w_1 w_2 \ldots w_n$

  a **common subsequence** of $\boldsymbol{v}$ and $\boldsymbol{w}$ is a sequence of positions in $\boldsymbol{v}$: $1 \leq i_1 < i_2 < \ldots < i_t \leq m$ and a sequence of positions in $\boldsymbol{w}$: $1 \leq j_1 < j_2 < \ldots < j_t \leq n$ such that the i_t-*th* letter of $\boldsymbol{v}$ is equal to the j_t-*th* letter of $\boldsymbol{w}$.

- Example: $\boldsymbol{v}$ = ATGCCAT, $\boldsymbol{w}$ = TCGGGCTATC. Then take:
  - $i_1 = 2, i_2 = 3, i_3 = 6, i_4 = 7$
  - $j_1 = 1, j_2 = 3, j_3 = 8, j_4 = 9$
  - This gives us that the common subsequence is **TGAT**.

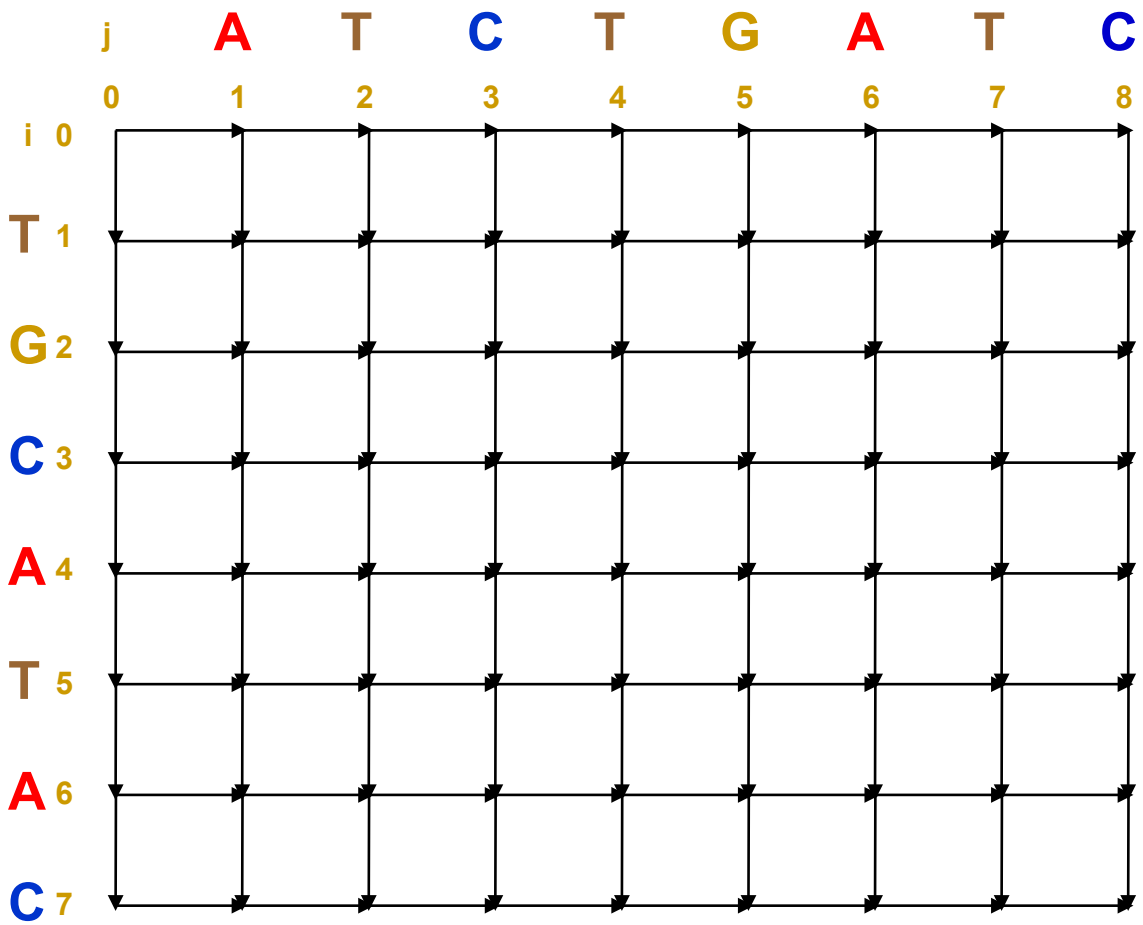
Longest Common Subsequence

- Given two sequences $\boldsymbol{v} = v_1 v_2 \ldots v_m$ and $\boldsymbol{w} = w_1 w_2 \ldots w_n$ the ***Longest*** **Common Subsequence** (LCS) of $\boldsymbol{v}$ and $\boldsymbol{w}$ is a sequence of positions in $\boldsymbol{v}$: $1 \leq i_1 < i_2 < \ldots < i_T \leq m$ and a sequence of positions in $\boldsymbol{w}$: $1 \leq j_1 < j_2 < \ldots < j_T \leq n$ such that the i_t*-th* letter of $\boldsymbol{v}$ is equal to j_t*-th* letter of $\boldsymbol{w}$ **and** $\boldsymbol{T}$ **is maximal.**

- Example: $\boldsymbol{v}$ = ATGCCAT, $\boldsymbol{w}$ = TCGGGCTATC.
  - Before we found that TGAT is a subsequence.
  - The *longest* subsequence is **TGCAT**.
  - But…how do we *find* the LCS of two sequences?

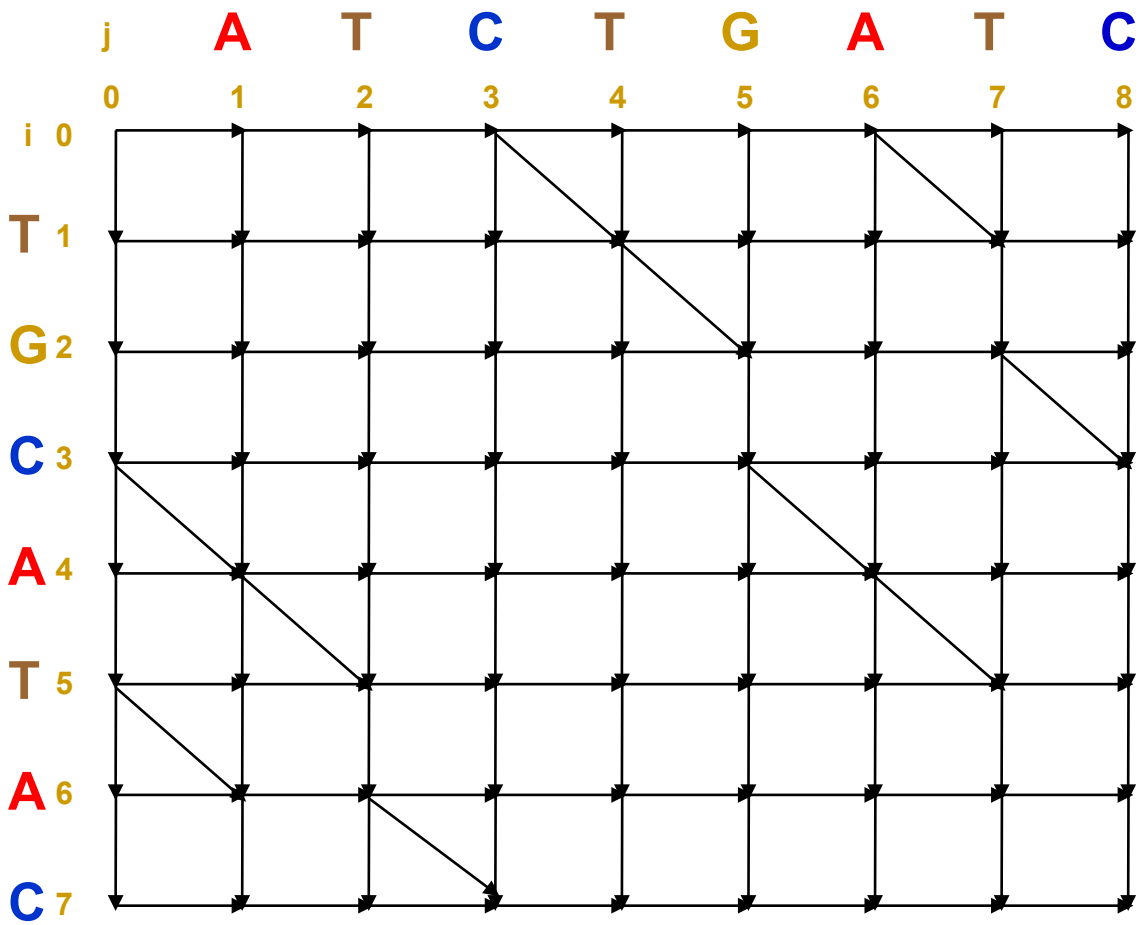
Edit Graph for LCS Problem

- Assign one sequence to the rows, and one to the columns.



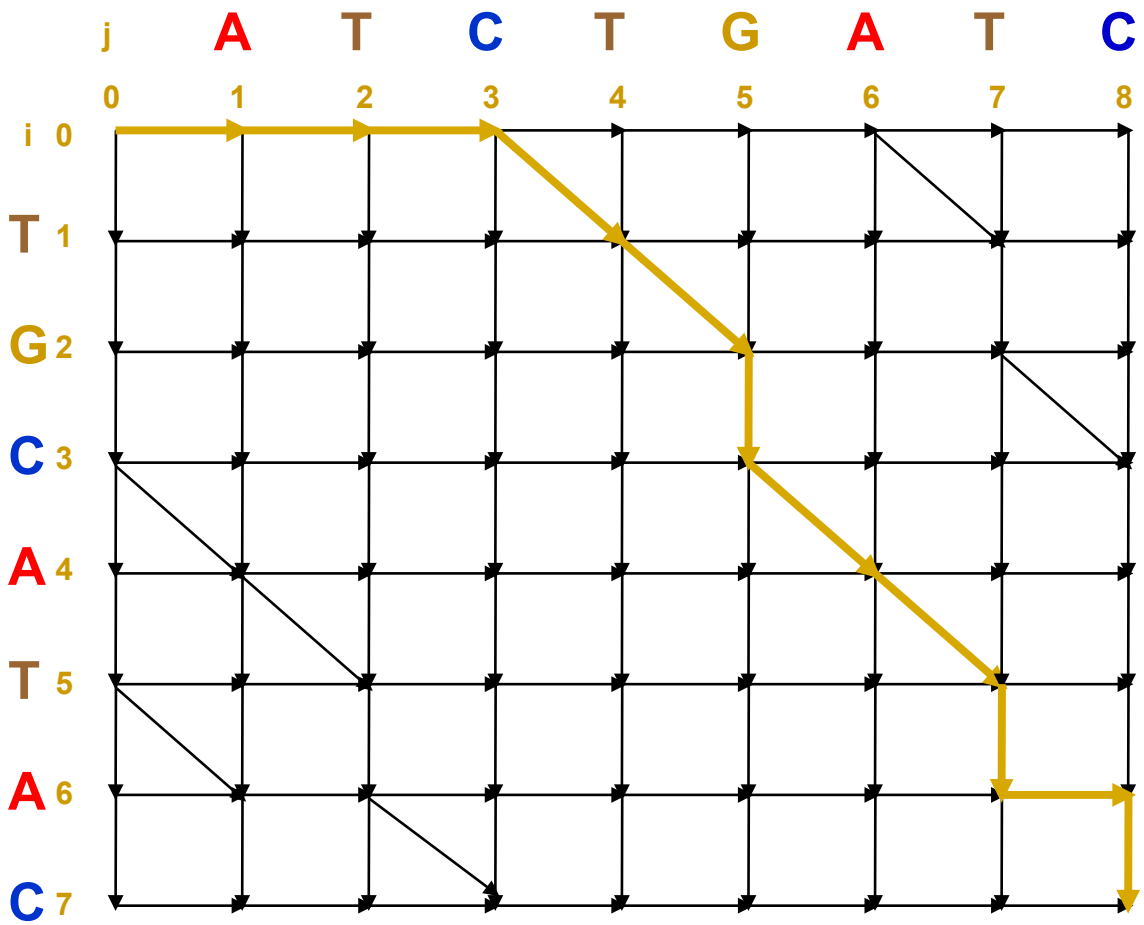
Edit Graph for LCS Problem

- Assign one sequence to the rows, and one to the columns.
- Every diagonal edge represents a match of elements.



