Dynamic Programming

Part Two

Announcements

- On-time Problem Set Four graded; will be returned at end of lecture.
  - Late submissions should be graded by Monday; sorry about that!
- Problem Set Five due Monday, or Wednesday using a late period.
  - Heads-up: No late days on the final project. You can a late day on Problem Set Six, but that will overlap with the final project.

Outline for Today

- **String Algorithms**
  - Processing text, genomes, etc.
- **Sequence Alignment**
  - Determining the similarity of DNA strands.
- **Levenshtein Distance**
  - Checking how close two strings are.

Recap from Last Time

Dynamic Programming

- Dynamic programming is a technique useful for solving problems exhibiting the following properties:
  - **Overlapping subproblems:** Different branches of the recursion will reuse each other's work.
  - **Optimal substructure:** The optimal solution for one problem instance is formed from optimal solutions for smaller problems.
  - **Polynomial subproblems:** The number of subproblems is small enough to be evaluated in polynomial time.

A Correction From Last Time…

Theorem: OPT(k) satisfies the previous recurrence.

Proof: If $k = 0$, no people can be covered, so OPT(0) = 0. If $k = 1$, we can choose tower 1 (value v_1) or no towers (value 0), so OPT(1) = v_1. So consider $k > 1$.

If $k \in C$, then $k - 1 \notin C$. Then all towers in C besides k are within the first $k - 2$ towers, so $C(k - 2) \leq$ OPT($k - 2$). Also, C(k – 2) ≥ OPT(k – 2); otherwise we could replace all towers in C except k with an optimal set of the first k – 2 towers to improve C. Thus OPT(k) = v_k + OPT(k – 2).

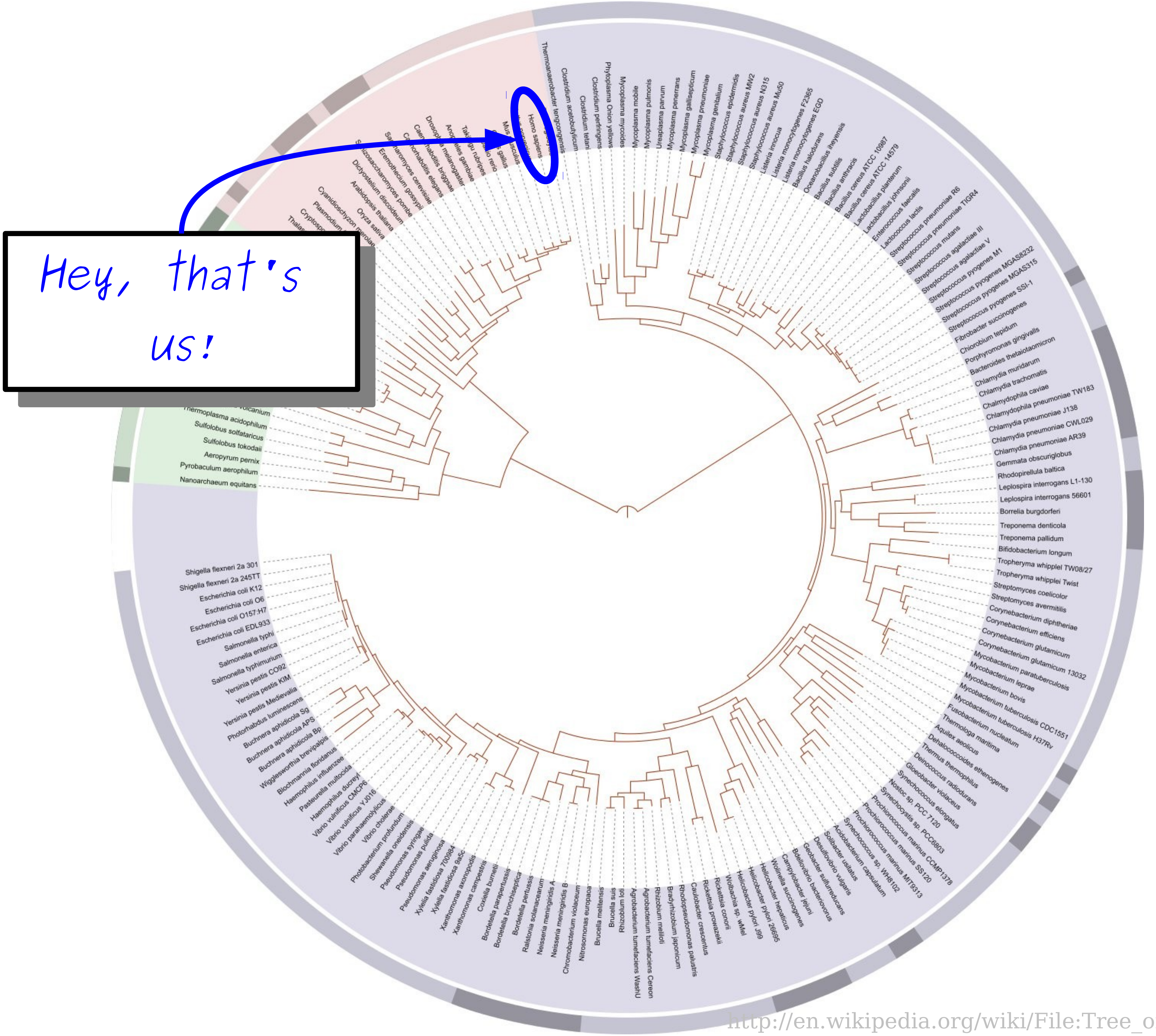
If $k \notin C$, all towers in C are in the first k – 1 towers. Thus $C(k - 1) \leq$ OPT(k – 1). Also, $C(k - 1) \geq$ OPT(k – 1); if not, we could improve C by replacing it with an optimal set of the first k – 1 towers. Therefore, OPT(k) = OPT(k – 1).

Since the optimal solution for k towers must be the better of these, OPT(k) = max{OPT(k – 1), v_k + OPT(k – 2)}. ■

Sequence Alignment

DNA Structure

- DNA strands consist of strings of **nucleotides**. There are four possible nucleotides: A, C, T, and G.
- Over time, mutations can occur in DNA strands:
  - **Point mutations**, where one nucleotide is replaced by another.
  - **Insertions**, where extra DNA is spliced in.
  - **Deletions**, where DNA is removed.
- Usually, the relative order of DNA letters remains the same.

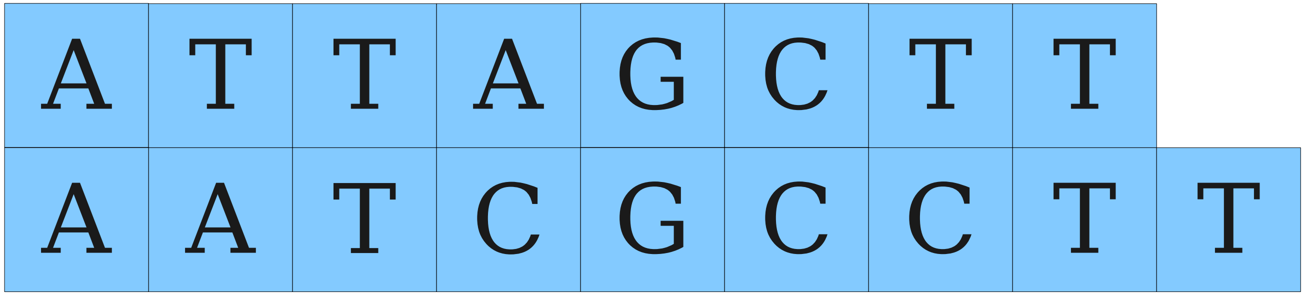


Hey, that's us!

http://en.wikipedia.org/wiki/File:Tree_of_life_SVG.svg

Aligning DNA Strands

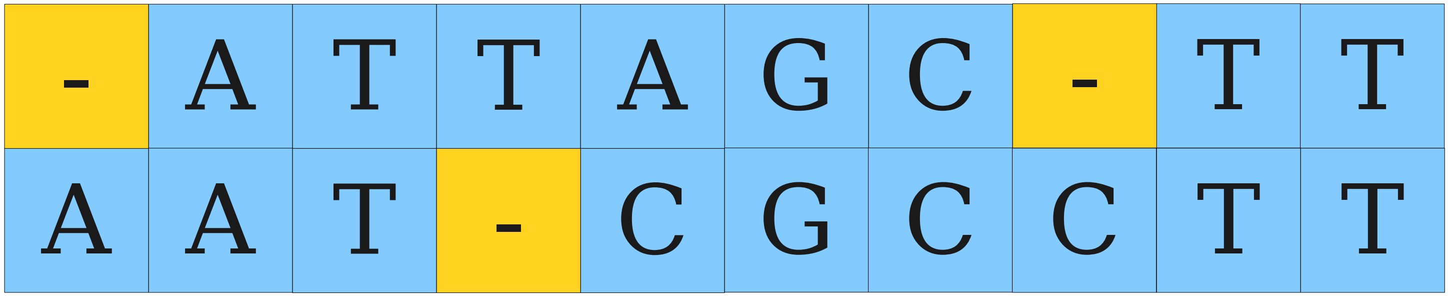
- DNA from related species often are similar, though not identical.
- We can try to **align** two DNA strands by inserting blanks (denoted by -) into the DNA strand.



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

Aligning DNA Strands

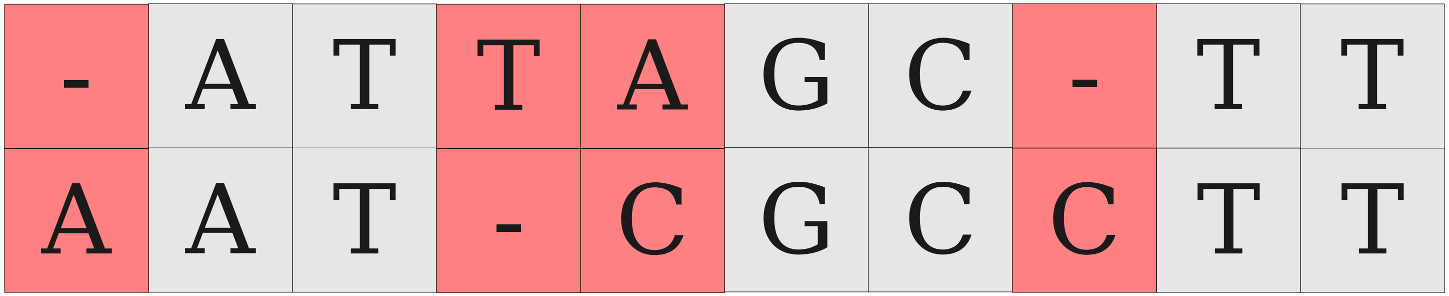
- DNA from related species often are similar, though not identical.
- We can try to **align** two DNA strands by inserting blanks (denoted by -) into the DNA strand.



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

Aligning DNA Strands

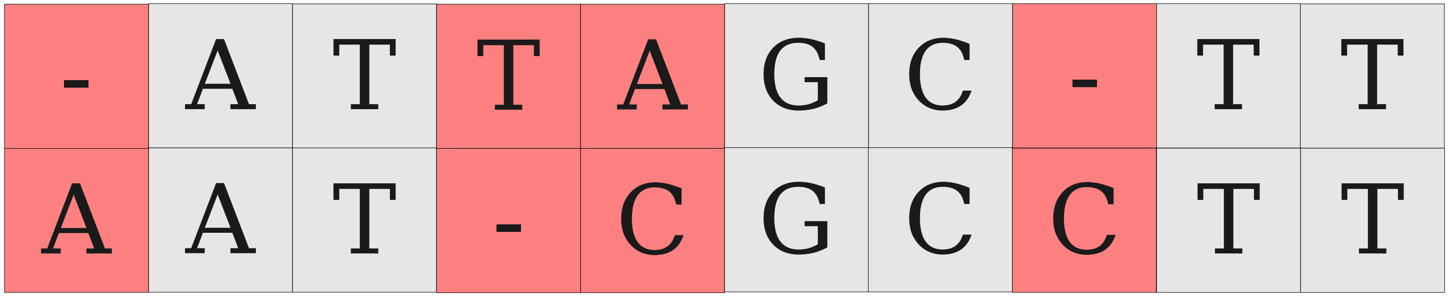
- DNA from related species often are similar, though not identical.
- We can try to **align** two DNA strands by inserting blanks (denoted by -) into the DNA strand.



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

Aligning DNA Strands

- DNA from related species often are similar, though not identical.
- We can try to **align** two DNA strands by inserting blanks (denoted by -) into the DNA strand.



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

- There is a cost associated with pairing a letter with a blank and with pairing two mismatched letters.

Sequence Alignment

- "Cost" of an alignment determined as follows:
  - For any characters *a* and *b*, cost of matching *a* and *b* is α_{ab}. This is usually 0 if the characters are the same and nonzero otherwise.
  - Cost of inserting a gap is δ.
- Assume α_{ab}'s and δ are external, fixed parameters.
- The **sequence alignment** problem is the following: find the alignment of the sequences with the least total cost.

An Insight

- The last column in the alignment must
  - Match the last characters from both strings:

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

  - Insert a gap up top:

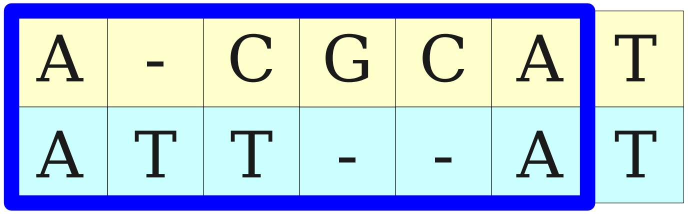
G	G	C	T	C	T	-
G	G	-	C	-	G	T

  - Insert a gap on bottom:

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

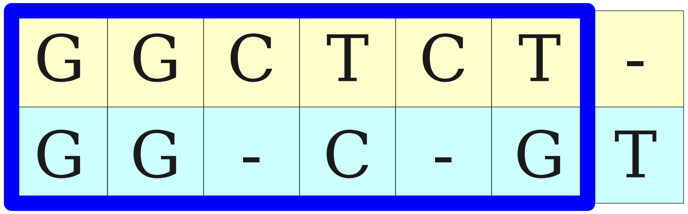
An Insight

- The last column in the alignment must
  - Match the last characters from both strings:



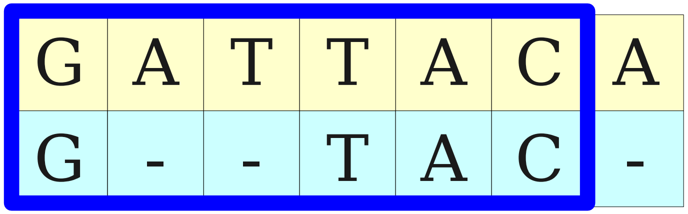
A	-	C	G	C	A	T
A	T	T	-	-	A	T

  - Insert a gap up top:



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

  - Insert a gap on bottom:



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

Some Notation

- Suppose we want to align the first i characters of A and the first j characters of B. (Denote this $A[1, i]$ and B[1, j]
- Let OPT(i, j) denote the optimal cost of such an alignment.
- **Claim:** OPT(i, j) satisfies the following:

Some Notation

- Suppose we want to align the first i characters of A and the first j characters of B. (Denote this $A[1, i]$ and $B[1, j]$
- Let OPT(i, j) denote the optimal cost of such an alignment.
- **Claim:** OPT(i, j) satisfies the following:

$$\mathrm{OPT}(i,j)=\begin{cases} \textbf{?} & \begin{array}{l} if\ i=0 \\ if\ j=0 \end{array} \\ min\begin{Bmatrix} \delta+OPT(i-1,j), \\ \delta+OPT(i,j-1), \\ \alpha_{A[i]B[j]}+OPT(i-1,j-1) \end{Bmatrix} & otherwise \end{cases}$$

Some Notation

- Suppose we want to align the first i characters of A and the first j characters of B. (Denote this $A[1, i]$ and $B[1, j]$
- Let OPT(i, j) denote the optimal cost of such an alignment.
- **Claim:** OPT(i, j) satisfies the following:

$$\mathrm{OPT}(i,j)=\begin{cases} j\delta & \text{if } i=0 \\ i\delta & \text{if } j=0 \\ \min\begin{Bmatrix} \delta+OPT(i-1,j), \\ \delta+OPT(i,j-1), \\ \alpha_{A[i]B[j]}+OPT(i-1,j-1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

Theorem: OPT(i, j) satisfies the previous recurrence.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj. By a similar argument, if $j = 0$, then OPT(i, j) = δi.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj. By a similar argument, if $j = 0$, then OPT(i, j) = δi. Otherwise, $i > 0$ and $j > 0$.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj. By a similar argument, if $j = 0$, then OPT(i, j) = δi. Otherwise, $i > 0$ and $j > 0$. Consider an optimal alignment M^* of $A[1, i]$ and $B[1, j]$.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj. By a similar argument, if $j = 0$, then OPT(i, j) = δi. Otherwise, $i > 0$ and $j > 0$. Consider an optimal alignment M^* of $A[1, i]$ and $B[1, j]$. There are three possibilities:

Case 1: M^* pairs $A[i]$ and $B[j]$.

Case 2: M^* pairs $A[i]$ with a blank.