Edit Graph for LCS Problem

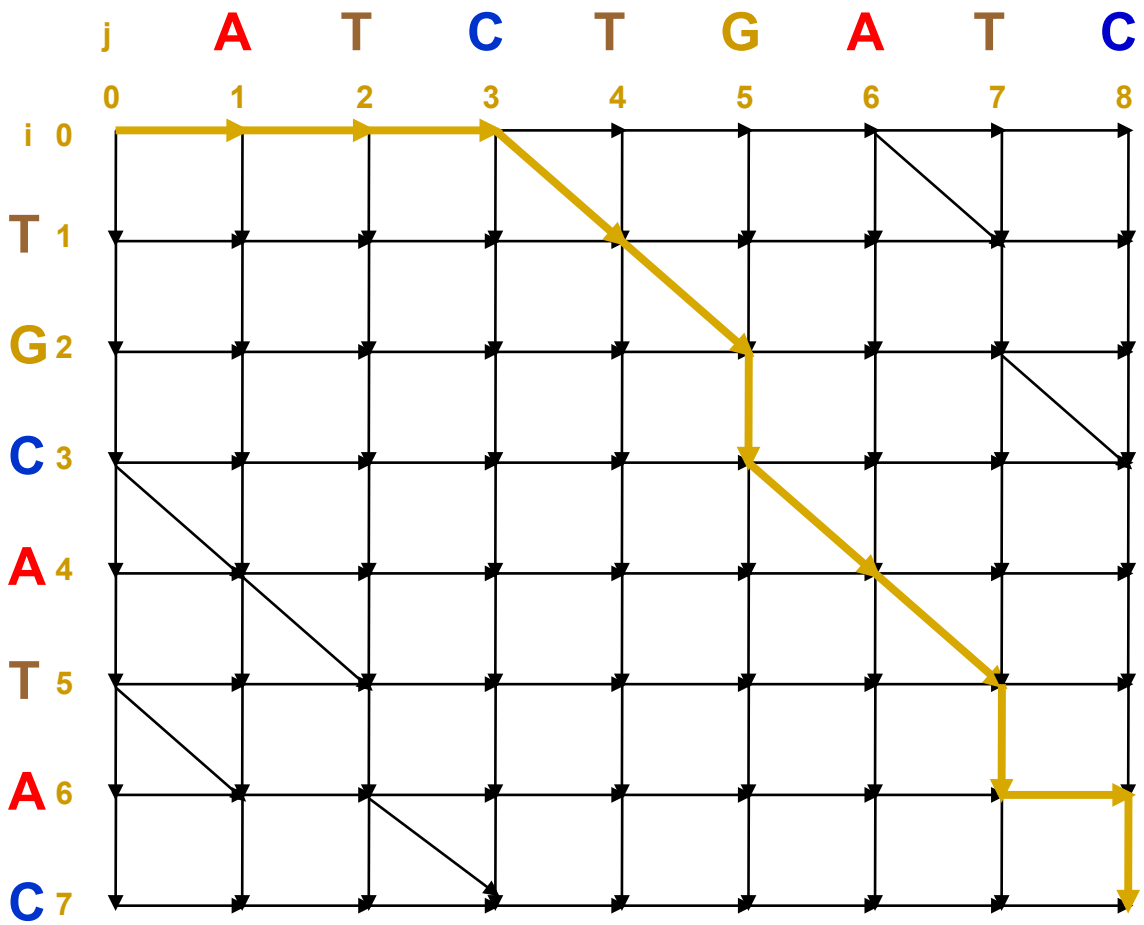
- Assign one sequence to the rows, and one to the columns.
- Every diagonal edge represents a match of elements.
- Therefore, in a path from source to sink, the diagonal edges represent a common subsequence.



Common Subsequence: TGAT

Edit Graph for LCS Problem

- **LCS Problem:**
  Find a path with the maximum number of diagonal edges.



Common Subsequence: TGAT

Computing the LCS: Dynamic Programming

- Let $\boldsymbol{v}_i$ = prefix of $\boldsymbol{v}$ of length i: $v_1 \ldots v_i$
- and $\boldsymbol{w}_j$ = prefix of $\boldsymbol{w}$ of length j: $w_1 \ldots w_j$

- The length of LCS($\boldsymbol{v}_i$,$\boldsymbol{w}_j$) is computed by:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1 \quad if \ v_i = w_j \end{cases}$$

Section 6: Edit Distance

Hamming Distance

- The **Hamming Distance** $d_H(\boldsymbol{v}, \boldsymbol{w})$ between two DNA sequences $\boldsymbol{v}$ and $\boldsymbol{w}$ of the same length is equal to the number of places in which the two sequences differ.

- Example: Given as follows, $d_H(\boldsymbol{v}, \boldsymbol{w}) = 8$:

v: ATATATAT
w: TATATATA

- However, note that these sequences are still very similar.
  - Hamming Distance is therefore not an ideal similarity score, because it ignores insertions and deletions.

Edit Distance

- Levenshtein (1966) introduced the **edit distance** between two strings as the minimum number of elementary operations (insertions, deletions, and substitutions) needed to transform one string into the other

$d(\mathbf{v},\mathbf{w})$ = MIN number of elementary operations to transform $\mathbf{v} \rightarrow \mathbf{w}$

Edit Distance: Example 1

- So in our previous example, we can shift *w* one nucleotide to the right, and see that *w* is obtained from *v* by one insertion and one deletion:

```
v:  ATATATAT-
w:  -TATATATA
```

- Hence the edit distance, $d(v, w) = 2$.

- **Note**: In order to provide this distance, we had to "fiddle" with the sequences. Hamming distance was easier to find.

Edit Distance: Example 2

- We can transform TGCATAT → ATCCGAT in 5 steps:

TGCATAT

Edit Distance: Example 2

- We can transform TGCATAT → ATCCGAT in 5 steps:

TGCATAT (delete last T)

Edit Distance: Example 2

- We can transform TGCATAT → ATCCGAT in 5 steps:

TGCATAT (delete last T)
TGCATA (delete last A)

Edit Distance: Example 2

- We can transform TGCATAT → ATCCGAT in 5 steps:

TGCATAT	(delete last T)
TGCATA	(delete last A)
ATGCAT	(insert A at front)

Edit Distance: Example 2

- We can transform TGCATAT → ATCCGAT in 5 steps:

TGCATAT	(delete last T)
TGCATA	(delete last A)
ATGCAT	(insert A at front)
ATCCAT	(substitute C for G)

Edit Distance: Example 2

- We can transform TGCATAT → ATCCGAT in 5 steps:

TGCATA**T**	(delete last T)
TGCAT**A**	(delete last A)
ATGCAT	(insert A at front)
ATC**C**AT	(substitute C for G)
ATCC**G**AT	(insert G before last A)

Edit Distance: Example 2

- We can transform TGCATAT → ATCCGAT in 5 steps:

TGCATAT	(delete last T)
TGCATA	(delete last A)
ATGCAT	(insert A at front)
ATCCAT	(substitute C for G)
ATCCGAT	(insert G before last A)

- **Note**: This *only* allows us to conclude that the edit distance is *at most* 5.

Edit Distance: Example 2

- Now we transform TGCATAT → ATCCGAT in 4 steps:

ATGCATAT

Edit Distance: Example 2

- Now we transform TGCATAT → ATCCGAT in 4 steps:

ATGCATAT (insert A at front)

Edit Distance: Example 2

- Now we transform TGCATAT → ATCCGAT in 4 steps:

ATGCATAT (insert A at front)
ATGCATAT (delete second T)

Edit Distance: Example 2

- Now we transform TGCATAT → ATCCGAT in 4 steps:

ATGCATAT	(insert A at front)
ATGCATAT	(delete second T)
ATGCGAT	(substitute G for A)

Edit Distance: Example 2

- Now we transform TGCATAT → ATCCGAT in 4 steps:

ATGCATAT	(insert A at front)
ATGCATAT	(delete second T)
ATGCGAT	(substitute G for A)
ATCCGAT	(substitute C for G)

Edit Distance: Example 2

- Now we transform TGCATAT → ATCCGAT in 4 steps:

ATGCATAT	(insert A at front)
ATGCATAT	(delete second T)
ATGCGAT	(substitute G for A)
ATCCGAT	(substitute C for G)

- Can we do even better? 3 steps? 2 steps? How can we know?

Key Result

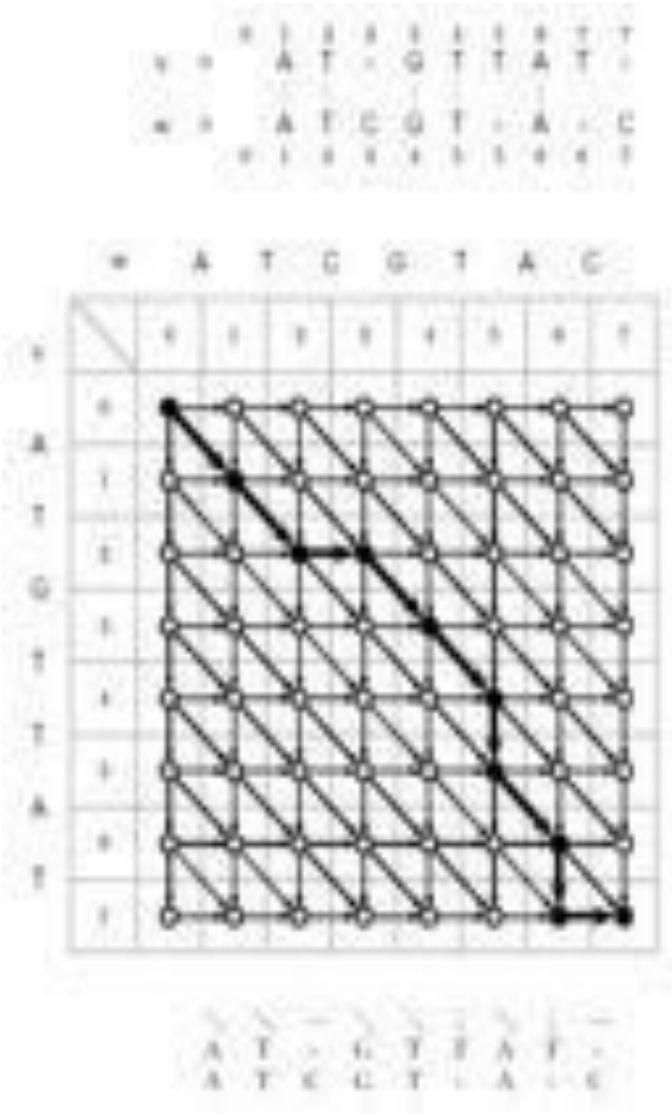
- **Theorem**: Given two sequences $\boldsymbol{v}$ and $\boldsymbol{w}$ of length m and n, the edit distance $d(\boldsymbol{v},\boldsymbol{w})$ is given by $d(\boldsymbol{v},\boldsymbol{w}) = m + n - s(\boldsymbol{v},\boldsymbol{w})$, where $s(\boldsymbol{v},\boldsymbol{w})$ is the length of the longest common subsequence of $\boldsymbol{v}$ and $\boldsymbol{w}$.

- This is great news, because it means that if solving the LCS problem for $\boldsymbol{v}$ and $\boldsymbol{w}$ is equivalent to finding the edit distance between them.

- Therefore, we will solve the LCS problem instead in the following slides.

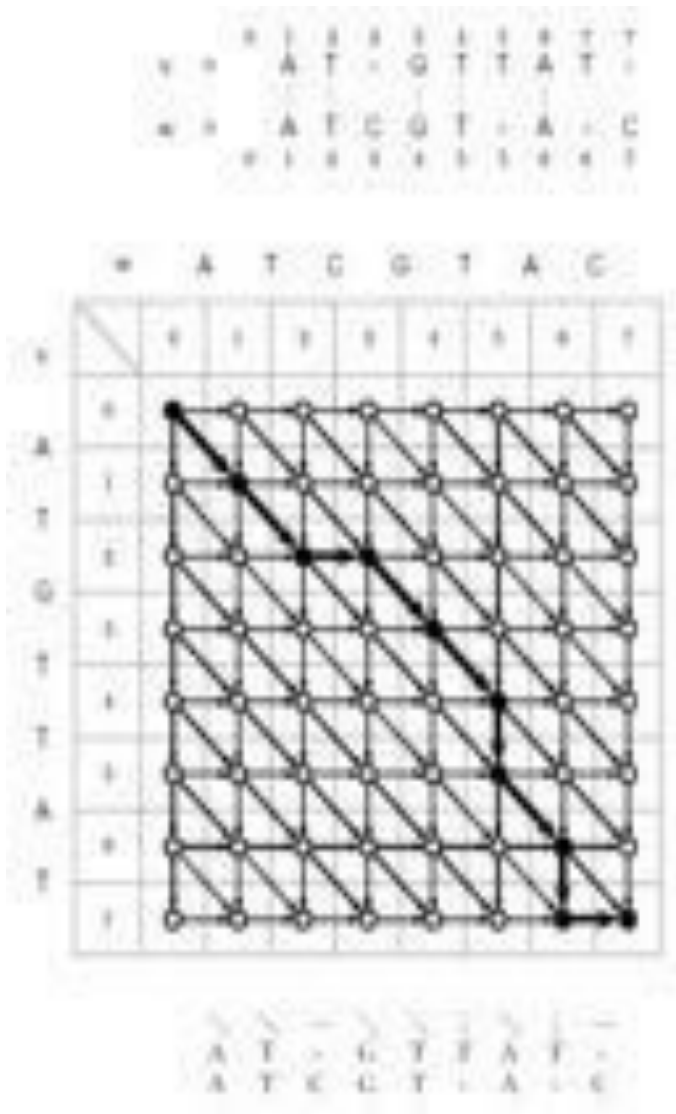
Return to the Edit Graph

- Every alignment corresponds to a path from source to sink.



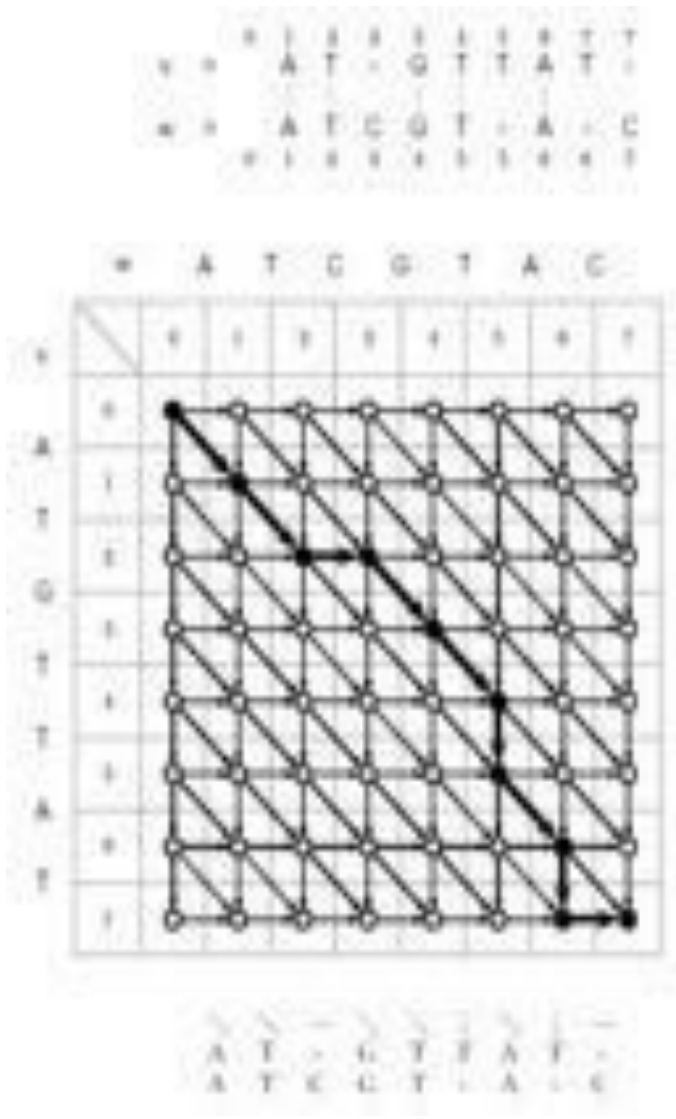
Return to the Edit Graph

- Every alignment corresponds to a path from source to sink.
- Horizontal and vertical edges correspond to indels (deletions and insertions).



Return to the Edit Graph

- Every alignment corresponds to a path from source to sink.
- Horizontal and vertical edges correspond to indels (deletions and insertions).
- Diagonal edges correspond to matches and mismatches.



Alignment as a Path in the Edit Graph: Example

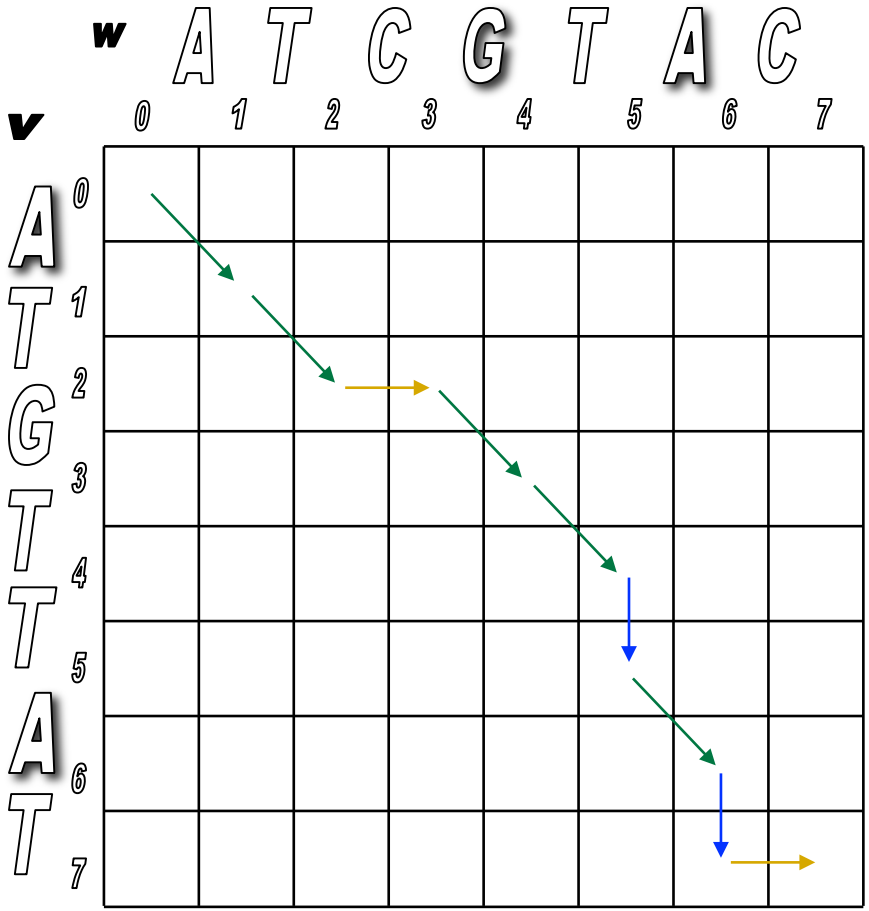
- Suppose that our sequences are ATCGTAC, ATGTTAT.

- One possible alignment is:

```
0 1 2 2 3 4 5 6 7 7
  A T _ G T T A T _
  A T C G T _ A _ C
0 1 2 3 4 5 5 6 6 7
```

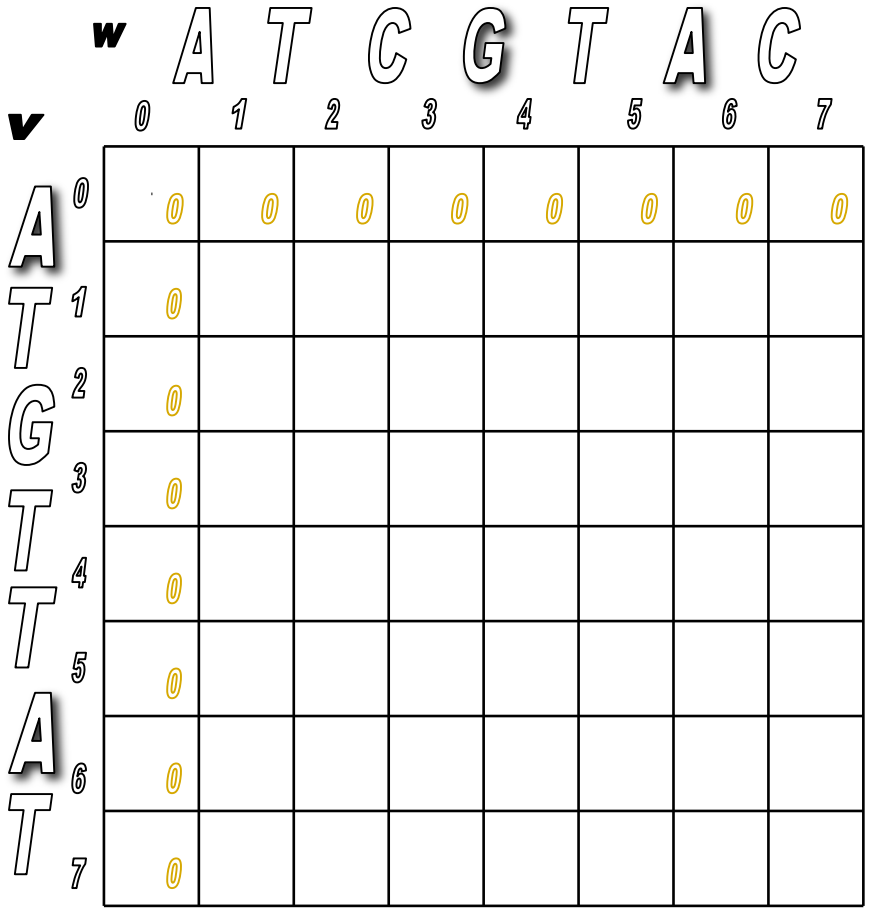
- Edit graph path:

(0,0)→(1,1)→(2,2)→(2,3)→(3,4)→etc.