Case 3: M^* pairs $B[j]$ with a blank.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj. By a similar argument, if $j = 0$, then OPT(i, j) = δi. Otherwise, $i > 0$ and $j > 0$. Consider an optimal alignment M^* of $A[1, i]$ and $B[1, j]$. There are three possibilities:

Case 1: M^* pairs $A[i]$ and $B[j]$. The rest of M^* aligns $A[1, i-1]$ and $B[1, j-1]$ and we claim it optimally aligns them; otherwise, changing M^* to optimally align $A[1, i-1]$ and $B[1, j-1]$ decreases the cost of M^*.

Case 2: M^* pairs $A[i]$ with a blank.

Case 3: M^* pairs $B[j]$ with a blank.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj. By a similar argument, if $j = 0$, then OPT(i, j) = δi. Otherwise, $i > 0$ and $j > 0$. Consider an optimal alignment M^* of $A[1, i]$ and $B[1, j]$. There are three possibilities:

Case 1: M^* pairs $A[i]$ and $B[j]$. The rest of M^* aligns $A[1, i - 1]$ and $B[1, j - 1]$ and we claim it optimally aligns them; otherwise, changing M^* to optimally align $A[1, i - 1]$ and $B[1, j - 1]$ decreases the cost of M^*. Therefore, $\text{OPT}(i, j) = \alpha_{A[i]B[j]} + \text{OPT}(i - 1, j - 1)$.

Case 2: M^* pairs $A[i]$ with a blank.

Case 3: M^* pairs $B[j]$ with a blank.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj. By a similar argument, if $j = 0$, then OPT(i, j) = δi. Otherwise, $i > 0$ and $j > 0$. Consider an optimal alignment M^* of $A[1, i]$ and $B[1, j]$. There are three possibilities:

Case 1: M^* pairs $A[i]$ and $B[j]$. The rest of M^* aligns $A[1, i-1]$ and $B[1, j-1]$ and we claim it optimally aligns them; otherwise, changing M^* to optimally align $A[1, i-1]$ and $B[1, j-1]$ decreases the cost of M^*. Therefore, $\text{OPT}(i, j) = \alpha_{A[i]B[j]} + \text{OPT}(i-1, j-1)$.

Case 2: M^* pairs $A[i]$ with a blank. The rest of M^* aligns $A[1, i-1]$ and $B[1, j]$ and we claim it optimally aligns them; otherwise, changing M^* to optimally align $A[1, i-1]$ and $B[1, j]$ decreases the cost of M^*.

Case 3: M^* pairs $B[j]$ with a blank.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj. By a similar argument, if $j = 0$, then OPT(i, j) = δi. Otherwise, $i > 0$ and $j > 0$. Consider an optimal alignment M^* of $A[1, i]$ and $B[1, j]$. There are three possibilities:

Case 1: M^* pairs $A[i]$ and $B[j]$. The rest of M^* aligns $A[1, i - 1]$ and $B[1, j - 1]$ and we claim it optimally aligns them; otherwise, changing M^* to optimally align $A[1, i - 1]$ and $B[1, j - 1]$ decreases the cost of M^*. Therefore, OPT(i, j) = $\alpha_{A[i]B[j]}$ + OPT($i - 1$, $j - 1$).

Case 2: M^* pairs $A[i]$ with a blank. The rest of M^* aligns $A[1, i - 1]$ and $B[1, j]$ and we claim it optimally aligns them; otherwise, changing M^* to optimally align $A[1, i - 1]$ and $B[1, j]$ decreases the cost of M^*. Thus OPT(i, j) = δ + OPT($i - 1$, j).

Case 3: M^* pairs $B[j]$ with a blank.

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj. By a similar argument, if $j = 0$, then OPT(i, j) = δi. Otherwise, $i > 0$ and $j > 0$. Consider an optimal alignment M^* of $A[1, i]$ and $B[1, j]$. There are three possibilities:

Case 1: M^* pairs $A[i]$ and $B[j]$. The rest of M^* aligns $A[1, i - 1]$ and $B[1, j - 1]$ and we claim it optimally aligns them; otherwise, changing M^* to optimally align $A[1, i - 1]$ and $B[1, j - 1]$ decreases the cost of M^*. Therefore, OPT(i, j) = $\alpha_{A[i]B[j]}$ + OPT($i - 1$, $j - 1$).

Case 2: M^* pairs $A[i]$ with a blank. The rest of M^* aligns $A[1, i - 1]$ and $B[1, j]$ and we claim it optimally aligns them; otherwise, changing M^* to optimally align $A[1, i - 1]$ and $B[1, j]$ decreases the cost of M^*. Thus OPT(i, j) = δ + OPT($i - 1$, j).

Case 3: M^* pairs $B[j]$ with a blank. By a similar argument to the previous case, we have OPT(i, j) = δ + OPT(i, $j - 1$).

Theorem: OPT(i, j) satisfies the previous recurrence.

Proof: If $i = 0$, the only way to match $A[1, 0]$ and $B[1, j]$ is to insert j gaps into A to match the j characters of B. This has cost δj, so OPT(i, j) = δj. By a similar argument, if $j = 0$, then OPT(i, j) = δi. Otherwise, $i > 0$ and $j > 0$. Consider an optimal alignment M^* of $A[1, i]$ and $B[1, j]$. There are three possibilities:

Case 1: M^* pairs $A[i]$ and $B[j]$. The rest of M^* aligns $A[1, i-1]$ and $B[1, j-1]$ and we claim it optimally aligns them; otherwise, changing M^* to optimally align $A[1, i-1]$ and $B[1, j-1]$ decreases the cost of M^*. Therefore, OPT(i, j) = $\alpha_{A[i]B[j]}$ + OPT($i-1, j-1$).

Case 2: M^* pairs $A[i]$ with a blank. The rest of M^* aligns $A[1, i-1]$ and $B[1, j]$ and we claim it optimally aligns them; otherwise, changing M^* to optimally align $A[1, i-1]$ and $B[1, j]$ decreases the cost of M^*. Thus OPT(i, j) = δ + OPT($i-1, j$).

Case 3: M^* pairs $B[j]$ with a blank. By a similar argument to the previous case, we have OPT(i, j) = δ + OPT($i, j-1$).

Since the optimal solution must be one of these three options, we have OPT(i, j) = min{$\alpha_{A[i]B[j]}$ + OPT($i-1, j-1$), δ + OPT($i, j-1$), δ + OPT($i-1, j$)}. ■

Evaluating the Recurrence

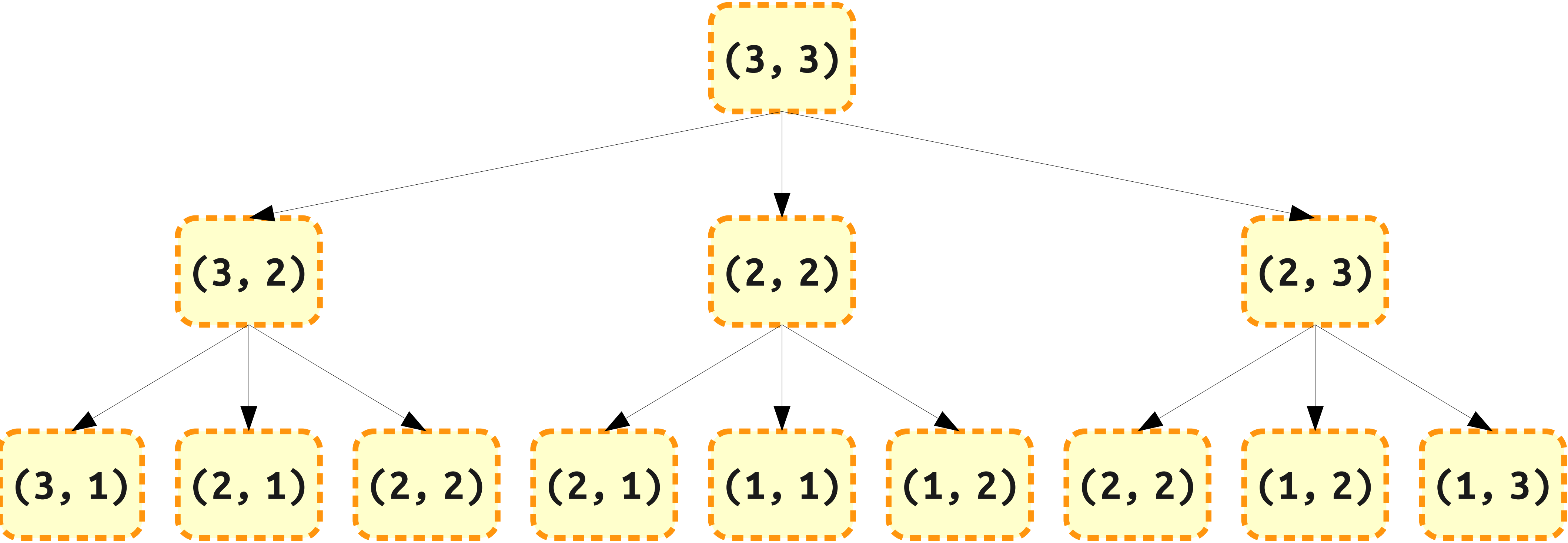
- If we can evaluate this recurrence:

$$\mathrm{OPT}(i,j)=\begin{cases} j\delta & \text{if } i=0 \\ i\delta & \text{if } j=0 \\ \min\begin{cases}\delta+OPT(i-1,j), \\ \delta+OPT(i,j-1), \\ \alpha_{A[i]B[j]}+OPT(i-1,j-1)\end{cases} & \text{otherwise}\end{cases}$$

  We can evaluate the cost of an optimal alignment.

- What happens if we evaluate it directly?

The Recursion Tree



(3, 3)

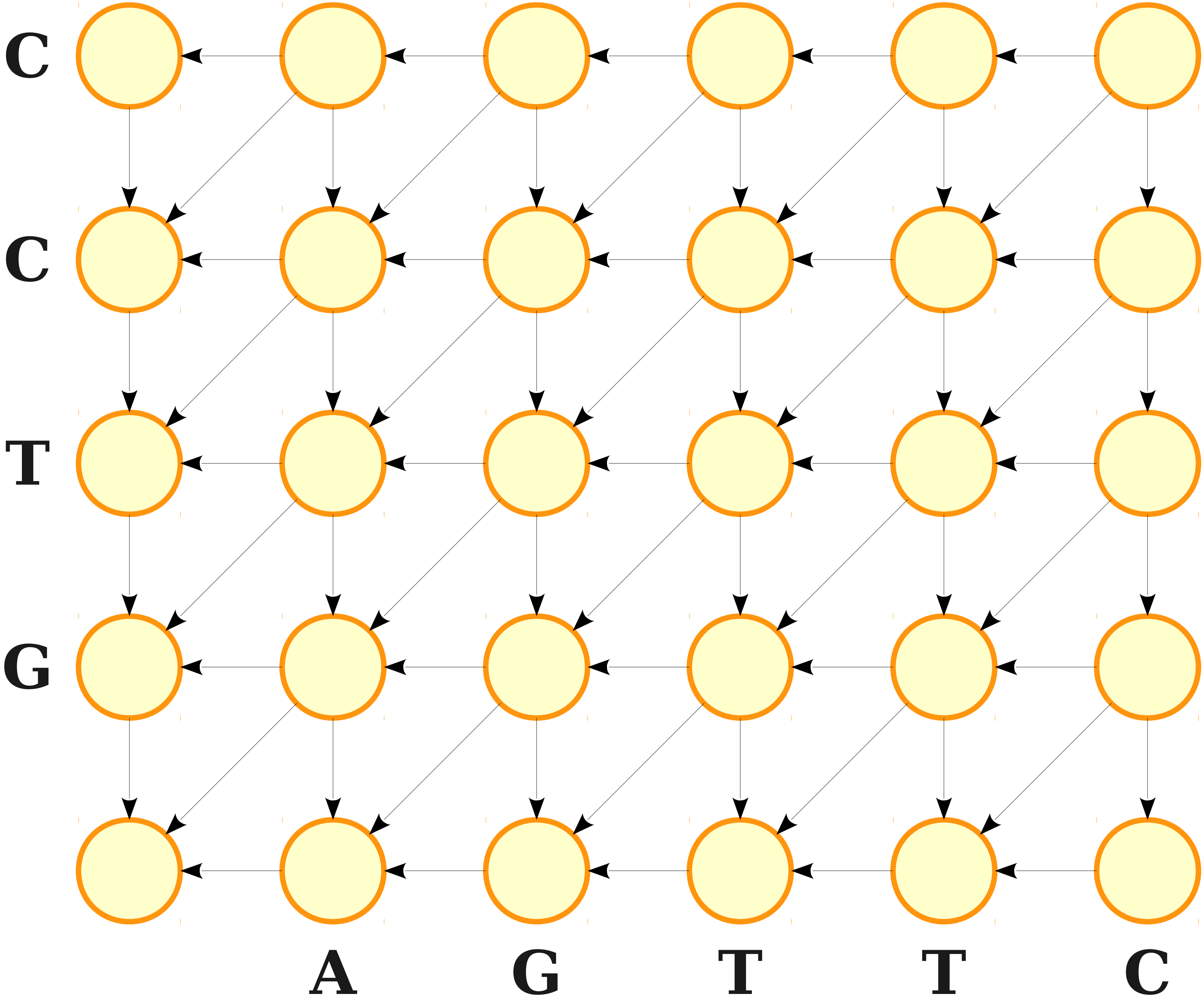
(3, 2)
(2, 2)
(2, 3)

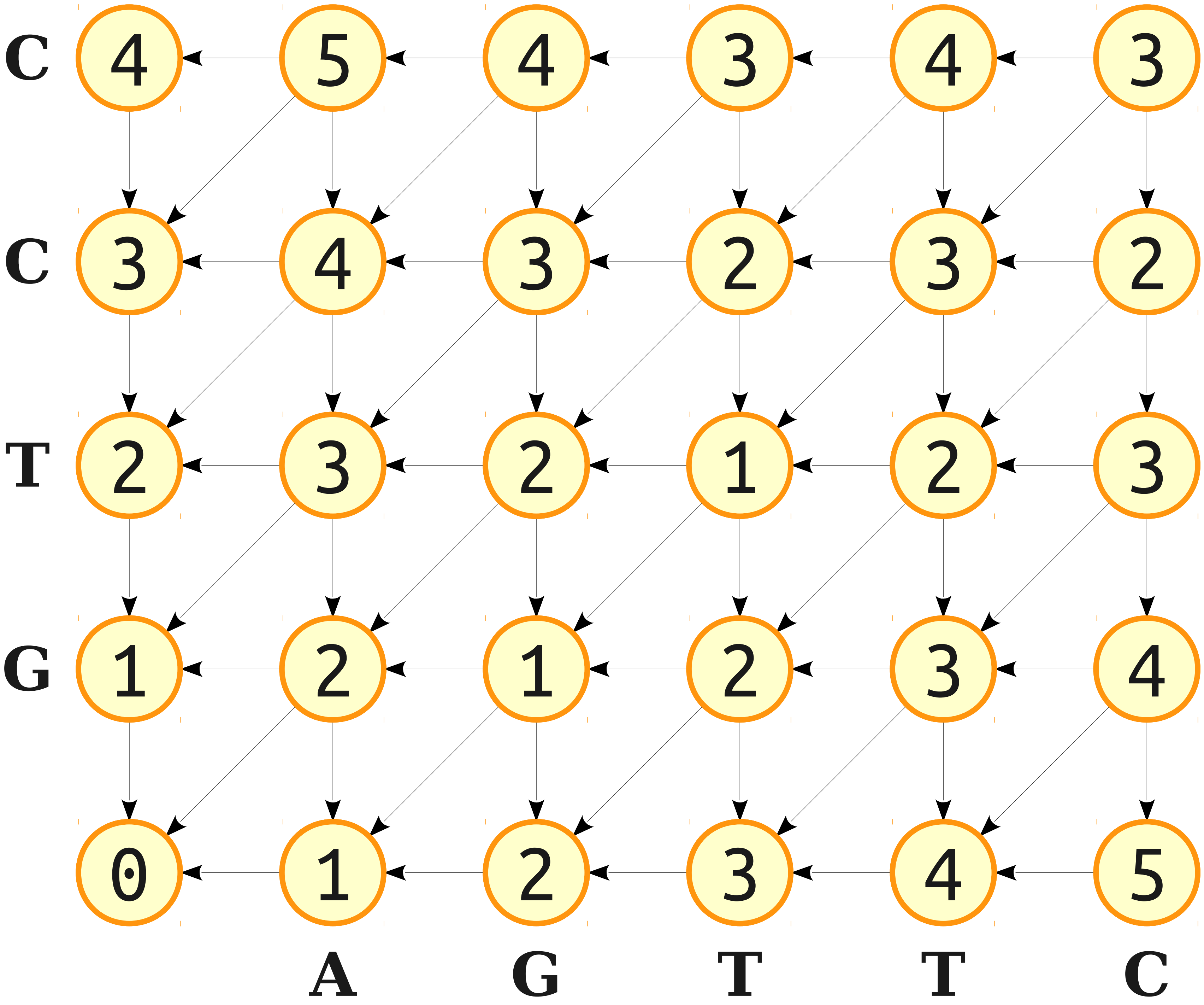
(3, 1)
(2, 1)
(2, 2)
(2, 1)
(1, 1)
(1, 2)
(2, 2)
(1, 2)
(1, 3)

...