Alignment with Dynamic Programming

- Initialize 0^{th} row and 0^{th} *column to be all zeroes.*

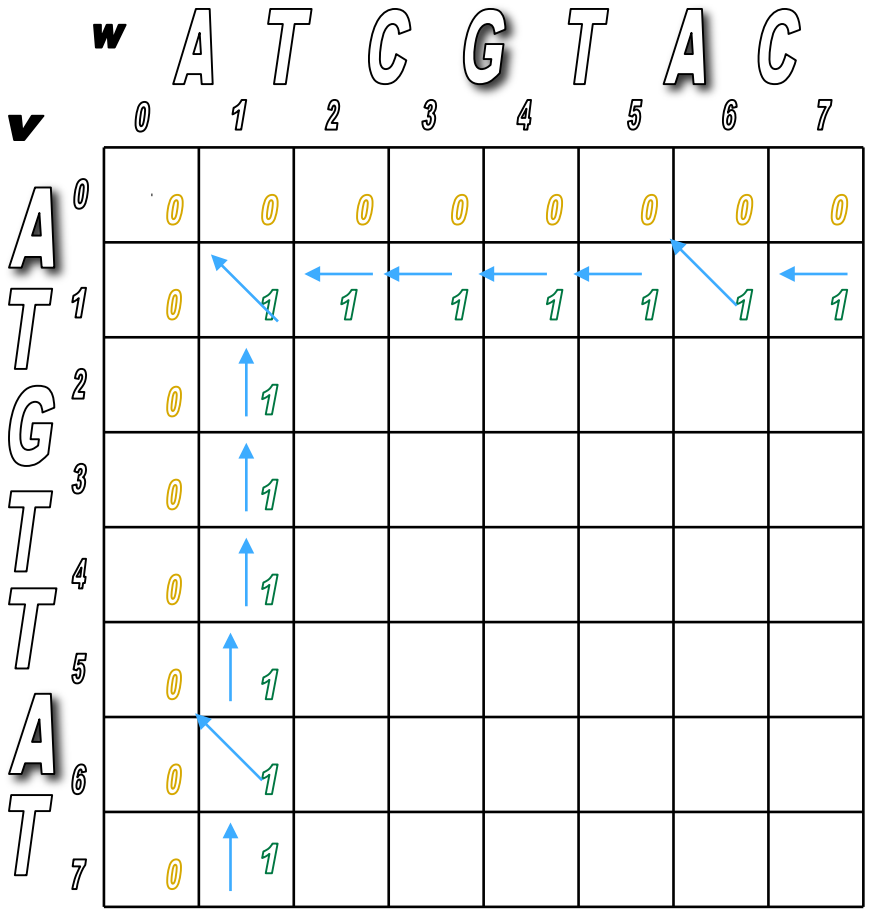


Alignment with Dynamic Programming

- Initialize 0^{th} row and 0^{th} *column to be all zeroes.*

- Use the following recursive formula to calculate $s_{i,j}$ for each i, j:

$$s_{i,j} = max \begin{cases} s_{i-1,j-1} + 1 \\ s_{i-1,j} \\ s_{i,j-1} \end{cases}$$

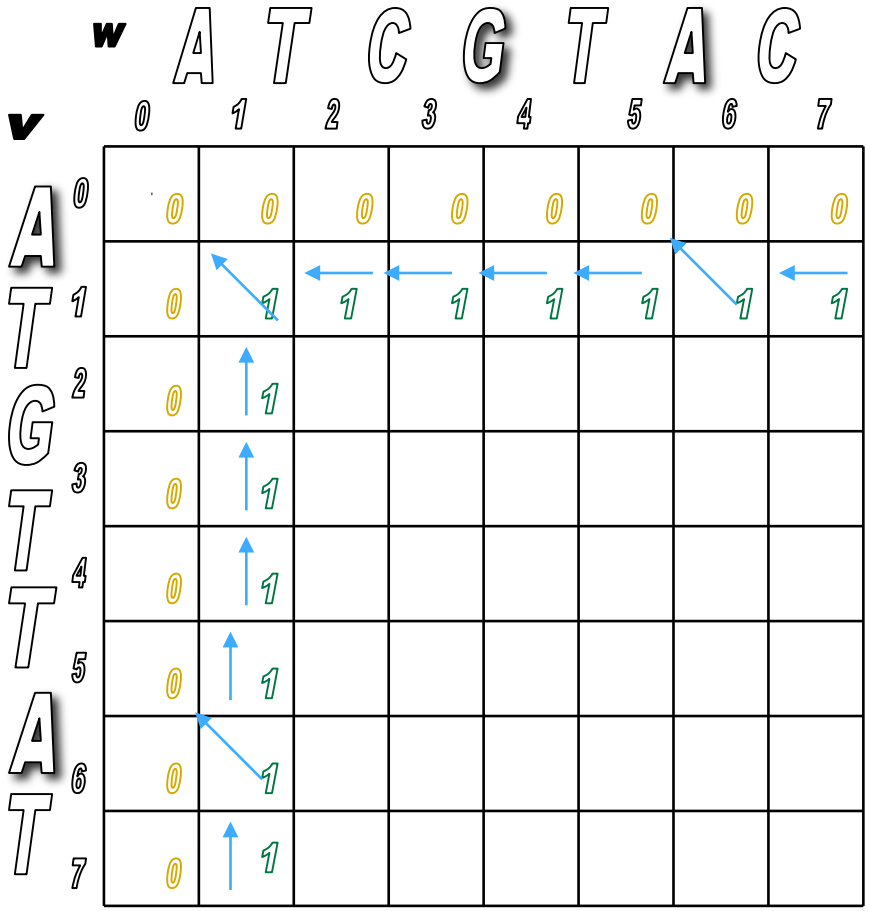


Dynamic Programming Example

- Note that the placement of the arrows shows where a given score originated from:
  - if from the top
  - if from the left
  -  if $v_i = w_j$

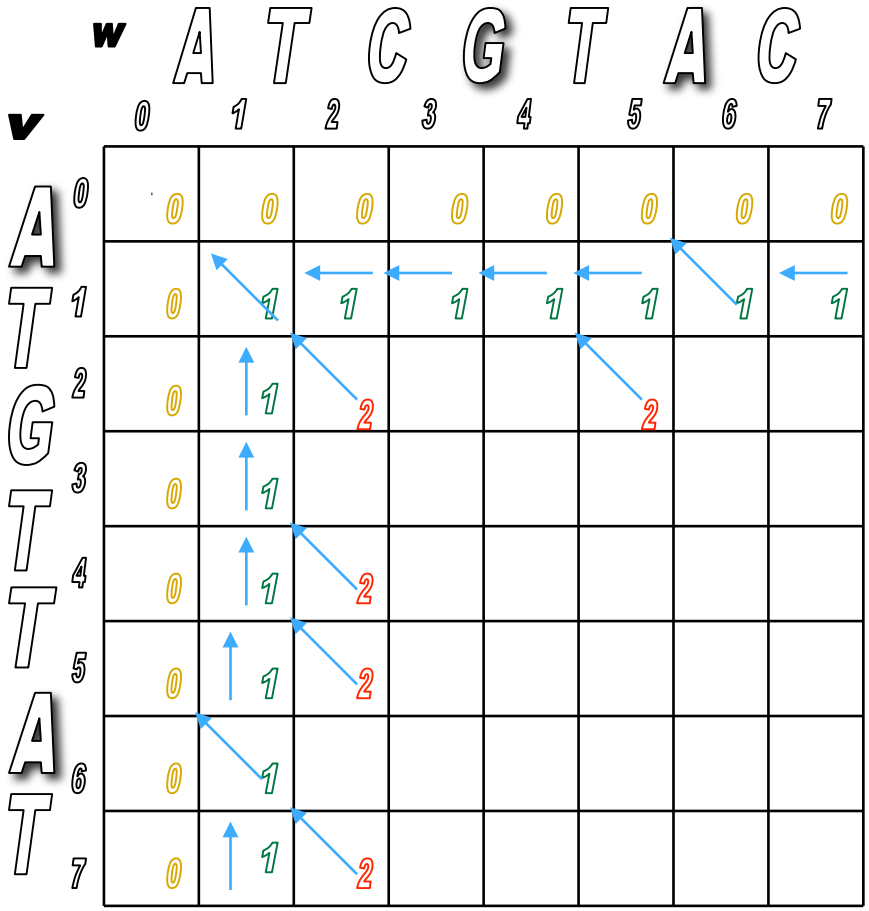
Dynamic Programming Example

- Continuing with the dynamic programming algorithm fills the table.



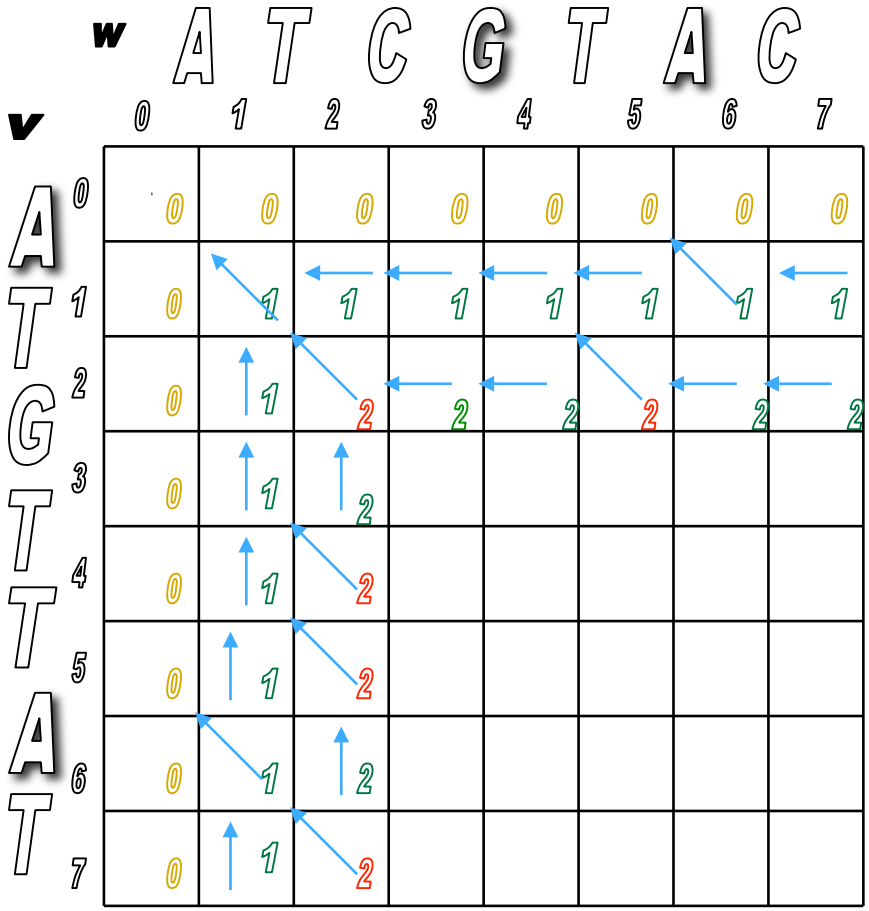
Dynamic Programming Example

- Continuing with the dynamic programming algorithm fills the table.
- We first look for matches, highlighted in red, and then we fill in the rest of that row and column.



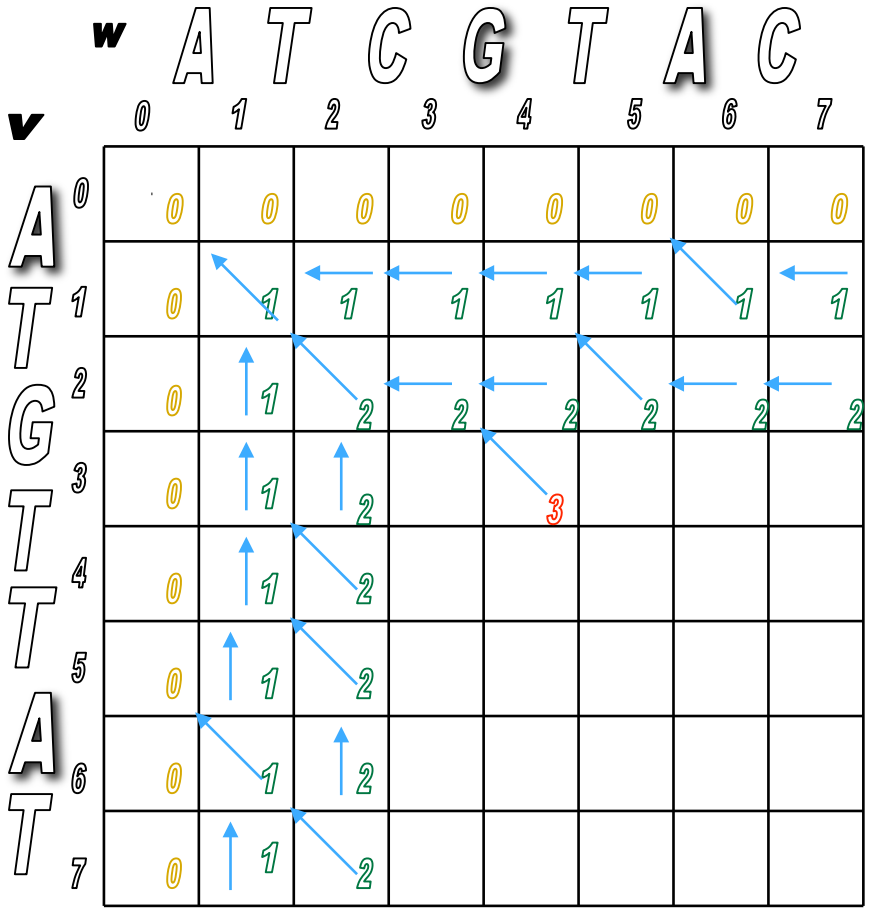
Dynamic Programming Example

- Continuing with the dynamic programming algorithm fills the table.
- We first look for matches, highlighted in red, and then we fill in the rest of that row and column.