Dynamic Programming

- Do we have these three properties?
  - **Overlapping subproblems**
  - **Optimal substructure**
  - **Polynomial subproblems**
- Time to bring out the dynamic programming solution!


C
C
T
G
A
G
T
T
C



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

The Algorithm

- Create an $(|A| + 1) \times (|B| + 1)$ grid DP.
- For $i = 0$ to $|A|$, set $\mathrm{DP}[i, 0] = \delta i$.
- For $j = 0$ to $|B|$, set $\mathrm{DP}[0, j] = \delta j$.
- For $i = 1$ to $|A|$:
  - For $j = 1$ to $|B|$:
    - Set $\mathrm{DP}[i][j]$ to the minimum of
      - $\mathrm{DP}[i - 1][j] + \delta$
      - $\mathrm{DP}[i][j - 1] + \delta$
      - $\mathrm{DP}[i - 1][j - 1] + \alpha_{\mathrm{A}[i]\mathrm{B}[j]}$

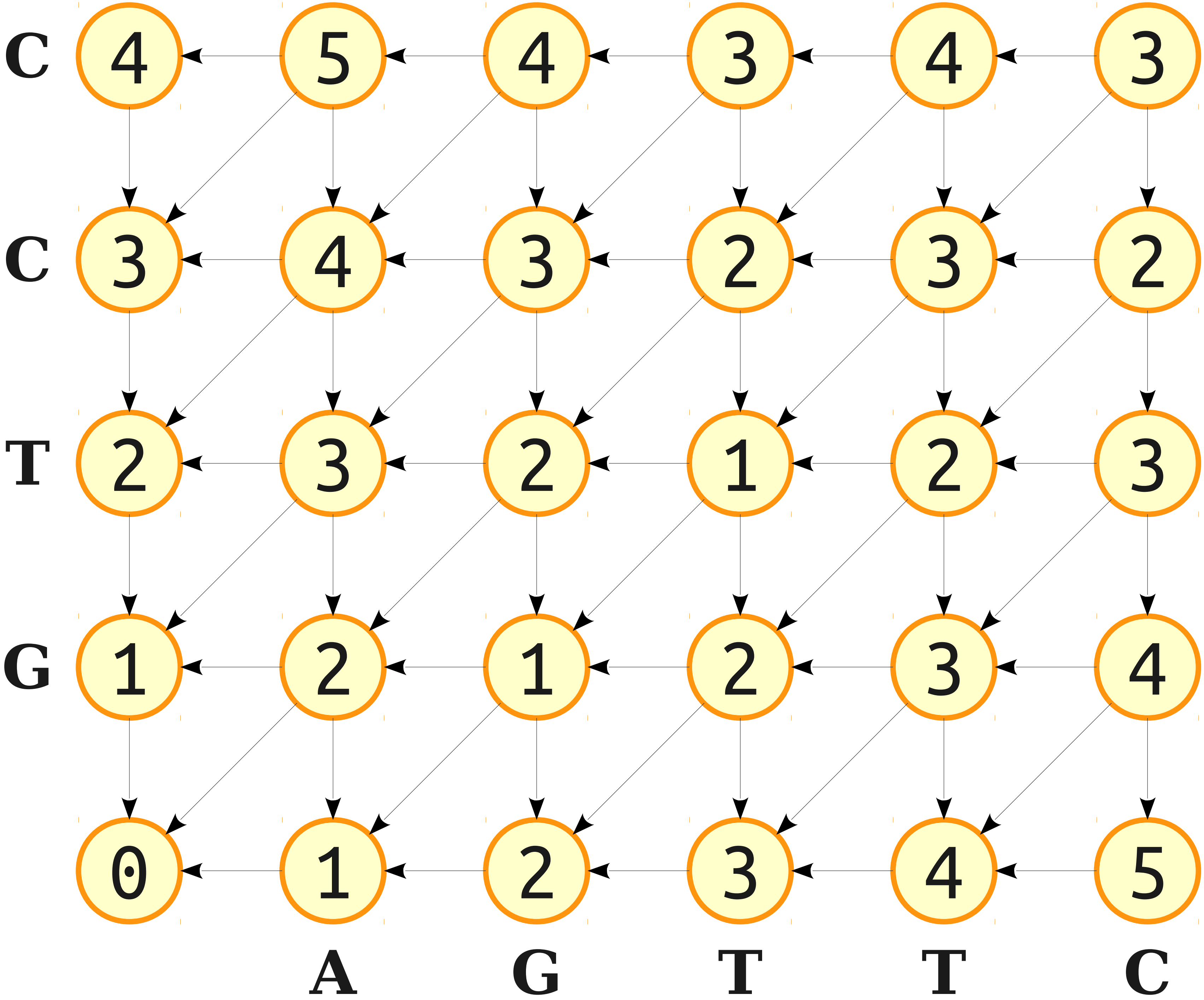
The Algorithm

- Create an $(|A| + 1) \times (|B| + 1)$ grid DP.
- For i = 0 to $|A|$, set DP[i, 0] = δi.
- For j = 0 to $|B|$, set DP[0, j] = δj.
- For i = 1 to $|A|$:
  - For j = 1 to $|B|$:
    - – Set DP[i][j] to the minimum of
      - DP[i – 1][j]
      - DP[i][j – 1]
      - DP[i – 1][j –

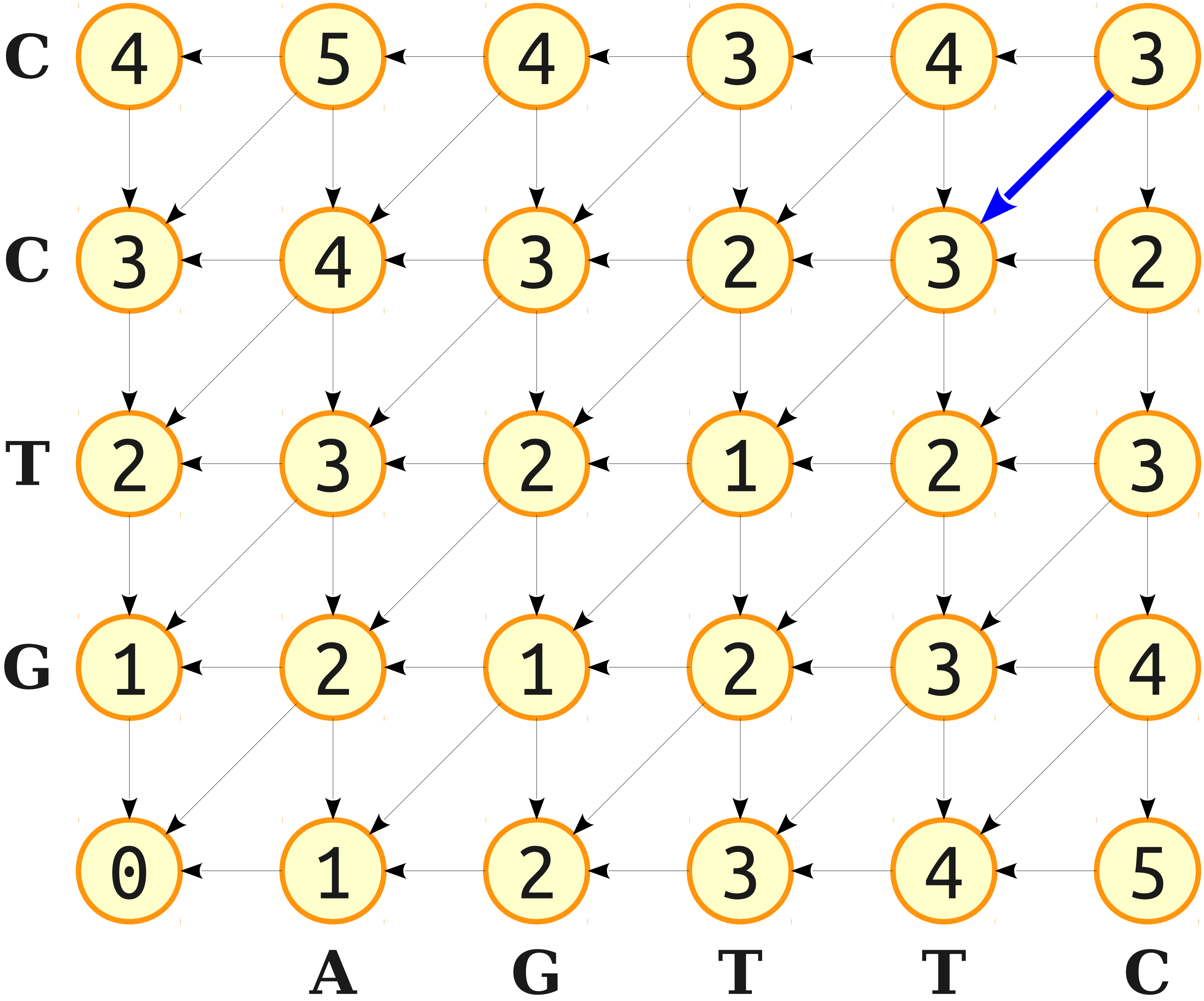
Question: Could we also go column-by-column instead of row-by-row?

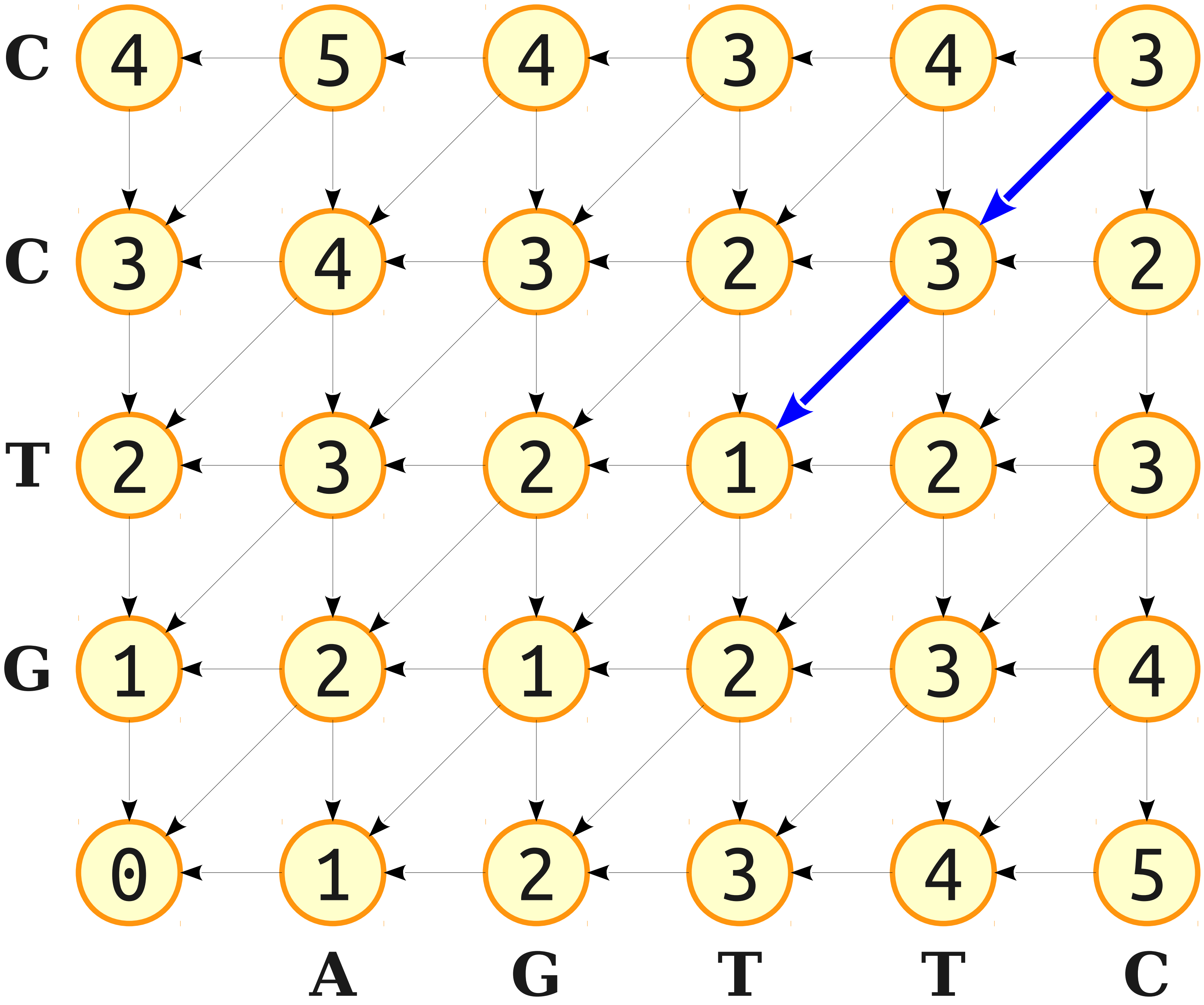
Analyzing the Algorithm

- Let $m = |A|$ and $n = |B|$.
- What is the runtime of this algorithm?
  - **$O(mn)$**
- What is the space usage of this algorithm?
  - **$O(mn)$**
- That's *way* less than the total number of possible alignments!

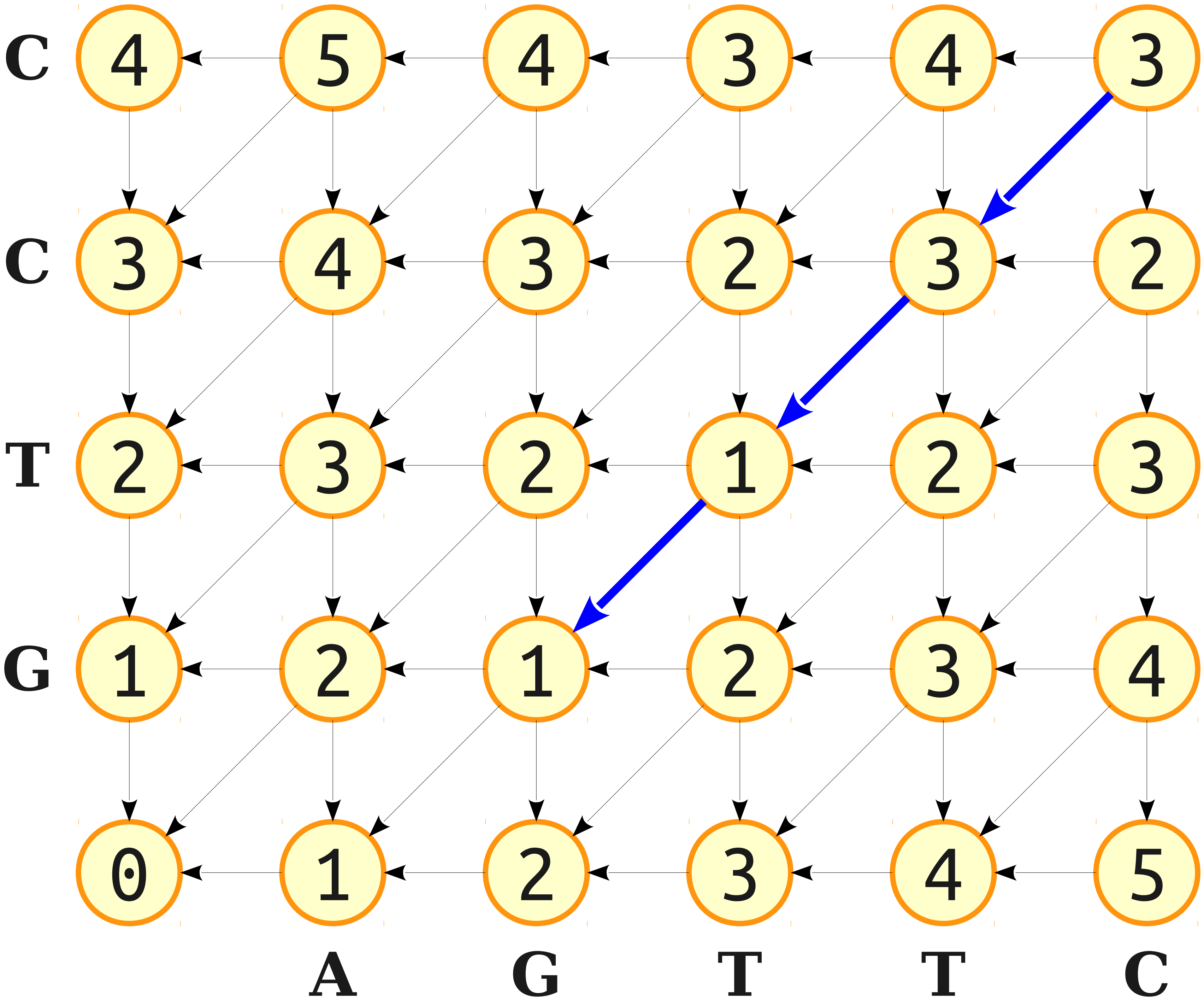


C	4	5	4	3	4	3
C	3	4	3	2	3	2
T	2	3	2	1	2	3
G	1	2	1	2	3	4
	0	1	2	3	4	5
		A	G	T	T	C

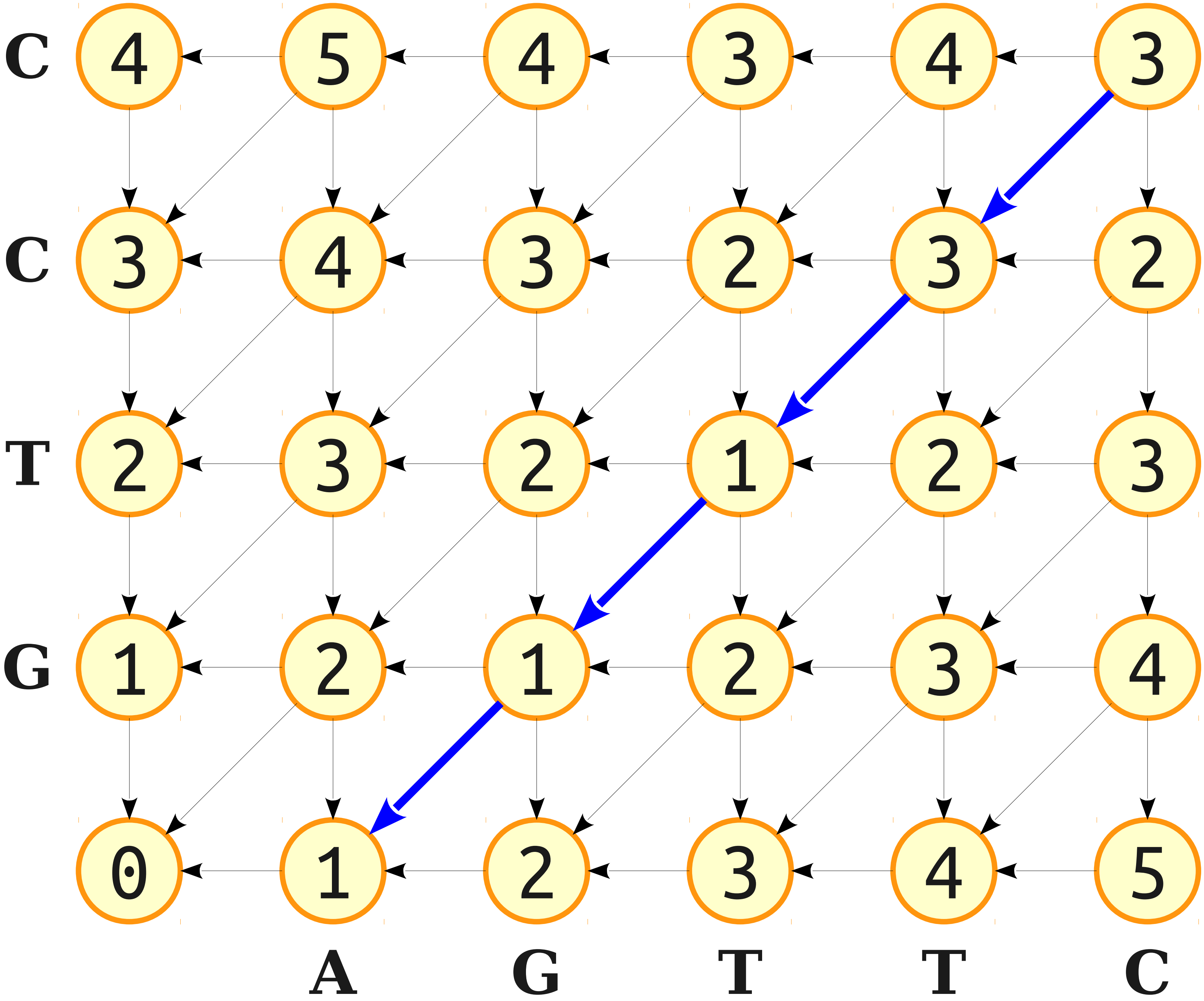

C 4 5 4 3 4 3
C 3 4 3 2 3 2
T 2 3 2 1 2 3
G 1 2 1 2 3 4
0 1 2 3 4 5
A G T T C

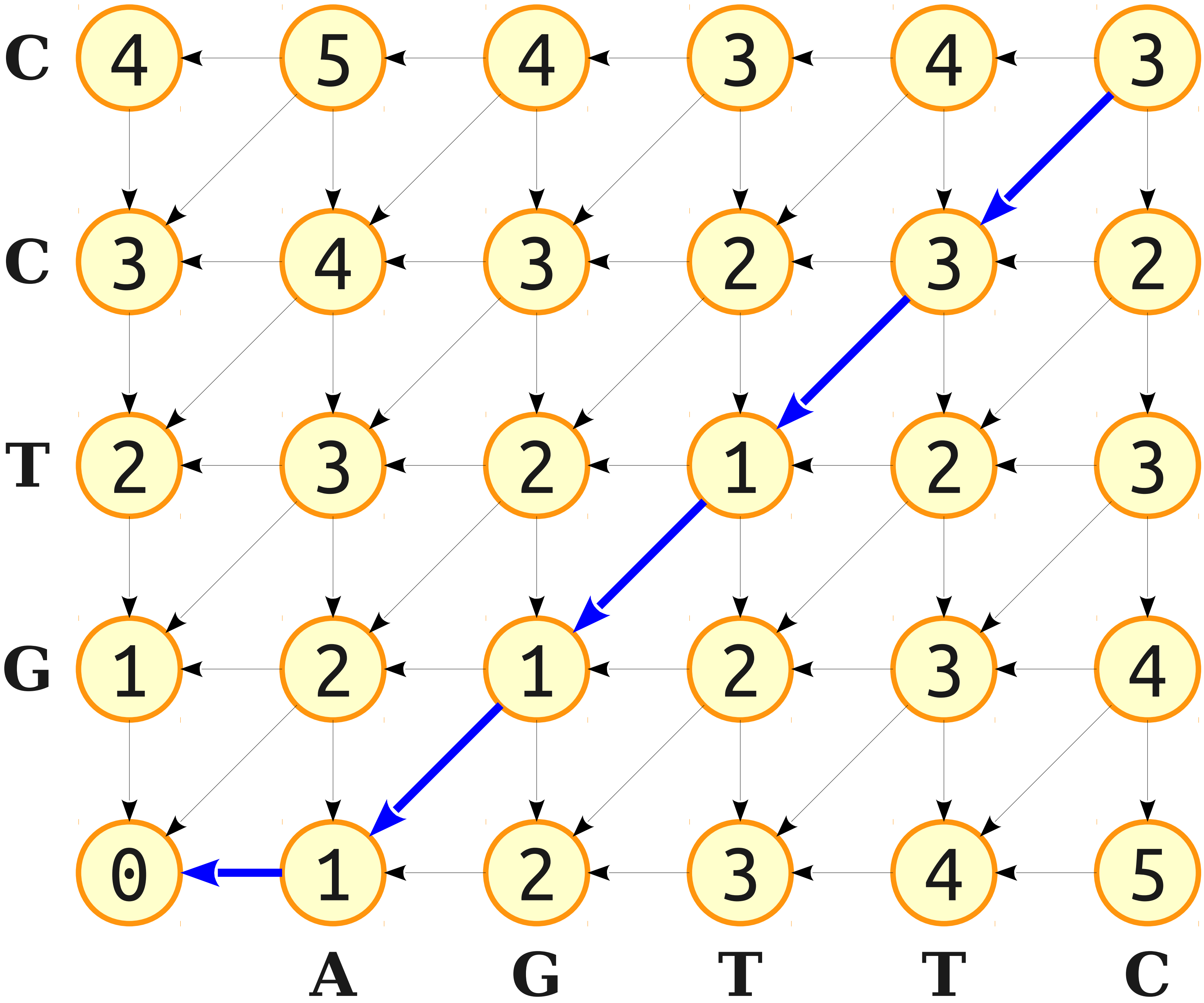


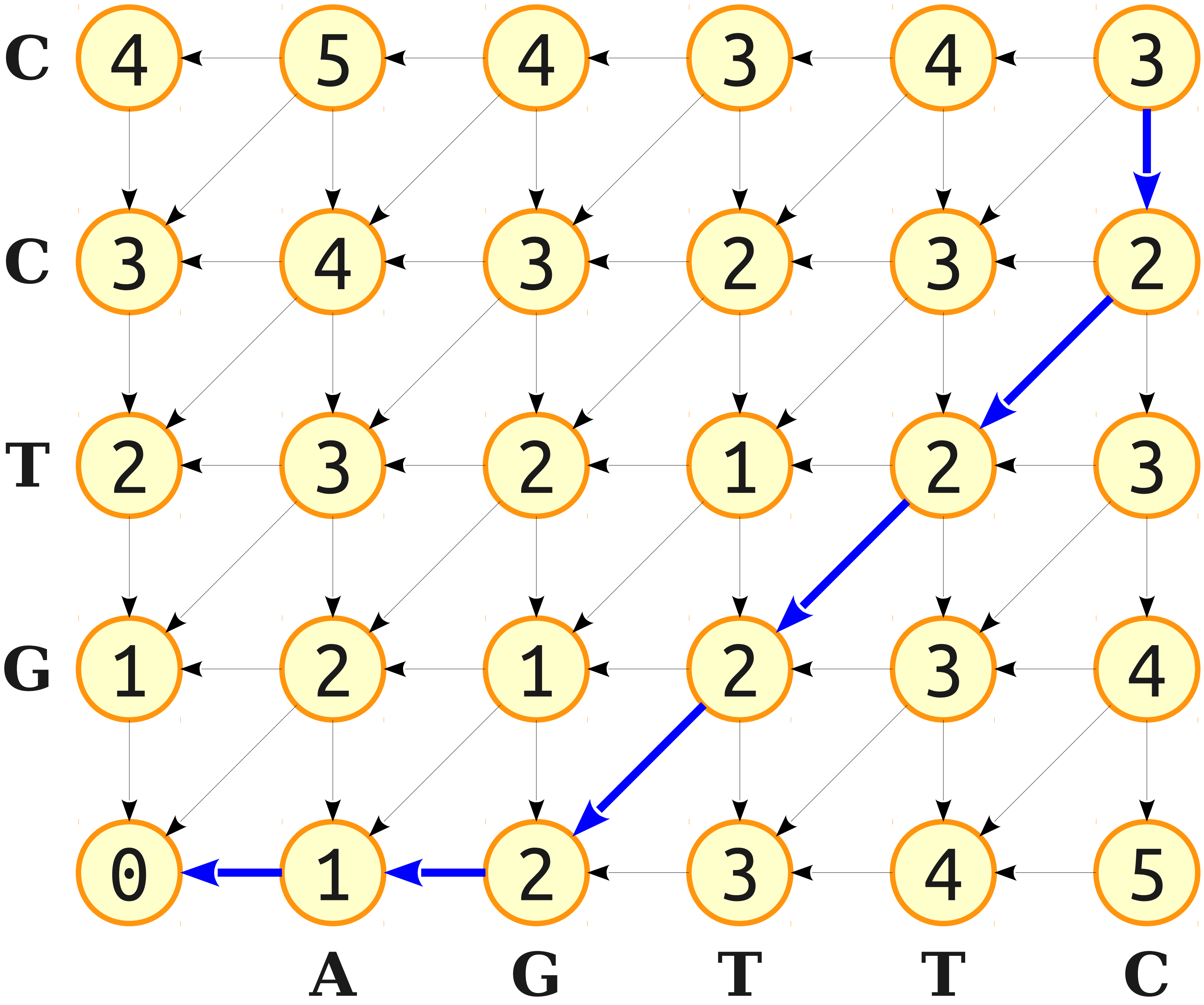
C
C
T
G
4
5
4
3
4
3
3
4
3
2
3
2
2
3
2
1
2
3
1
2
1
2
3
4
0
1
2
3
4
5
A
G
T
T
C



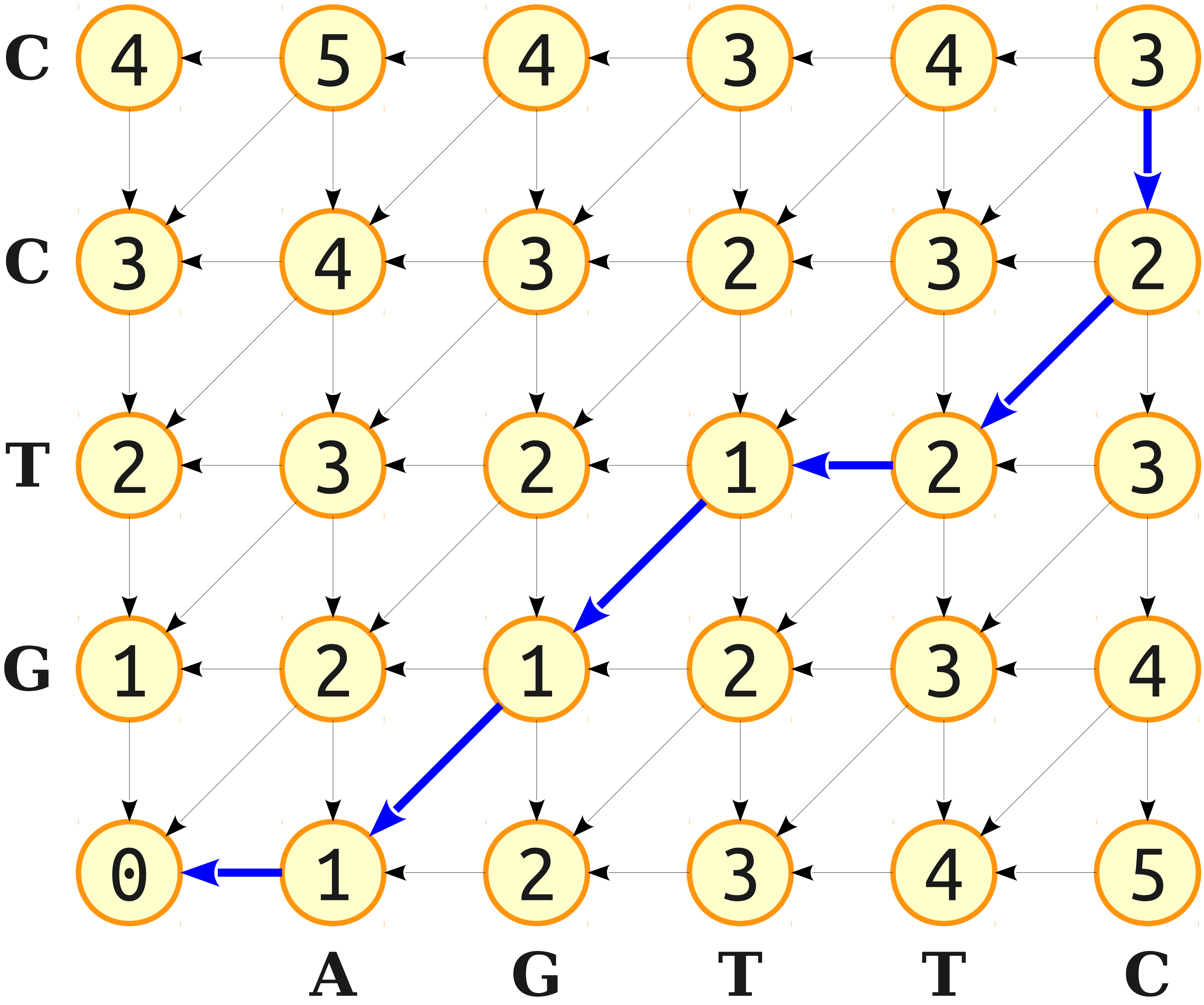
C	4	5	4	3	4	3
C	3	4	3	2	3	2
T	2	3	2	1	2	3
G	1	2	1	2	3	4
	0	1	2	3	4	5
		A	**G**	**T**	**T**	**C**