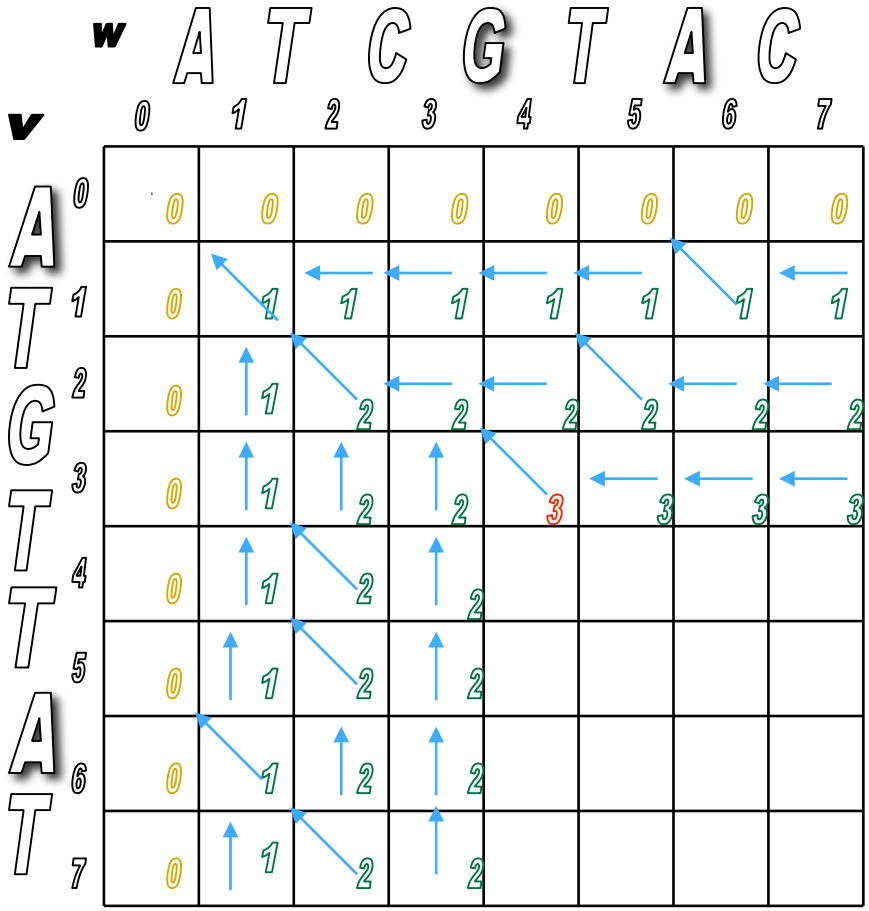
Dynamic Programming Example

- Continuing with the dynamic programming algorithm fills the table.
- We first look for matches, highlighted in red, and then we fill in the rest of that row and column.
- As we can see, we do not simply add 1 each time.



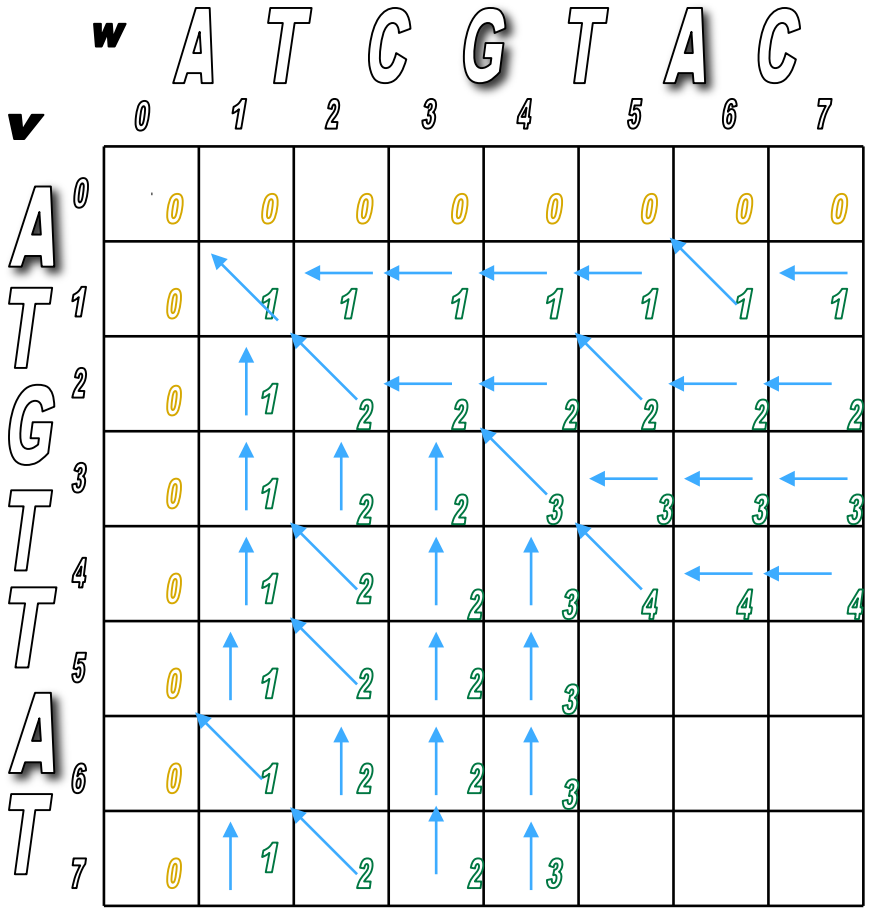
Dynamic Programming Example

- Continuing with the dynamic programming algorithm fills the table.
- We first look for matches, highlighted in red, and then we fill in the rest of that row and column.
- As we can see, we do not simply add 1 each time.



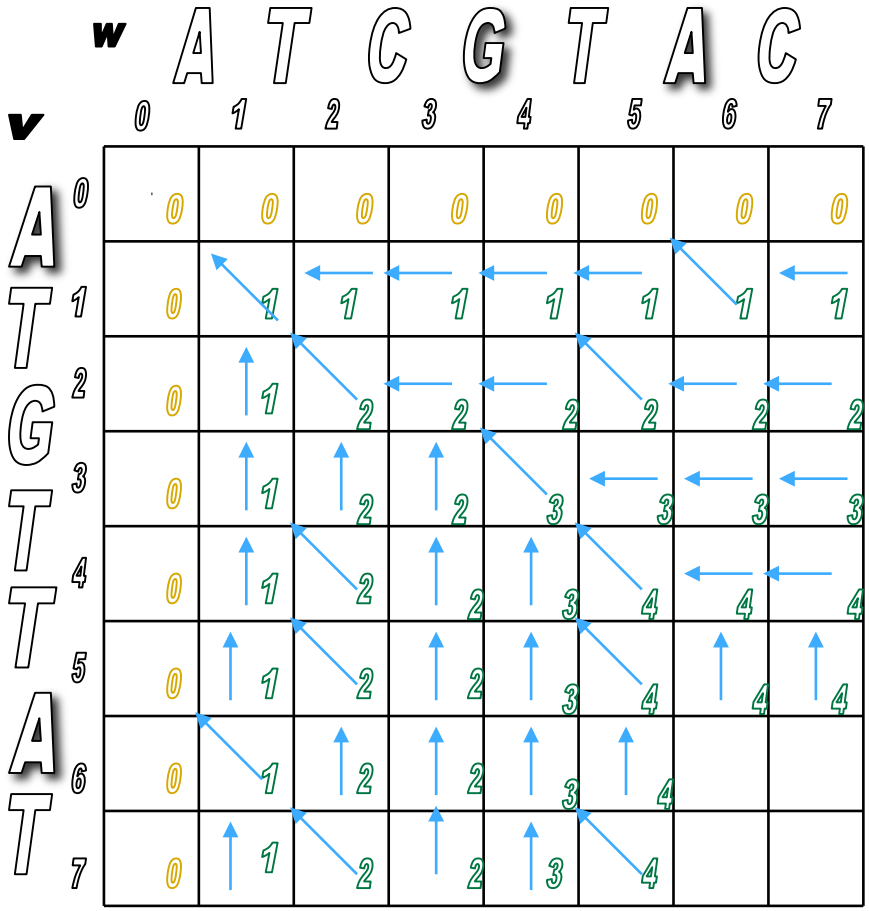
Dynamic Programming Example

- Continuing with the dynamic programming algorithm fills the table.
- We first look for matches, highlighted in red, and then we fill in the rest of that row and column.
- As we can see, we do not simply add 1 each time.



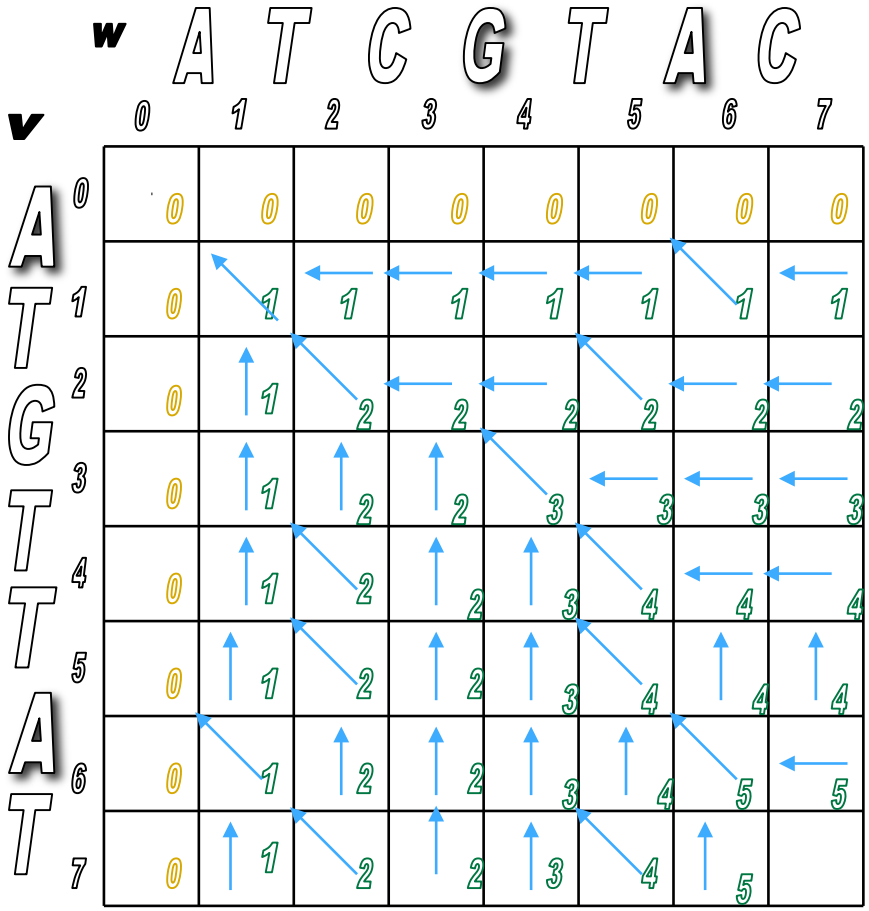
Dynamic Programming Example

- Continuing with the dynamic programming algorithm fills the table.
- We first look for matches, highlighted in red, and then we fill in the rest of that row and column.
- As we can see, we do not simply add 1 each time.