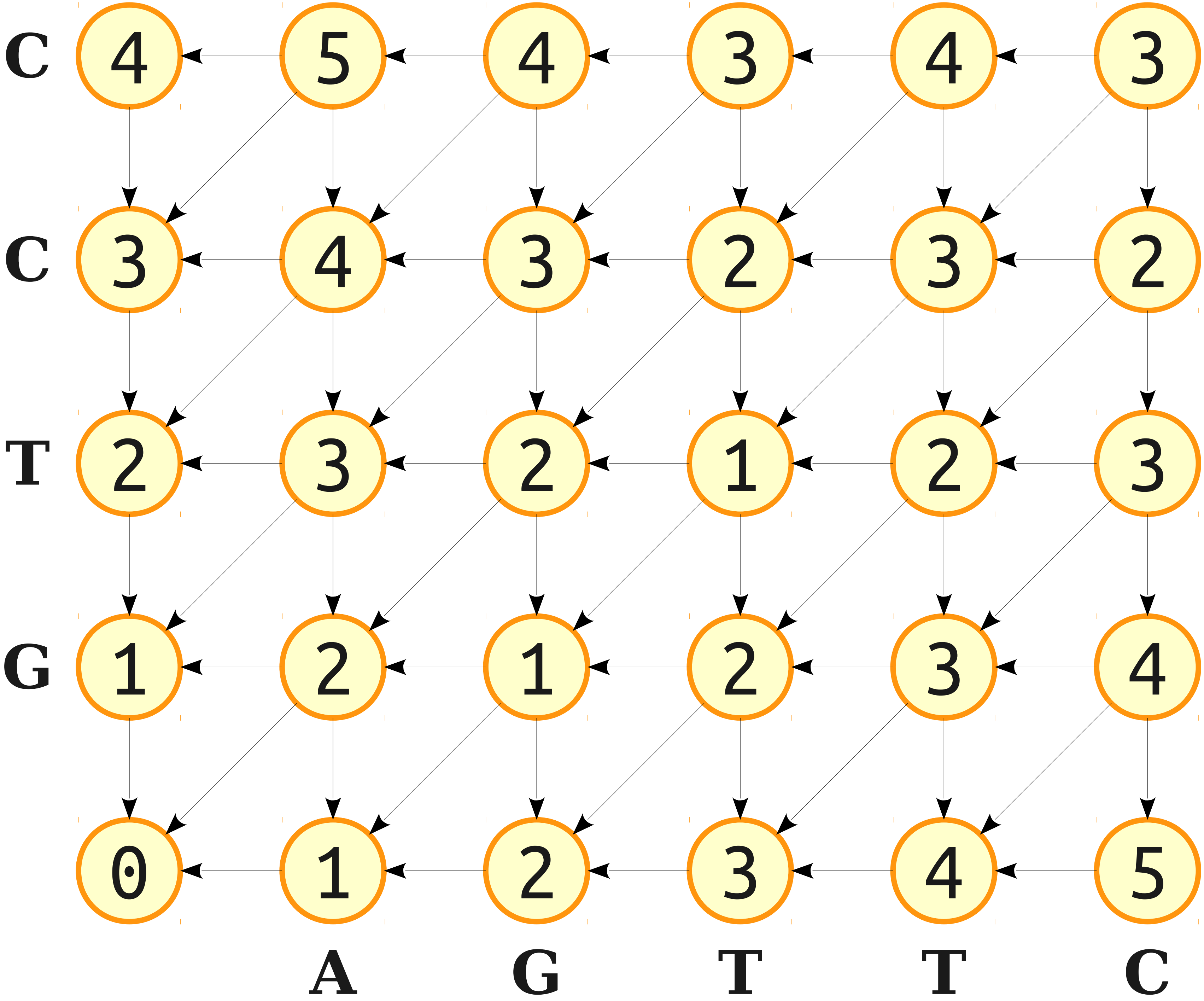

C 4 5 4 3 4 3
C 3 4 3 2 3 2
T 2 3 2 1 2 3
G 1 2 1 2 3 4
0 1 2 3 4 5
A G T T C

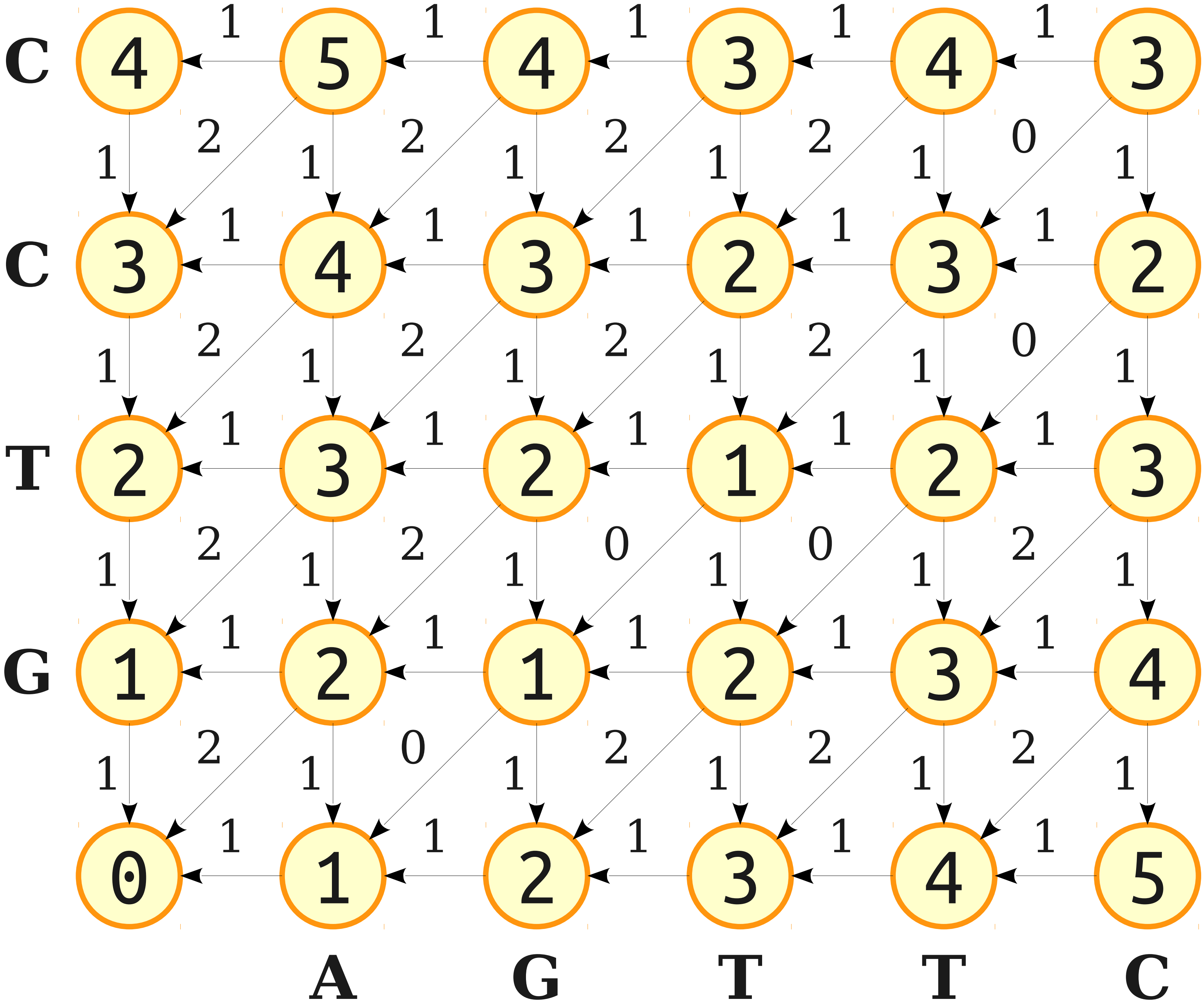

C 4 5 4 3 4 3
C 3 4 3 2 3 2
T 2 3 2 1 2 3
G 1 2 1 2 3 4
0 1 2 3 4 5
A G T T C

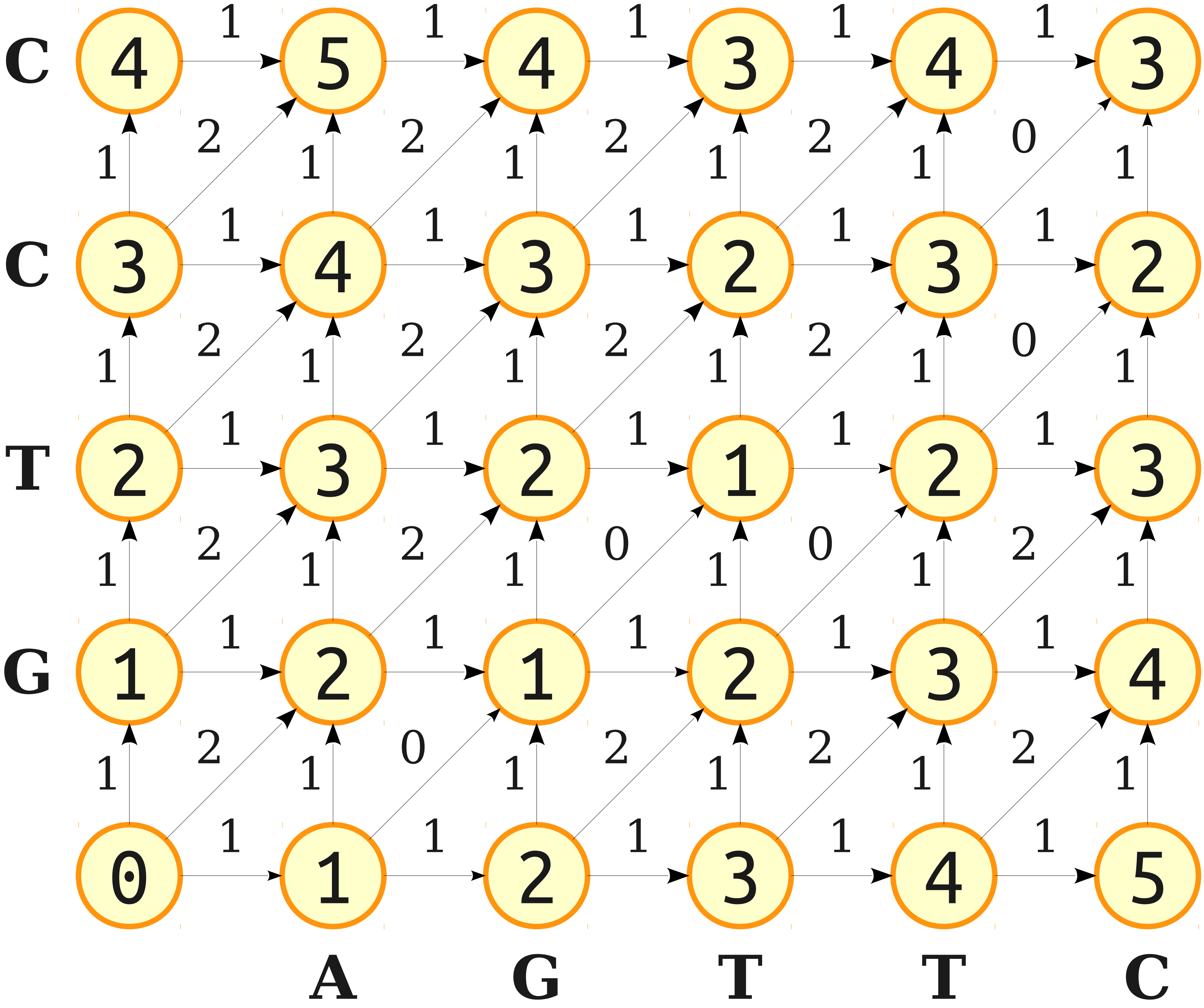


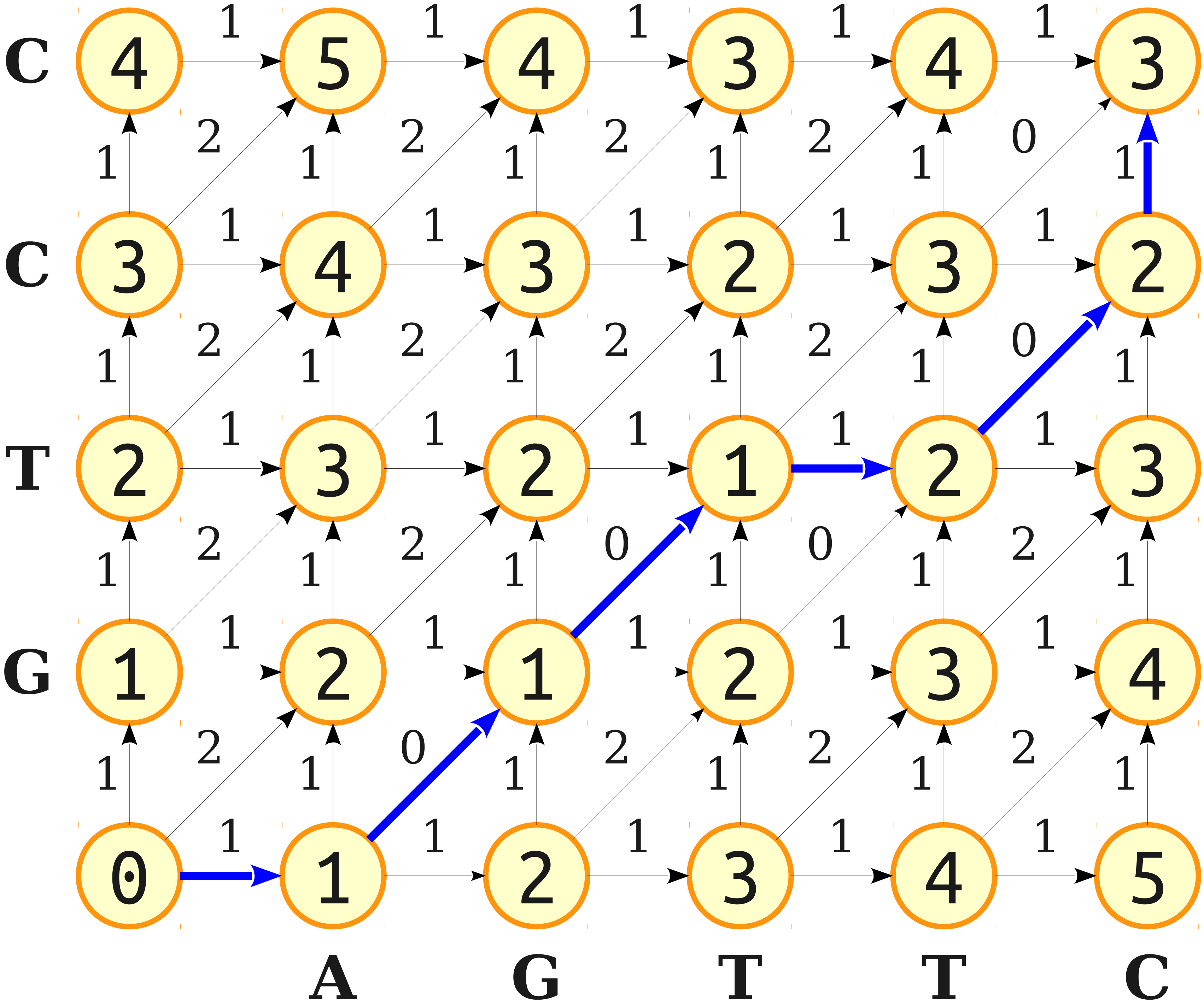
C
C
T
G
A
G
T
T
C
4 5 4 3 4 3
3 4 3 2 3 2
2 3 2 1 2 3
1 2 1 2 3 4
0 1 2 3 4 5