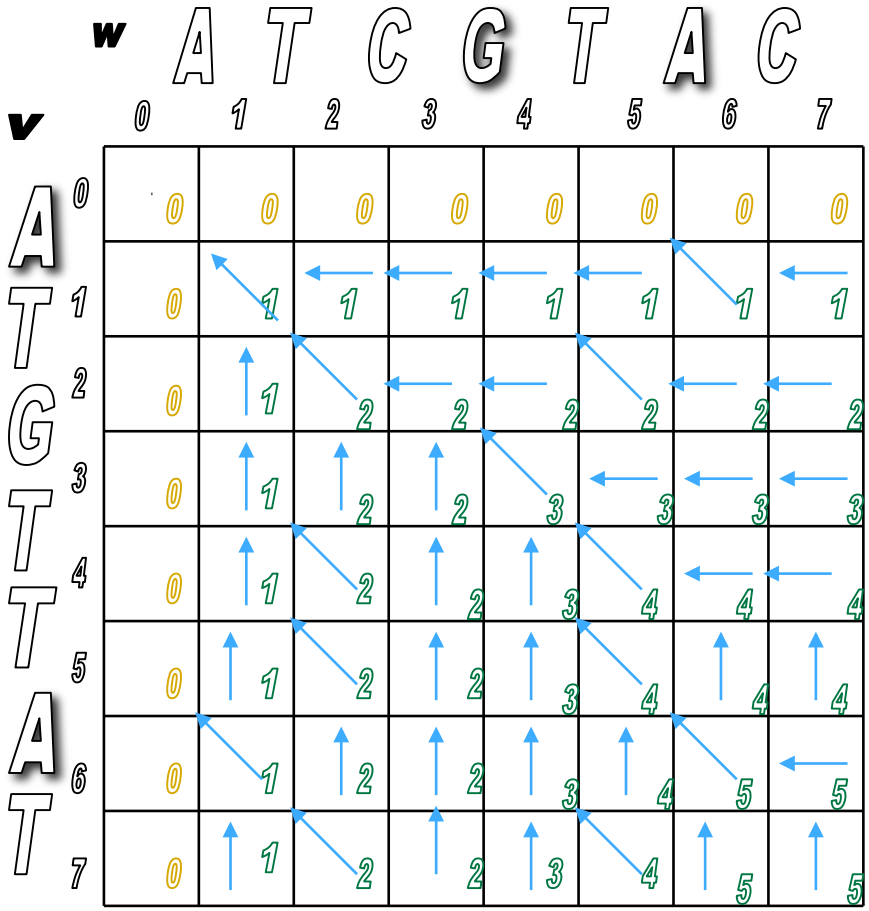
Dynamic Programming Example

- Continuing with the dynamic programming algorithm fills the table.
- We first look for matches, highlighted in red, and then we fill in the rest of that row and column.
- As we can see, we do not simply add 1 each time.



Dynamic Programming Example

- Continuing with the dynamic programming algorithm fills the table.
- We first look for matches, highlighted in red, and then we fill in the rest of that row and column.
- As we can see, we do not simply add 1 each time.

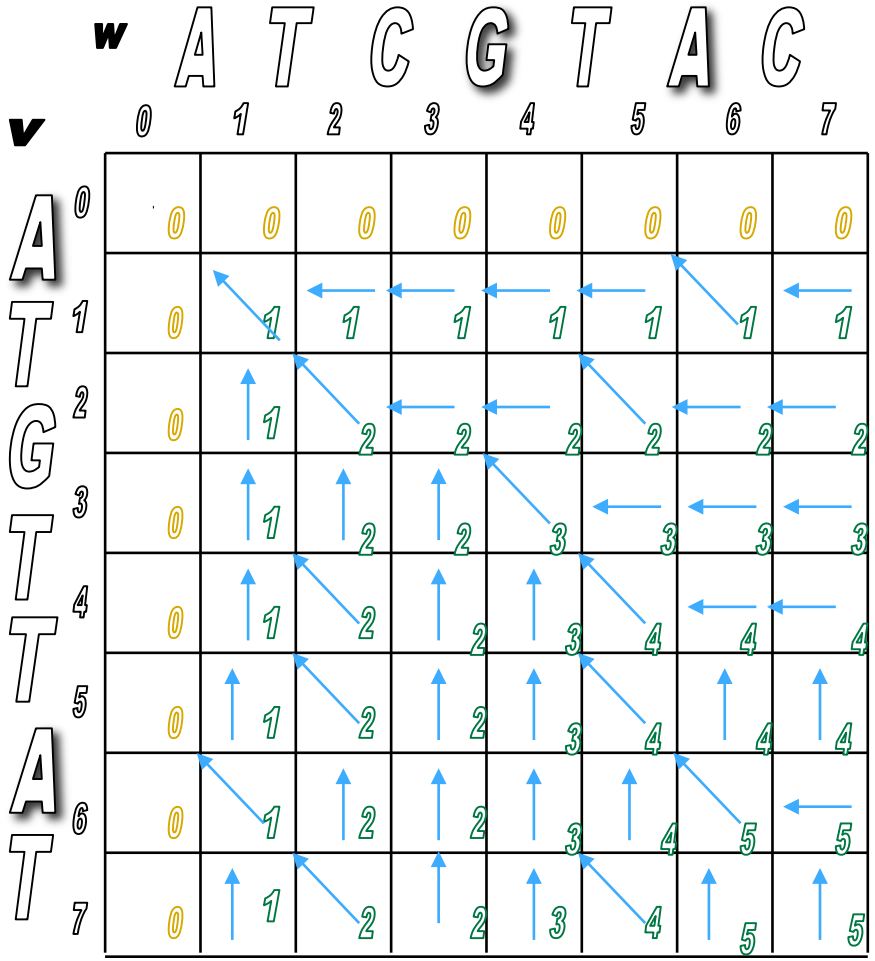


Dynamic Alignment: Pseudocode

```
1. LCS(v,w)
2.   for i ← 1 to n
3.     s_{i,0} ← 0
4.   for j ← 1 to m
5.     s_{0,j} ← 0
6.   for i ← 1 to n
7.     for j ← 1 to m
8.
9.       s_{i,j} ← max { s_{i-1,j}
10.                    { s_{i,j-1}
11.                    { s_{i-1,j-1} + 1, if v_i = w_j
•        b_{i,j} ← { "↑" if s_{i,j} = s_{i-1,j}
•                  { "←" if s_{i,j} = s_{i,j-1}
•                  { "↖" if s_{i,j} = s_{i-1,j-1} + 1
•    return (s_{n,m}, b)
```

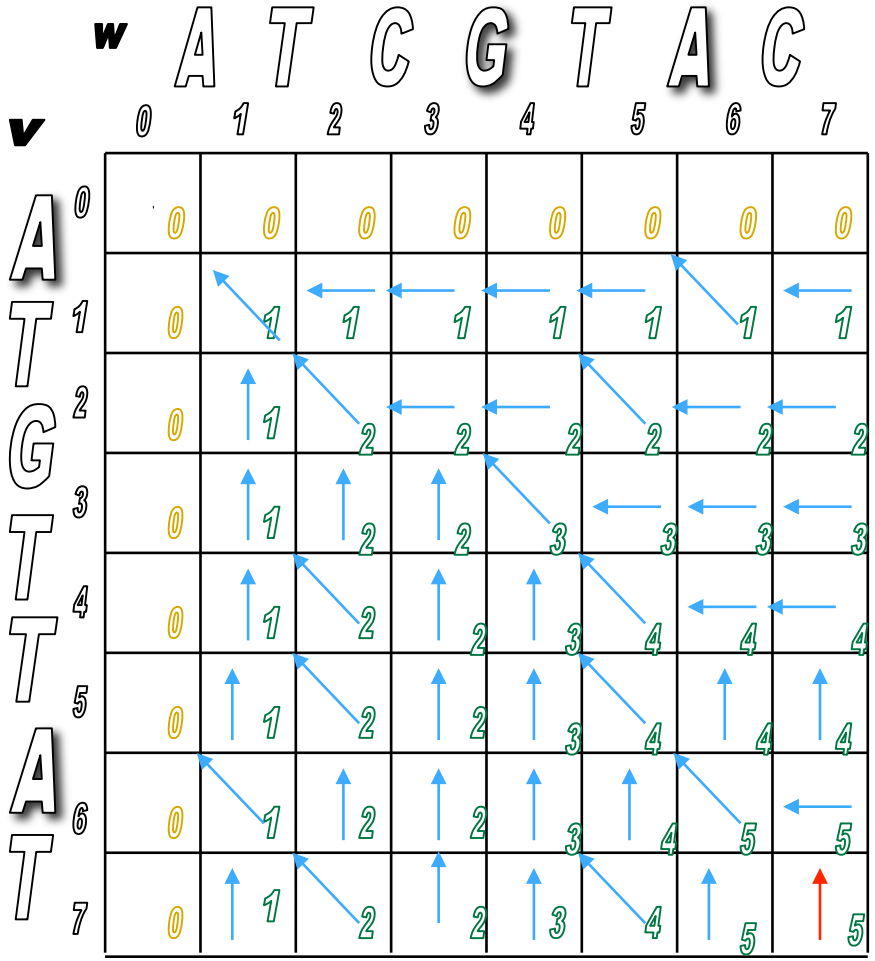
Finding an Optimal Alignment

- LCS(v,w) created the alignment grid.
- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:



Finding an Optimal Alignment

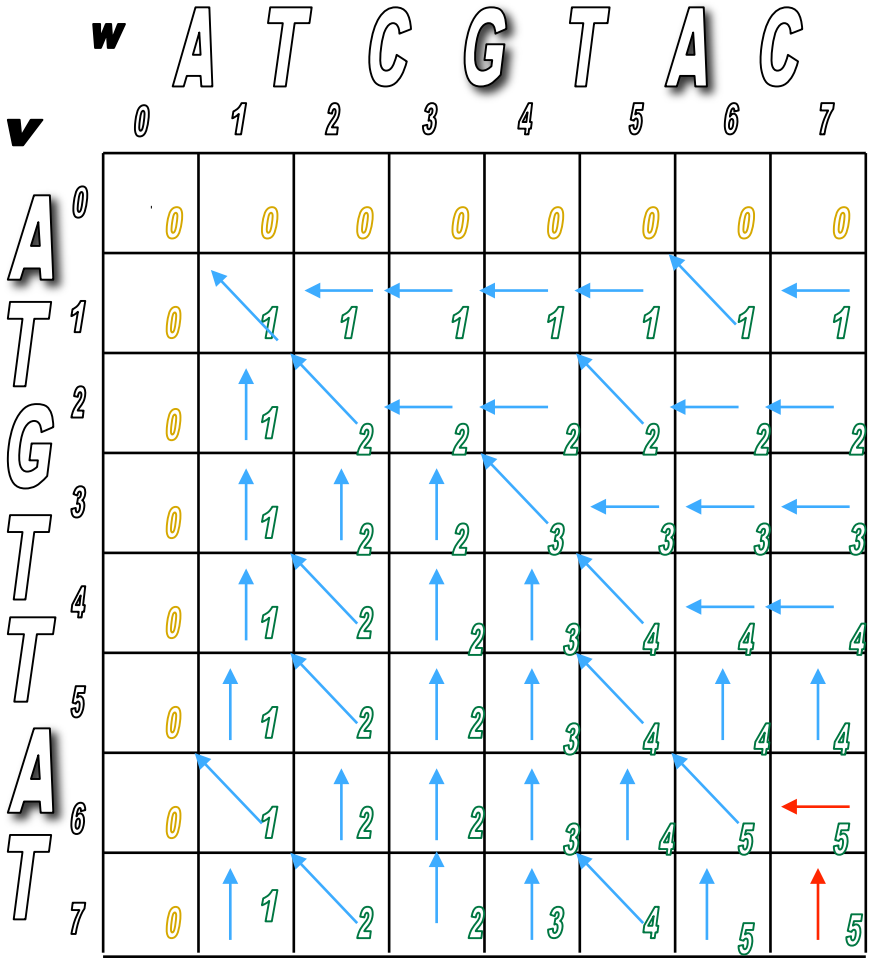
- LCS(v,w) created the alignment grid.
- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:



7
T
–
7

Finding an Optimal Alignment

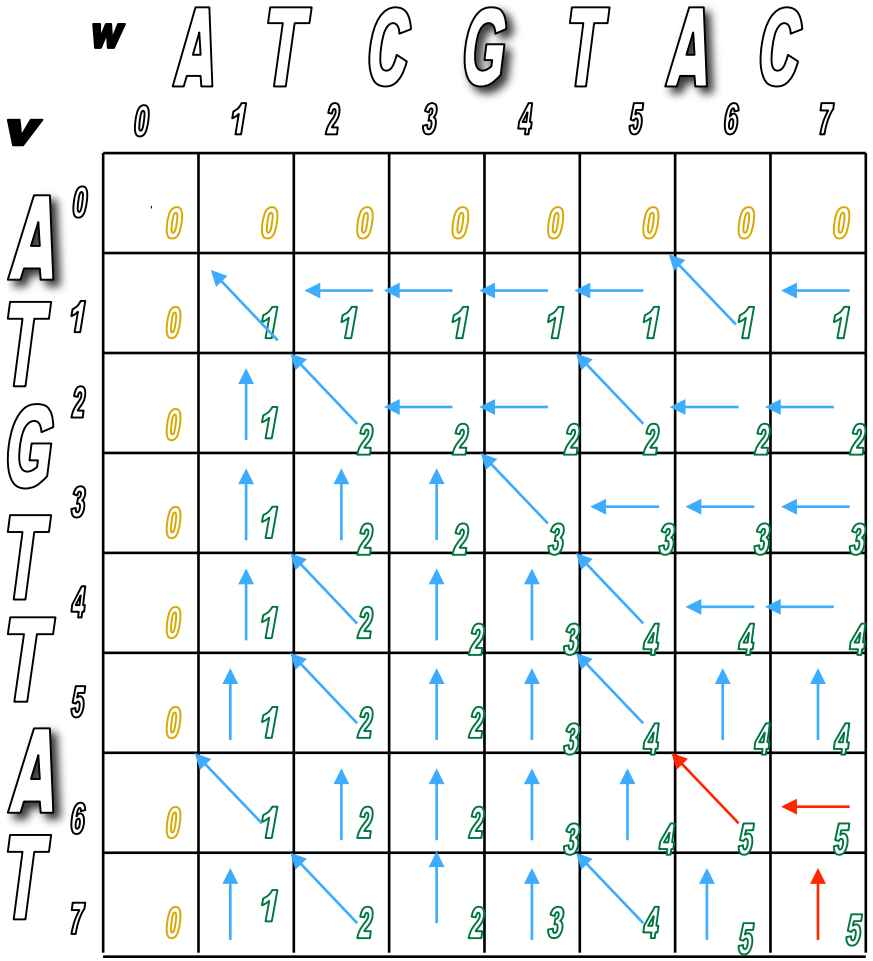
- LCS(v,w) created the alignment grid.
- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:



7 7
– T
C –
6 7

Finding an Optimal Alignment

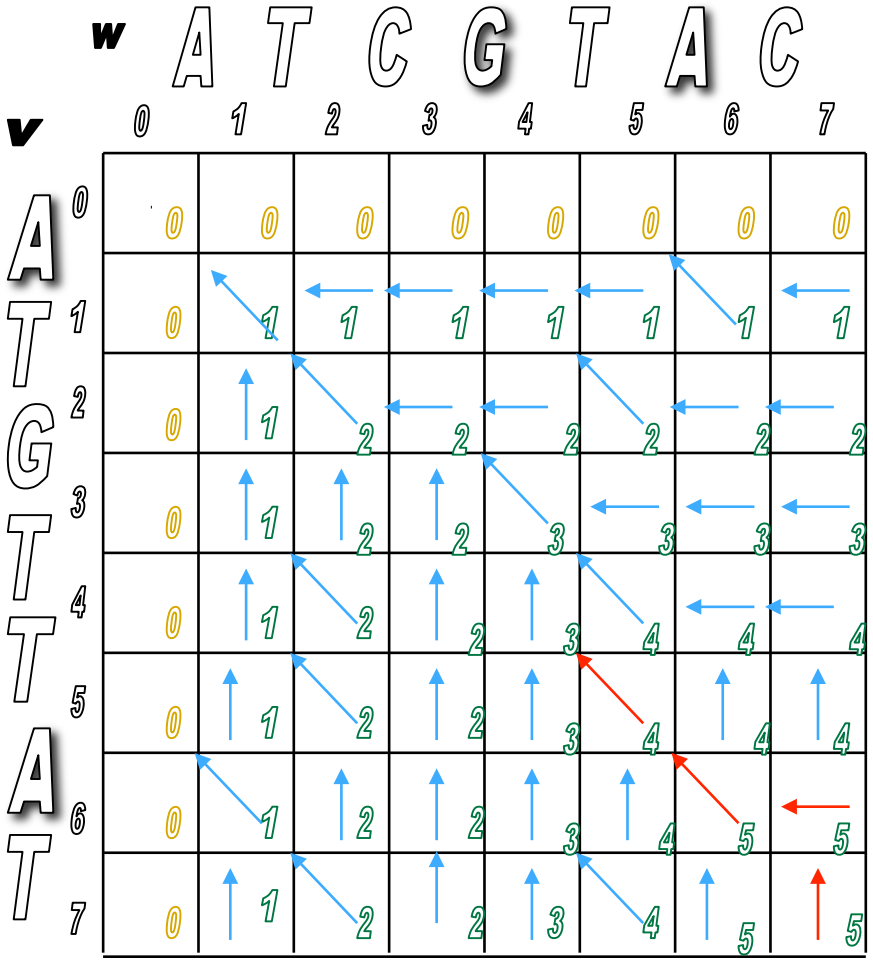
- LCS(v,w) created the alignment grid.
- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:



6	7	7
A	_	T
A	C	_
6	6	7

Finding an Optimal Alignment

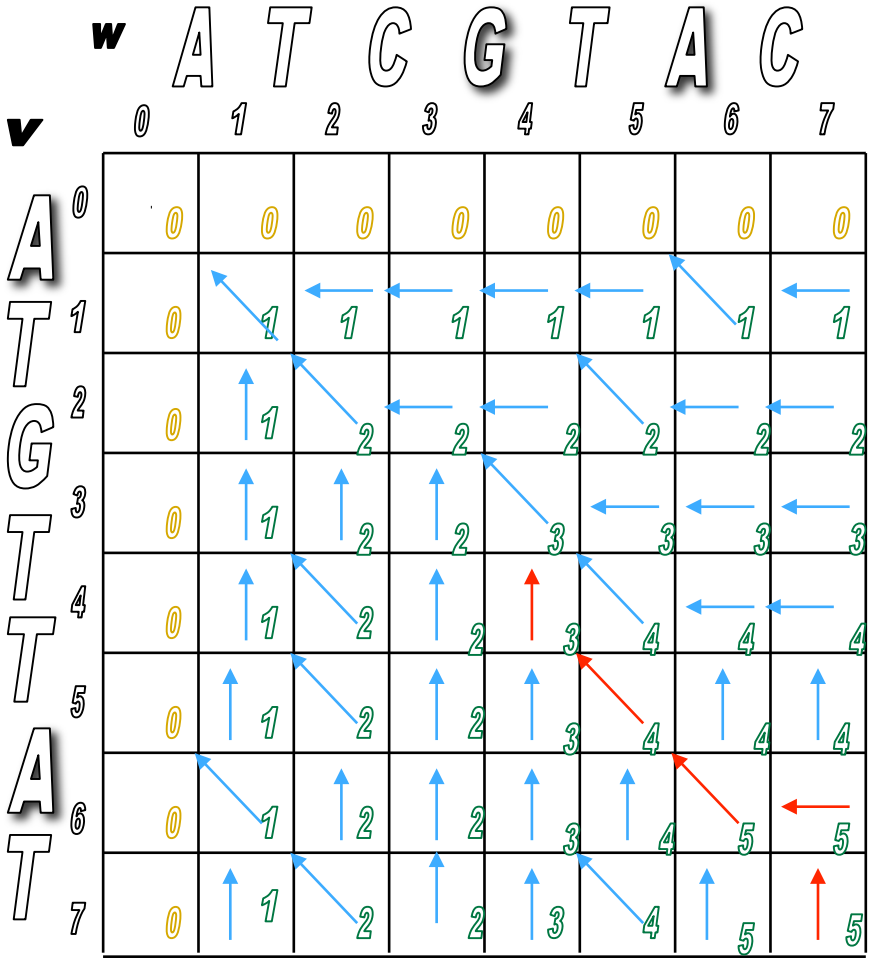
- LCS(v,w) created the alignment grid.
- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:



5	6	7	7
T	A	_	T
_	A	C	_
5	6	6	7

Finding an Optimal Alignment

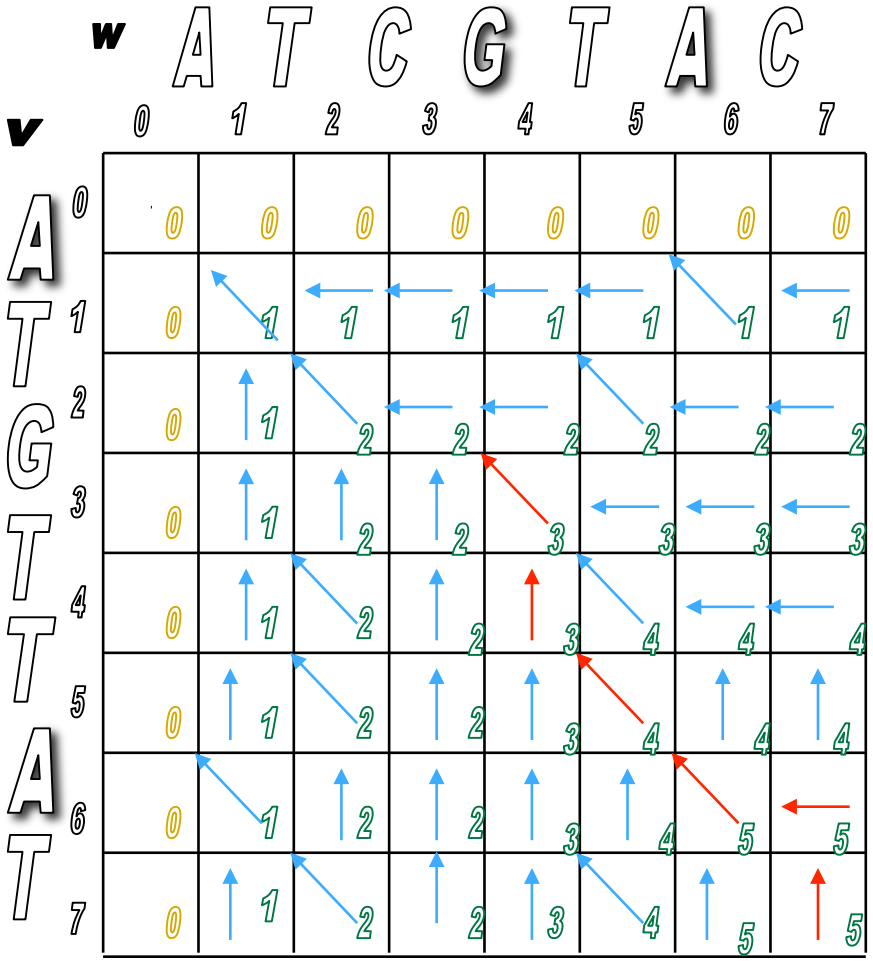
- LCS(v,w) created the alignment grid.
- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:



4	5	6	7	7
T	T	A	_	T
T	_	A	C	_
5	5	6	6	7

Finding an Optimal Alignment

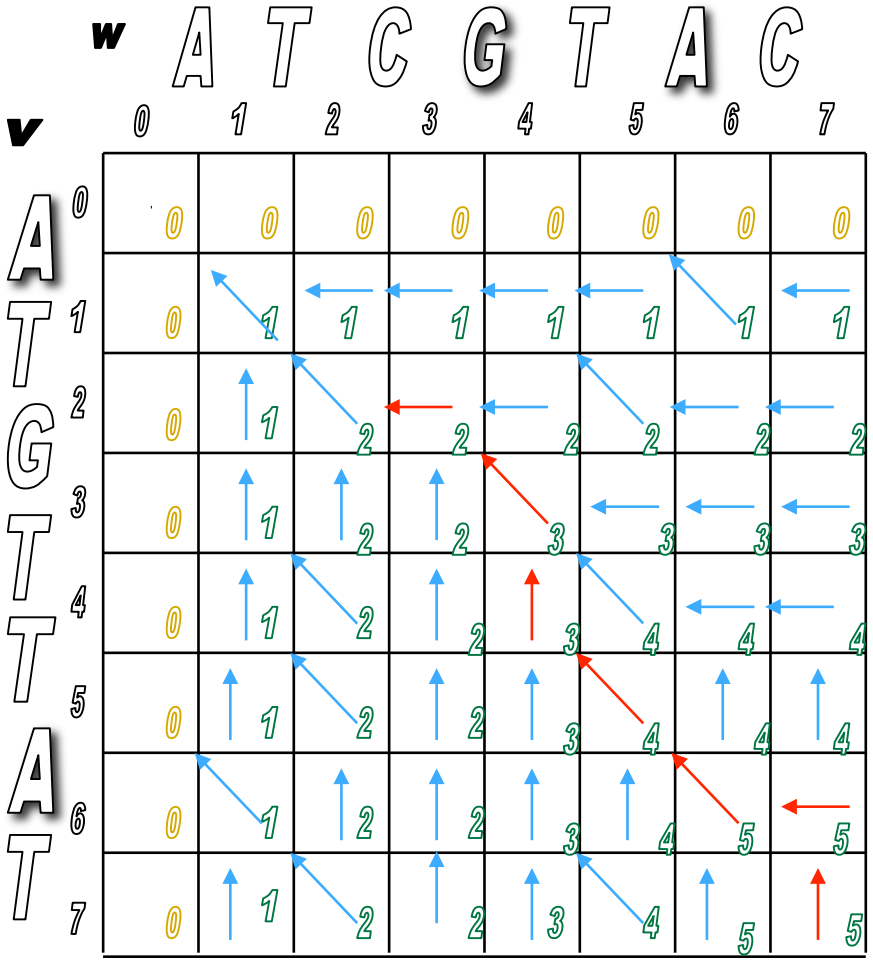
- LCS(v,w) created the alignment grid.
- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:



3	4	5	6	7	7
G	T	T	A	_	T
G	T	_	A	C	_
4	5	5	6	6	7

Finding an Optimal Alignment

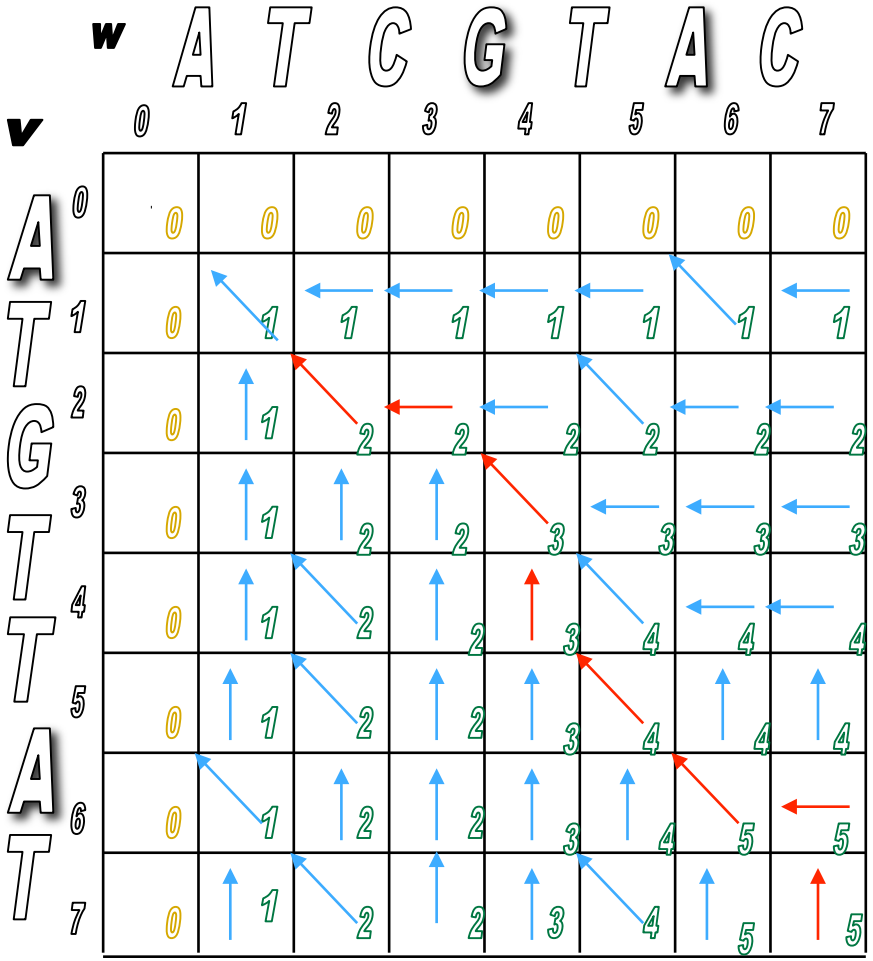
- LCS(v,w) created the alignment grid.
- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:



2	3	4	5	6	7	7
_	G	T	T	A	_	T
C	G	T	_	A	C	_
3	4	5	5	6	6	7

Finding an Optimal Alignment

- LCS(v,w) created the alignment grid.
- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:

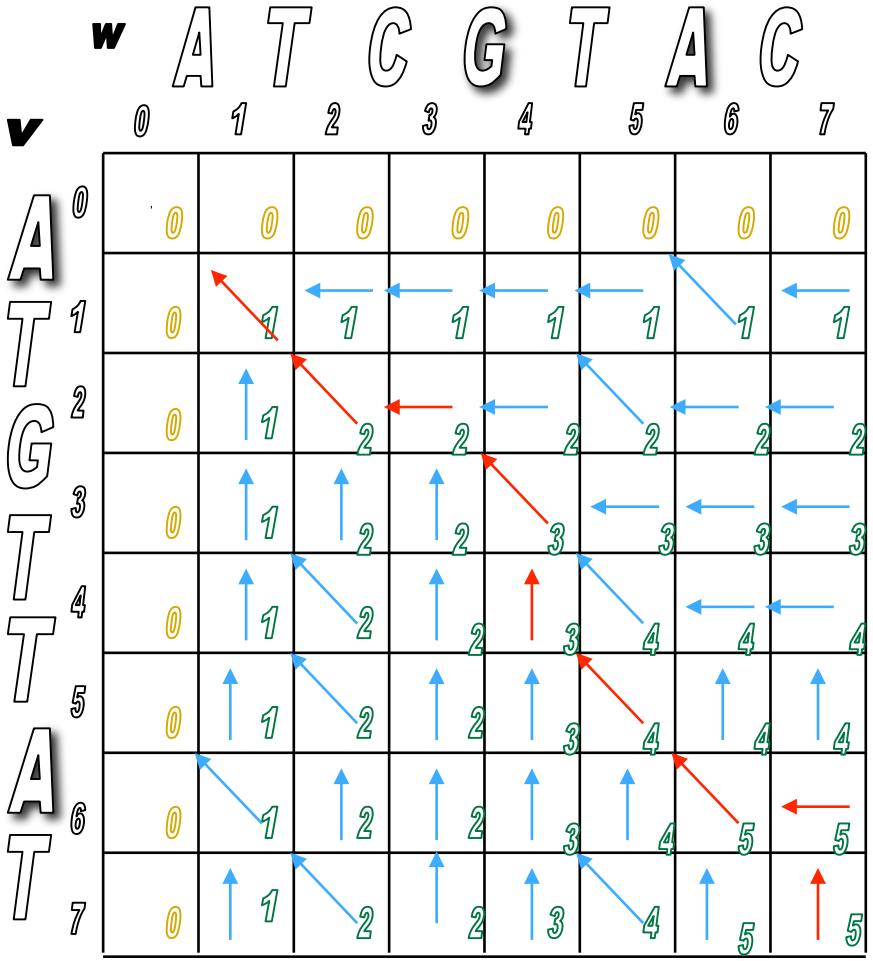


2	2	3	4	5	6	7	7
T	_	G	T	T	A	_	T
T	C	G	T	_	A	C	_
2	3	4	5	5	6	6	7

Finding an Optimal Alignment

- LCS(v,w) created the alignment grid.

- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:

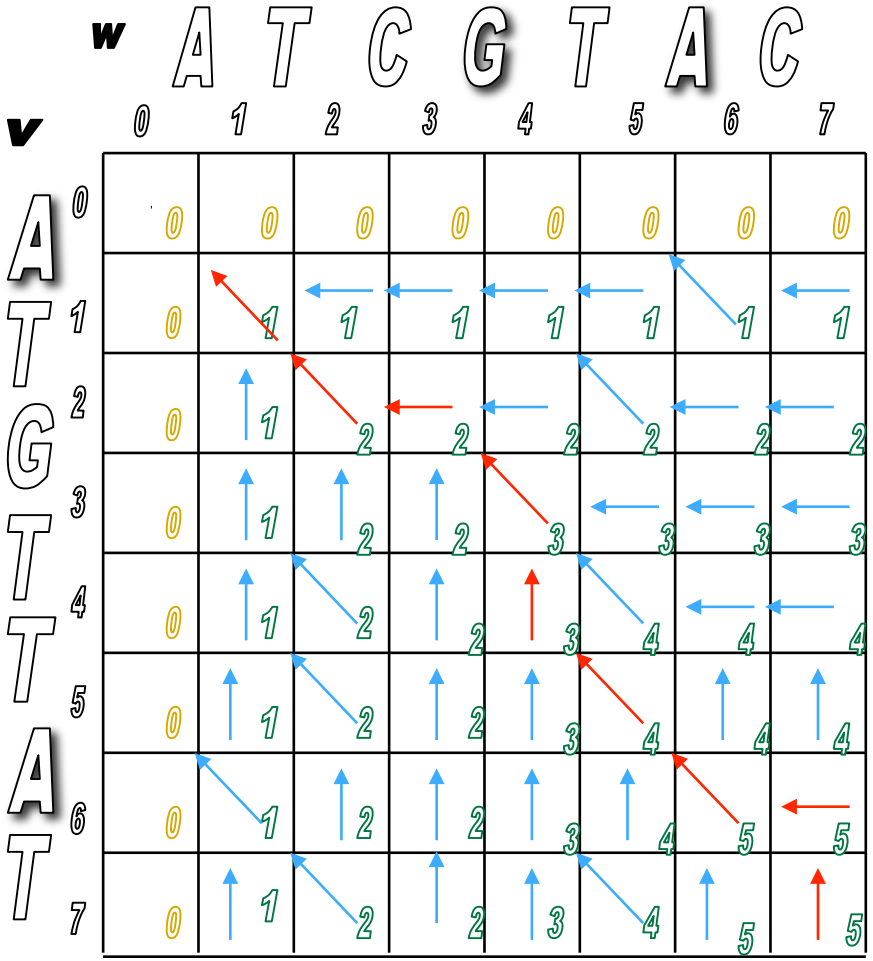



```
0 1 2 2 3 4 5 6 7 7
  A T _ G T T A _ T
  A T C G T _ A C _
0 1 2 3 4 5 5 6 6 7
```

Finding an Optimal Alignment

- LCS(v,w) created the alignment grid.

- Follow the arrows backwards from the sink to the source to obtain the path corresponding to an optimal alignment:




```
0 1 2 2 3 4 5 6 7 7
  A T _ G T T A _ T
  A T C G T _ A C _
0 1 2 3 4 5 5 6 6 7
```

Printing LCS: Backtracking

```
1.  PrintLCS(b,v,i,j)
2.      if  i = 0 or j = 0
3.          return
4.      if b_{i,j} = " ↖ "
5.          PrintLCS(b,v,i-1,j-1)
6.          print v_i
7.      else
8.          if b_{i,j} = " ↑ "
9.              PrintLCS(b,v,i-1,j)
10.         else
11.             PrintLCS(b,v,i,j-1)
```

LCS: Runtime

- It takes O(*nm*) time to fill in the *nxm* dynamic programming matrix: the pseudocode consists of a nested "for" loop inside of another "for" loop.

- This is a very positive result for a problem that appeared much more difficult initially.