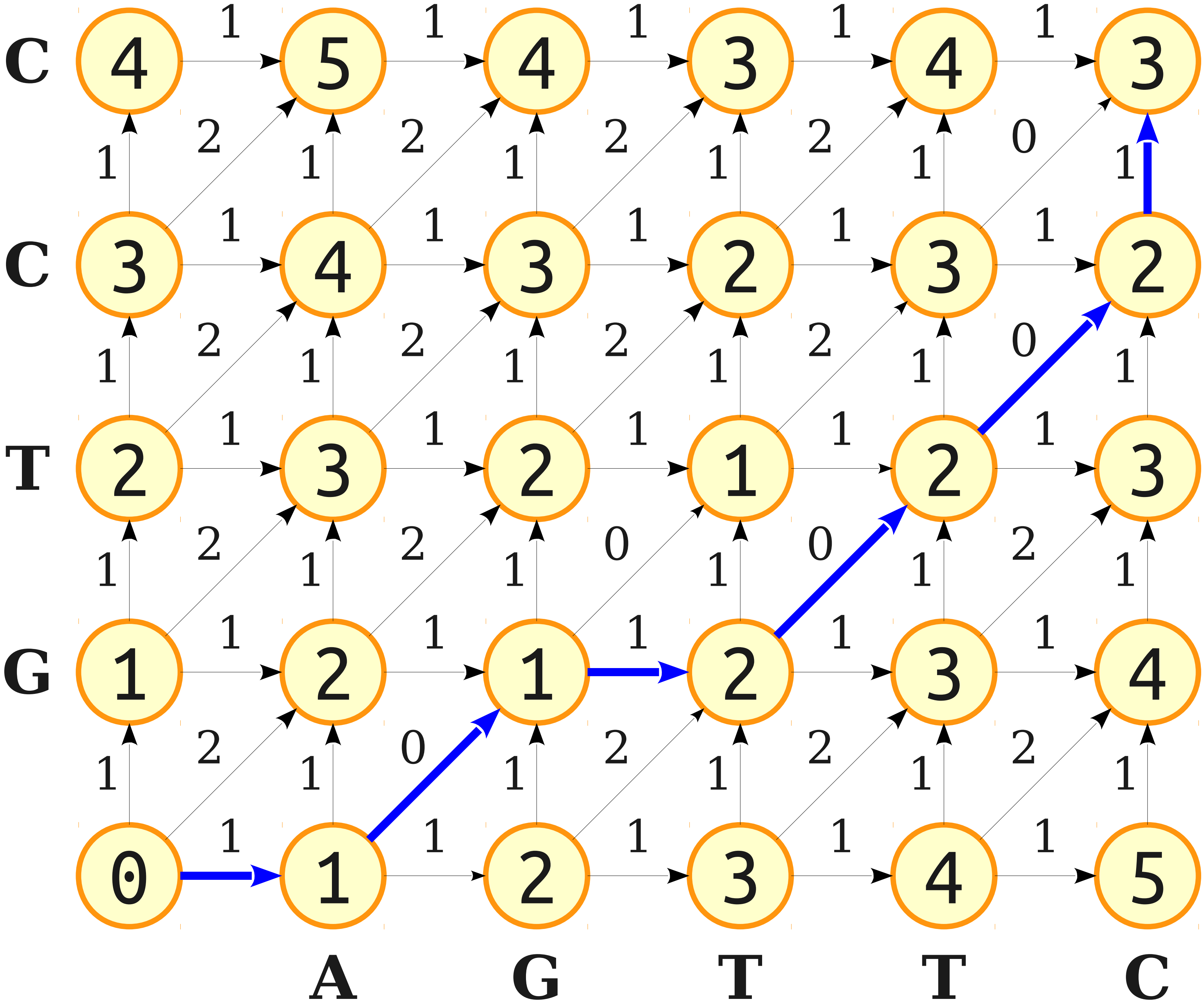

C 4 5 4 3 4 3
C 3 4 3 2 3 2
T 2 3 2 1 2 3
G 1 2 1 2 3 4
0 1 2 3 4 5
A G T T C

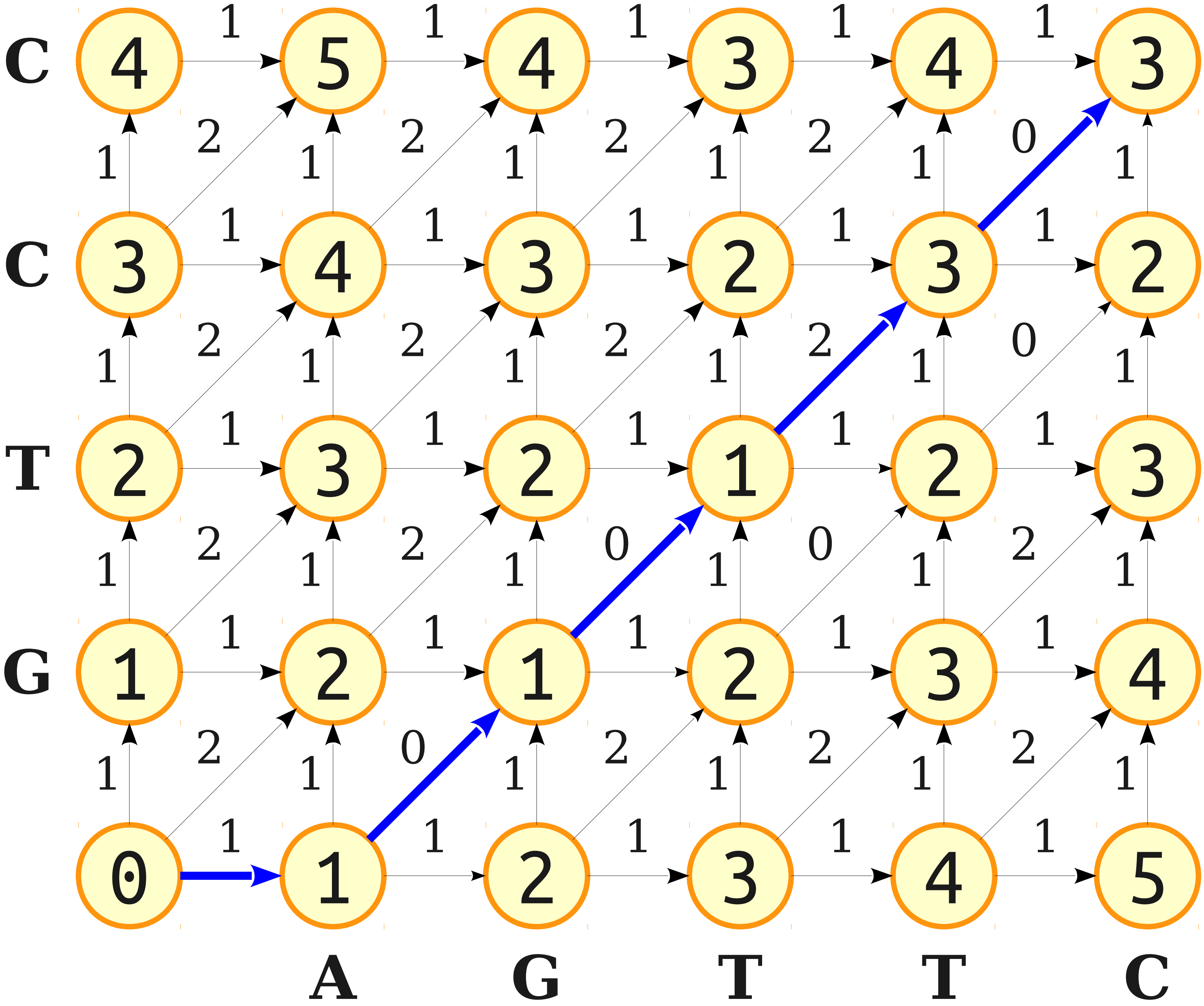

C 4 5 4 3 4 3
C 3 4 3 2 3 2
T 2 3 2 1 2 3
G 1 2 1 2 3 4
0 1 2 3 4 5
A G T T C

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


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

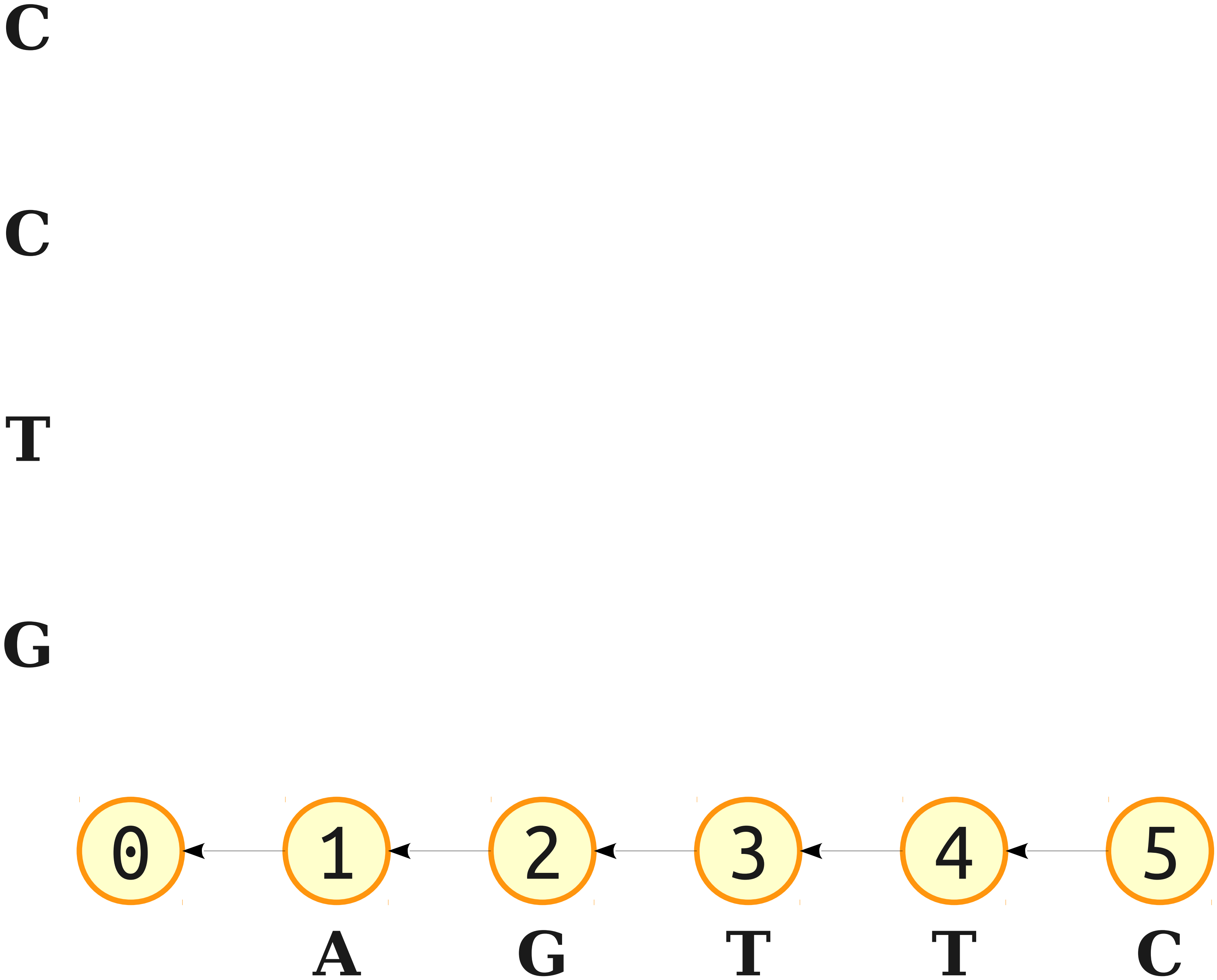

C
C
T
G
A
G
T
T
C

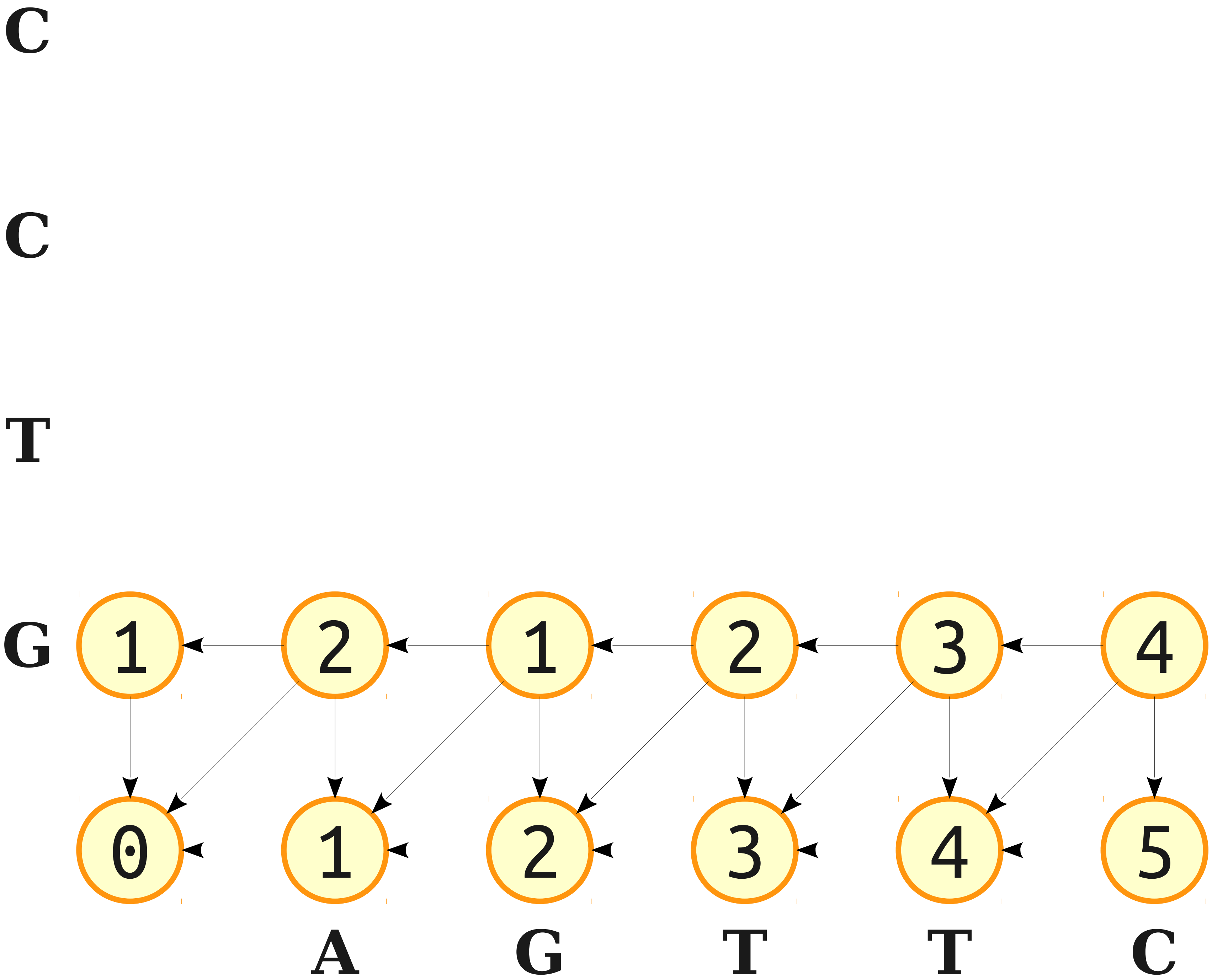

C
C
T
G
4 5 4 3 4 3
3 4 3 2 3 2
2 3 2 1 2 3
1 2 1 2 3 4
0 1 2 3 4 5
A
G
T
T
C

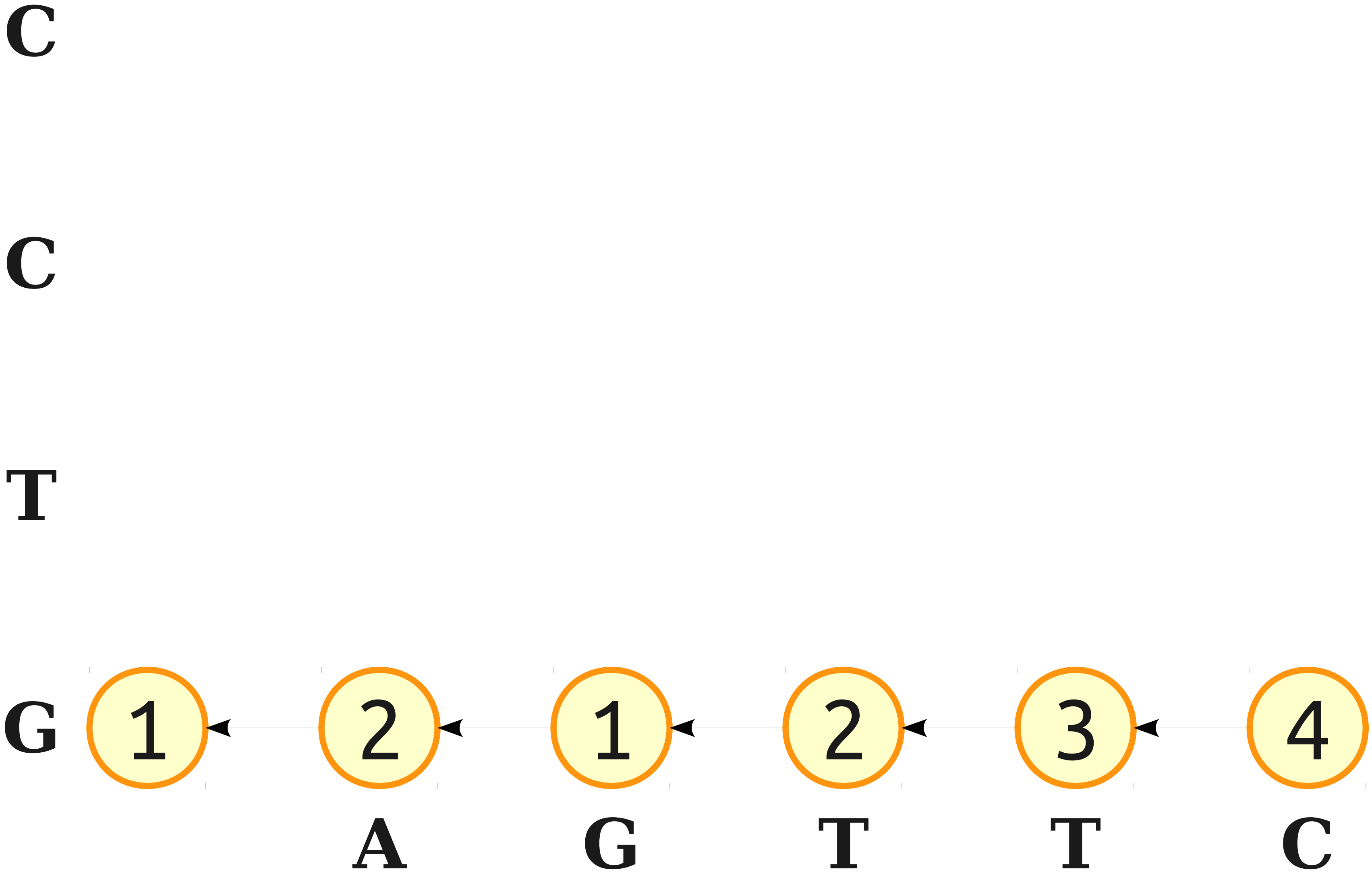

C
4 5 4 3 4 3
C
3 4 3 2 3 2
T
2 3 2 1 2 3
G
1 2 1 2 3 4
0 1 2 3 4 5
A
G
T
T
C

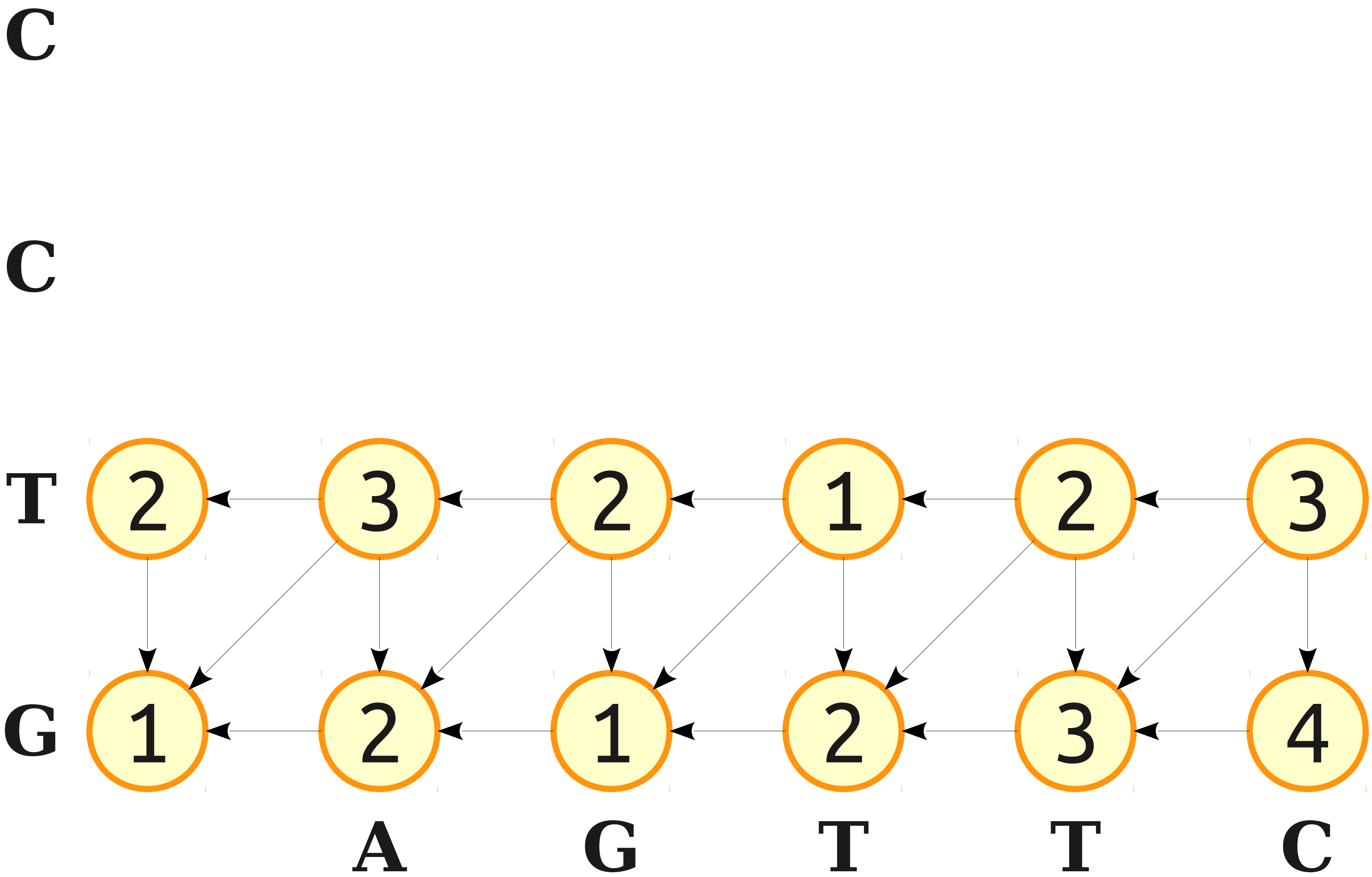
Finding the Alignment

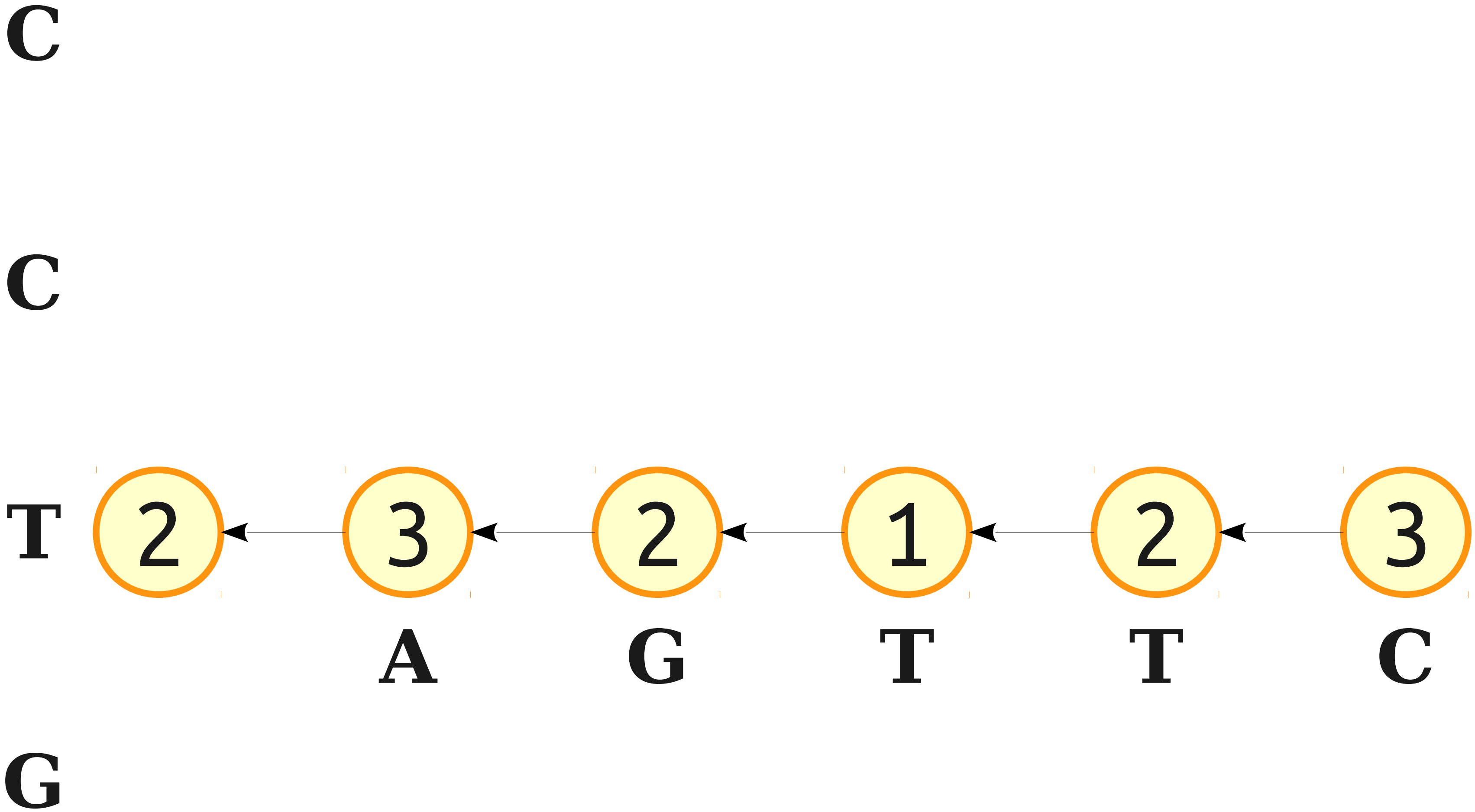
- As with the DP algorithms we saw last time, we can recover the optimal sequence alignment by running the recurrence in reverse.
- Option 1: Start in the upper-left corner and walk backwards through the grid, at each point choosing a successor such that the total cost matches.
- Option 2: Treat the problem as finding the shortest path from the lower-left corner to the upper-right corner.

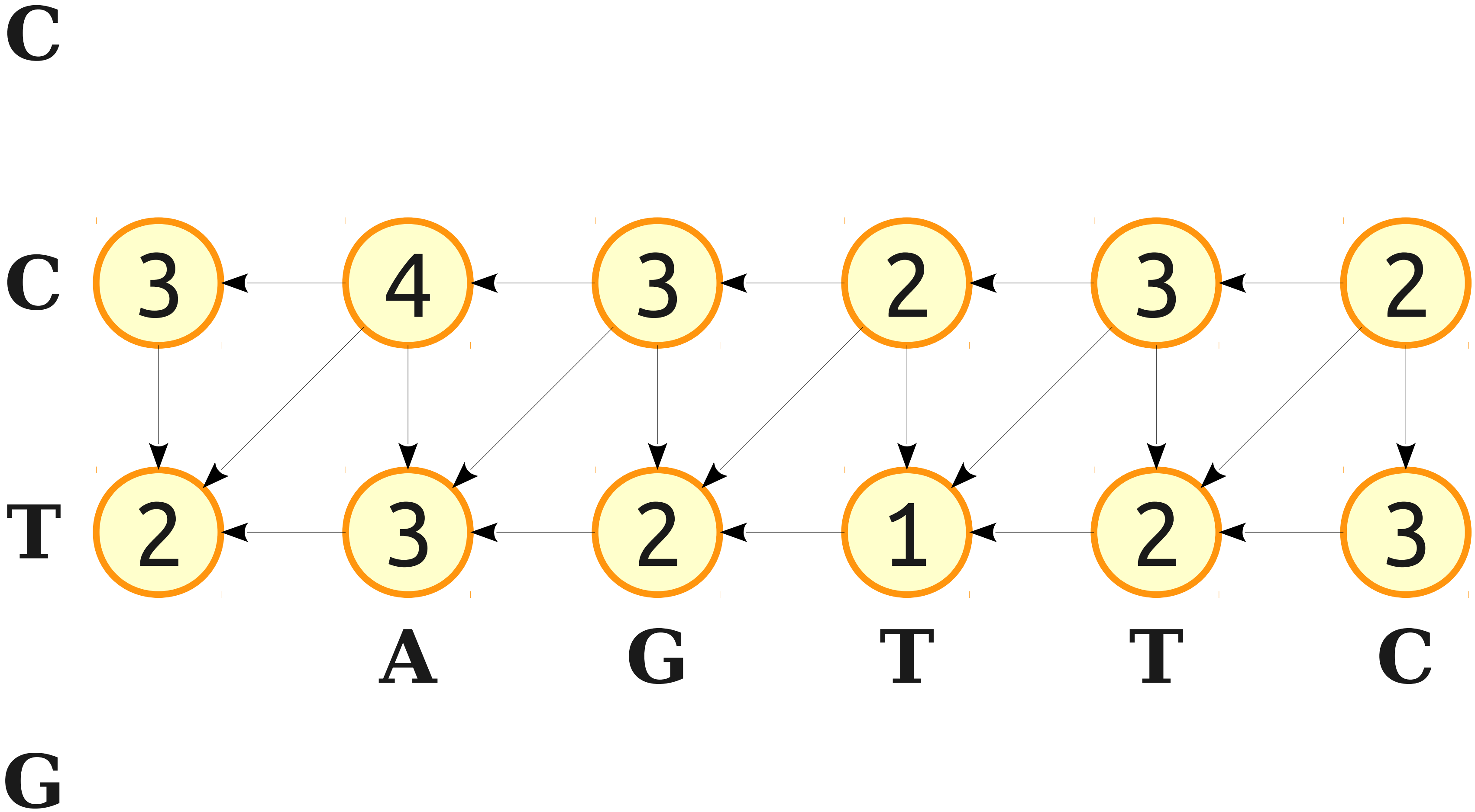

C
C
T
G
0
1
2
3
4
5
A
G
T
T
C

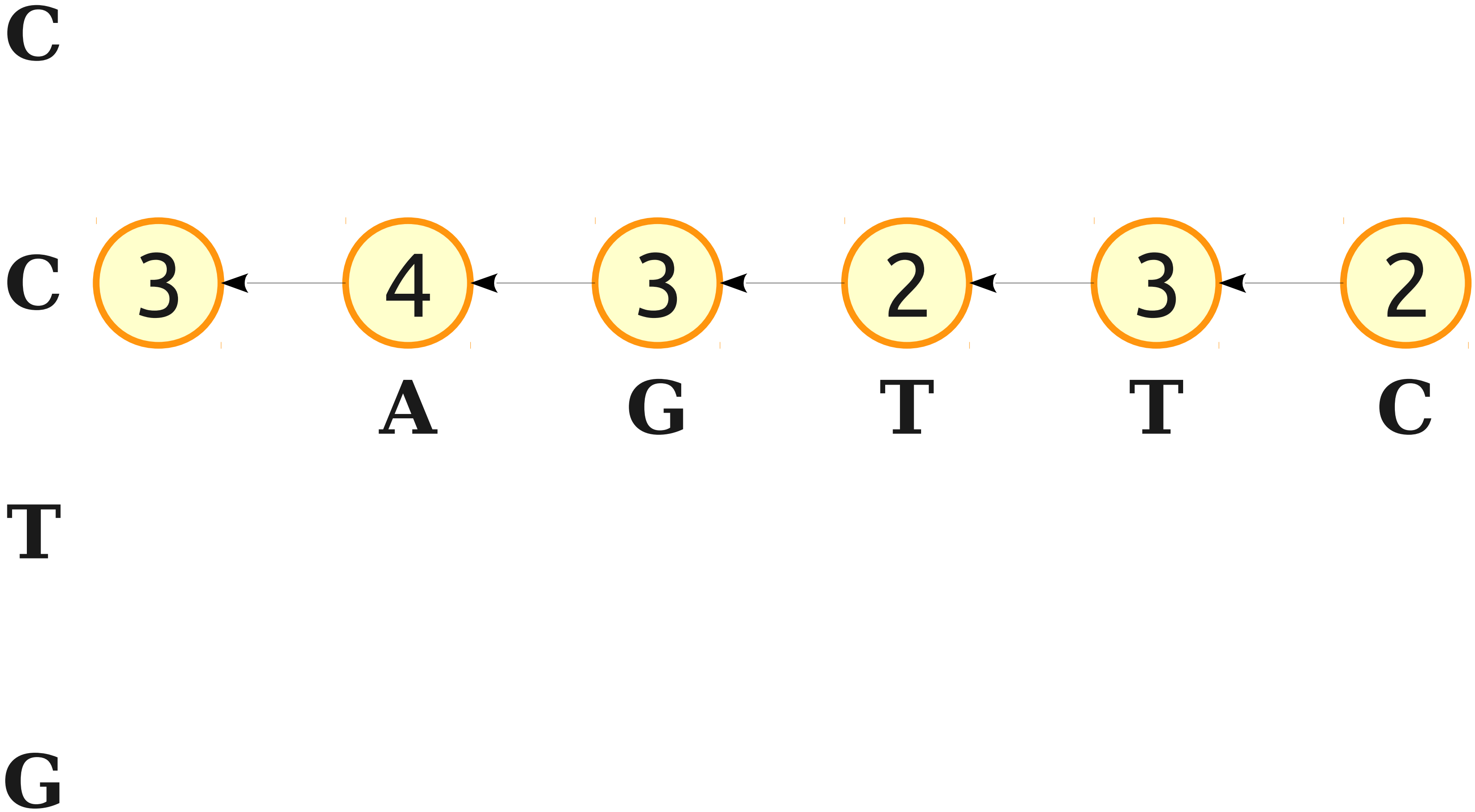

C
C
T
G
1
2
1
2
3
4
0
1
2
3
4
5
A
G
T
T
C

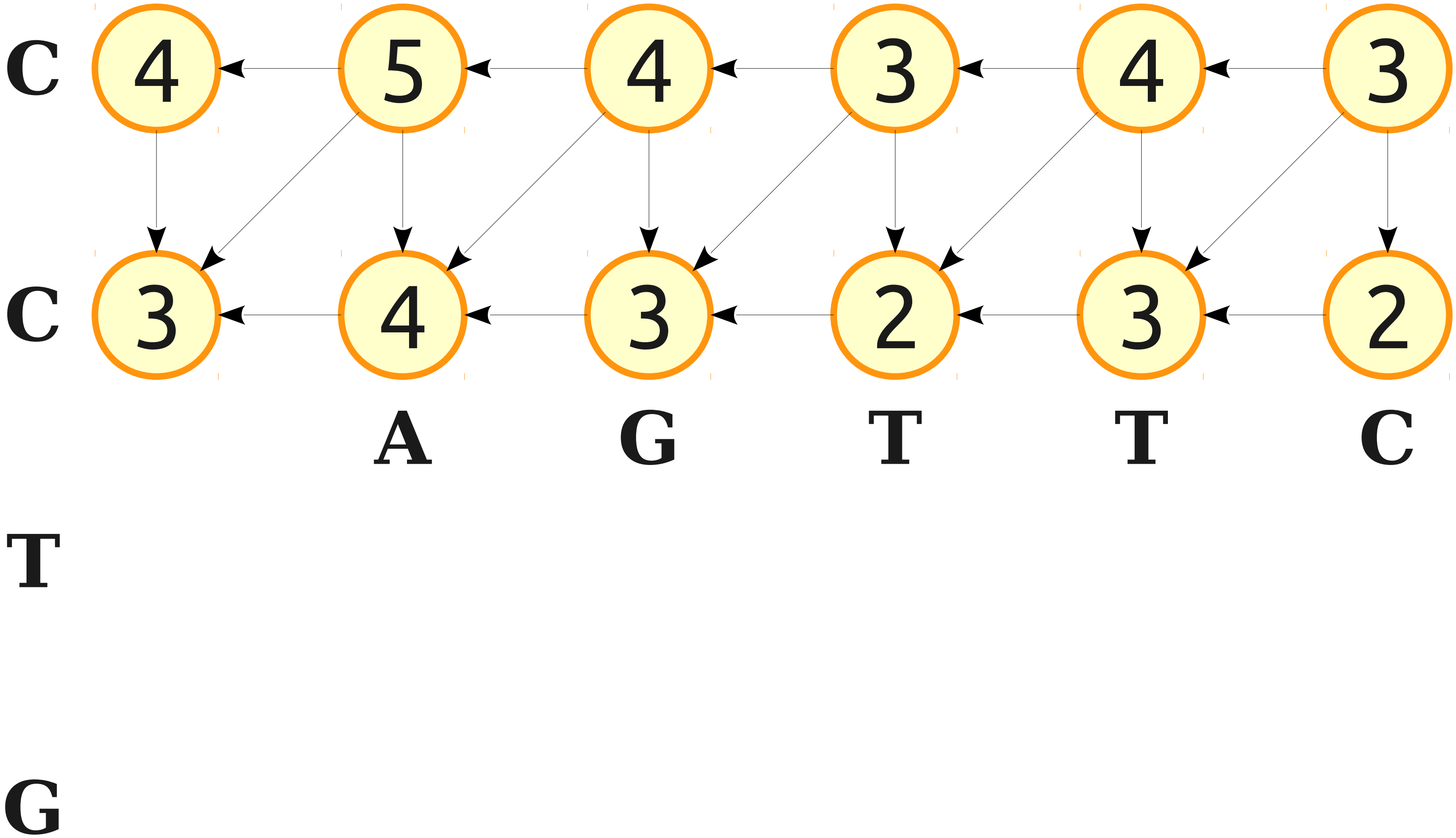

C
C
T
G
1
2
1
2
3
4
A
G
T
T
C

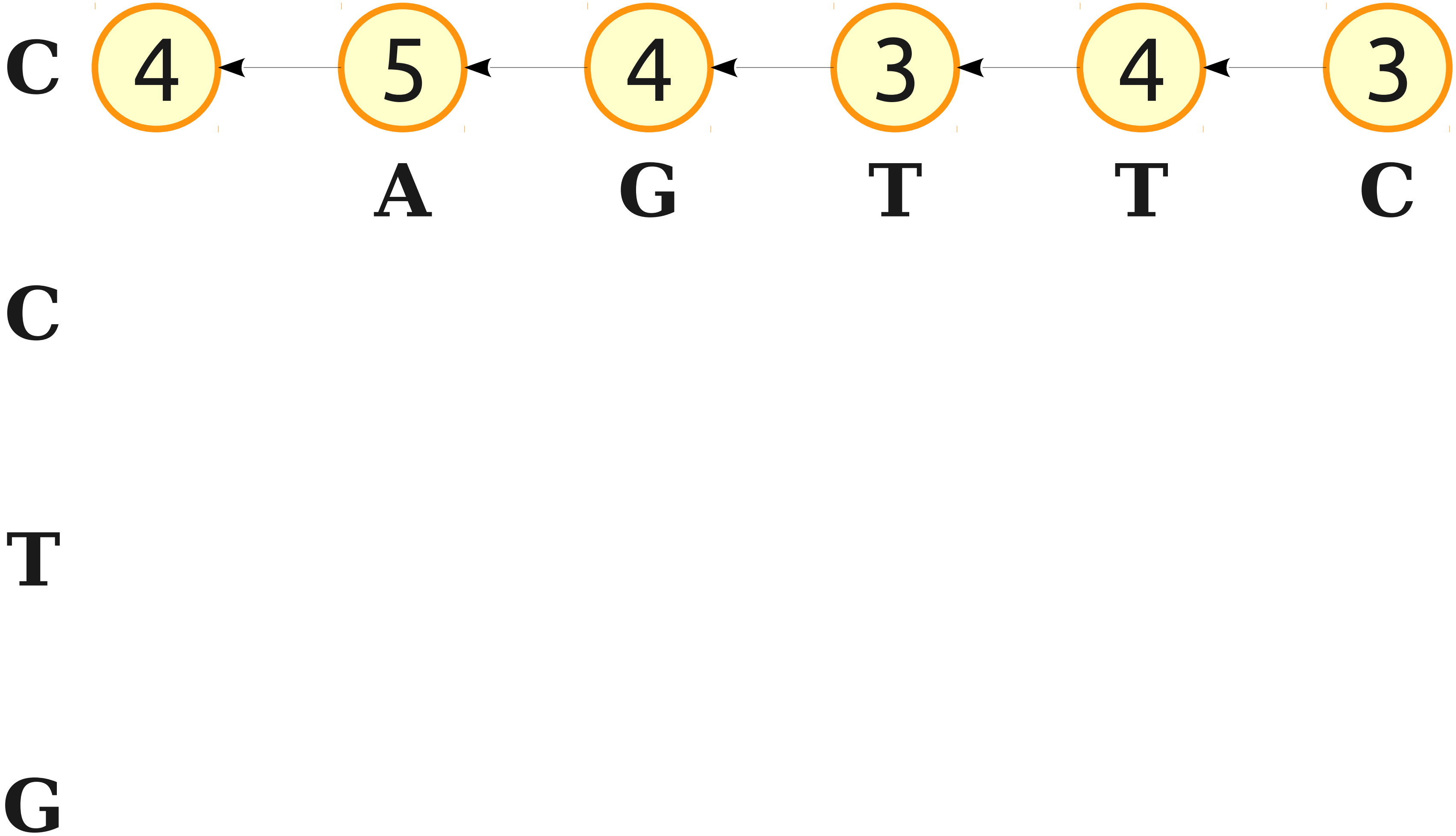

C
C
T
2
3
2
1
2
3
G
1
2
1
2
3
4
A
G
T
T
C

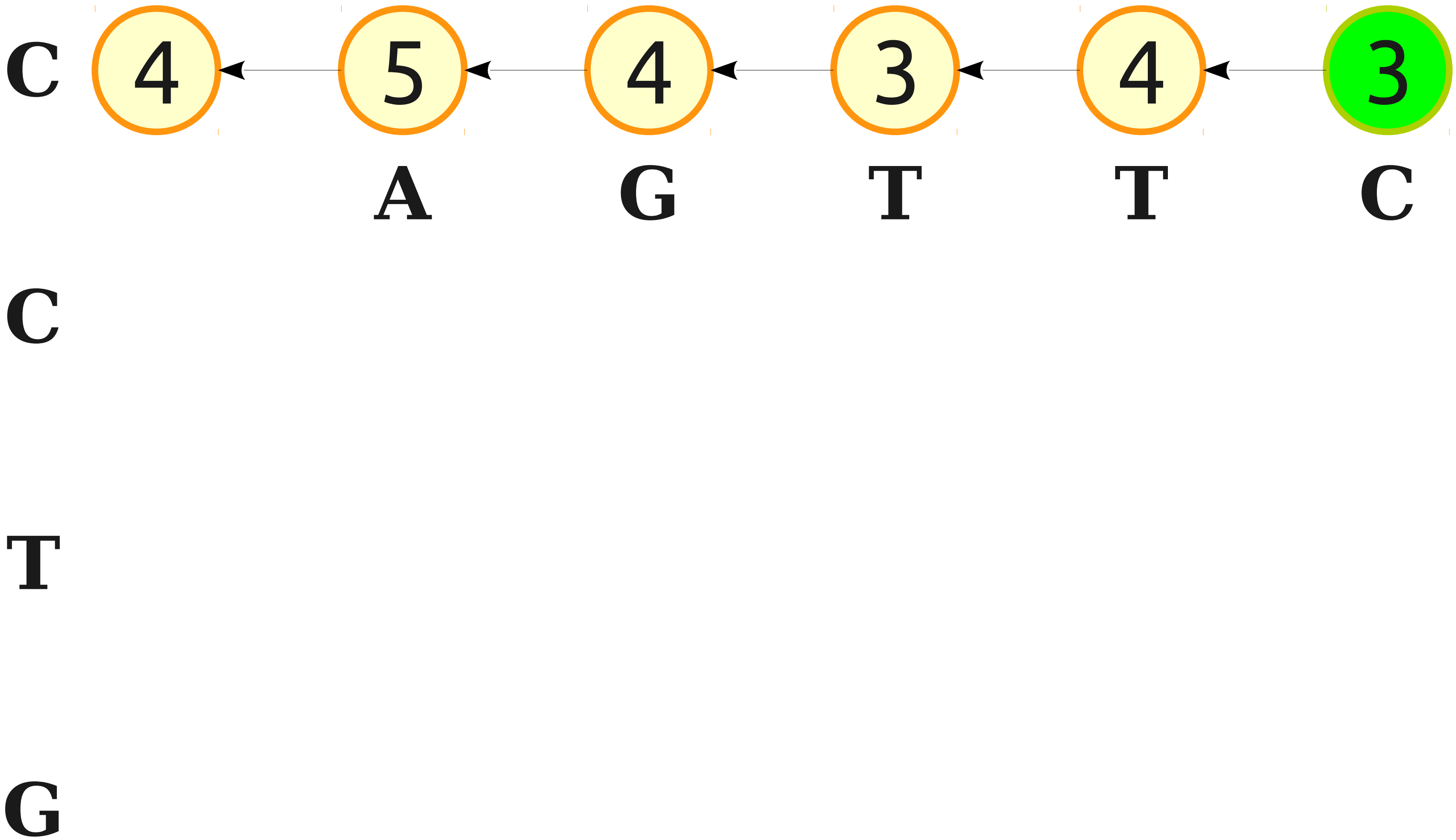

C
C
T
G
2
3
2
1
2
3
A
G
T
T
C


C
C
T
G
3
4
3
2
3
2
2
3
2
1
2
3
A
G
T
T
C


C
C
T
G
3
4
3
2
3
2
A
G
T
T
C


C
4
5
4
3
4
3
C
3
4
3
2
3
2
A
G
T
T
C
T
G


C
4
5
4
3
4
3
A
G
T
T
C
C
T
G


A G T T C
C C T G
4 5 4 3 4 3

Reducing Space

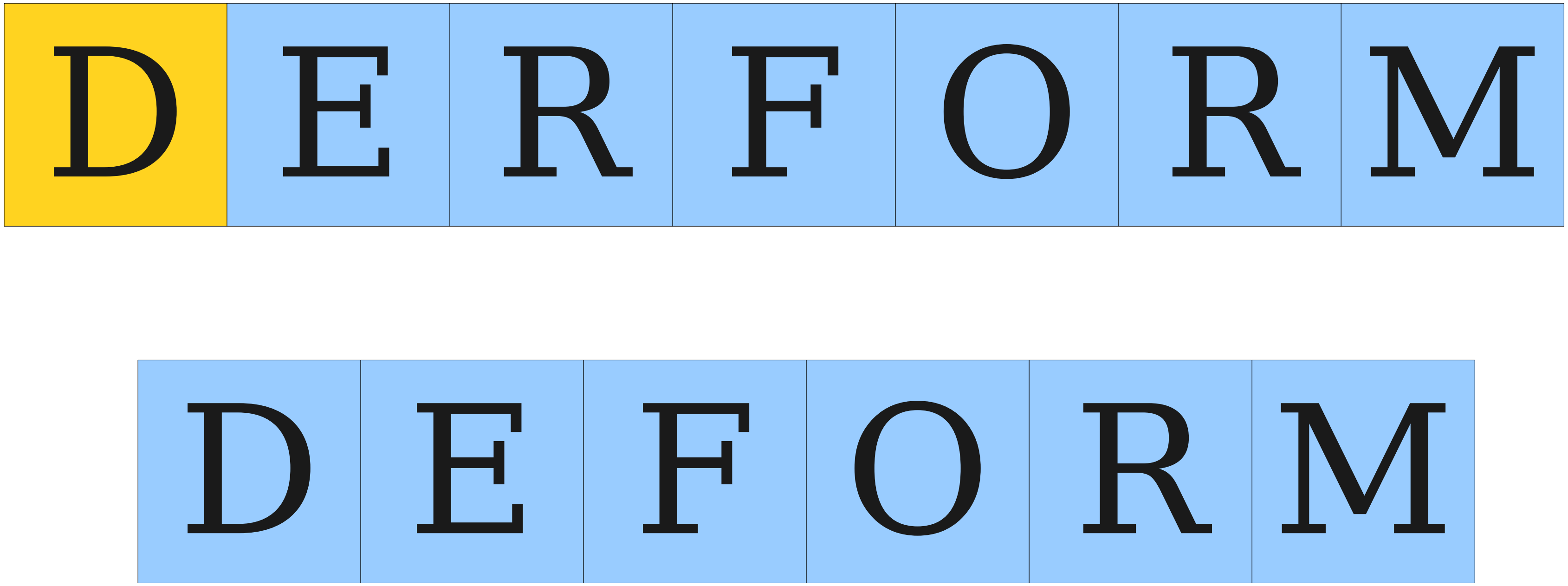
- If you only care about the *value* of the optimal solution and not the actual solution, you can compress the DP table by only storing the last row.
- Runtime now O(mn) with space O(min{m, n}), which is better than before.
- **Clever Trick:** See Kleinberg and Tardos section 6.7 for a way to get an O(mn)-time, O($m + n$)-space algorithm that does recover the optimal solution.

A Quick History Lesson

Another Algorithm: **Levenshtein Distance**

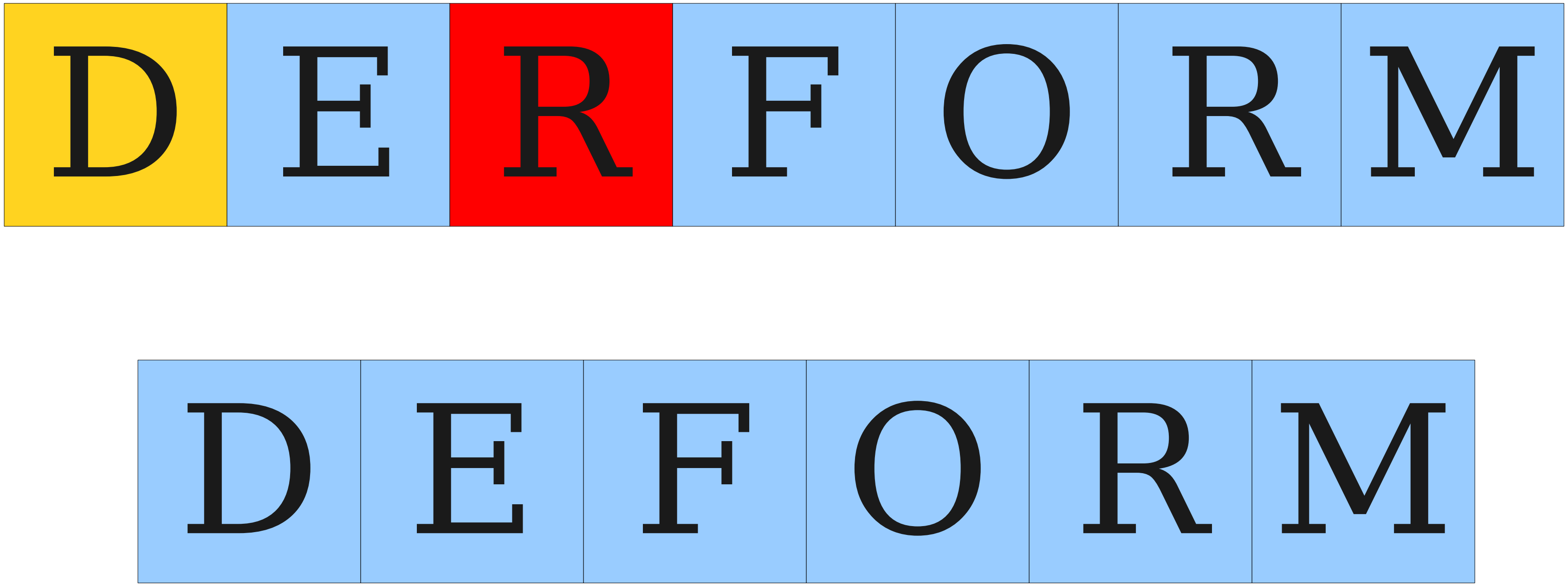
PERFORM

DEFORM



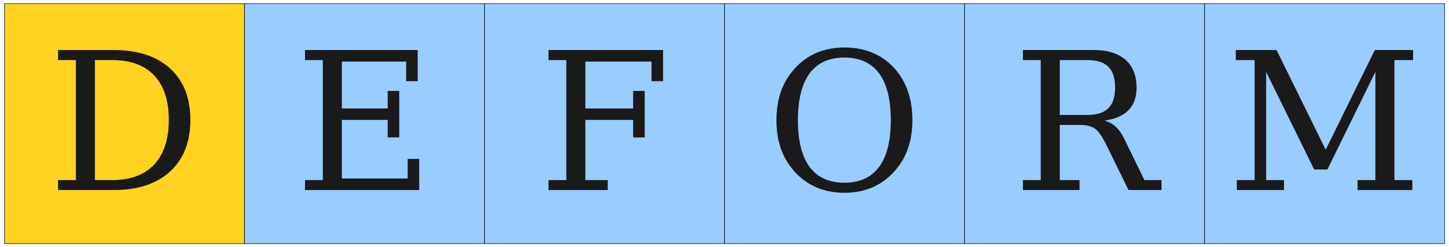
D E R F O R M

D E F O R M

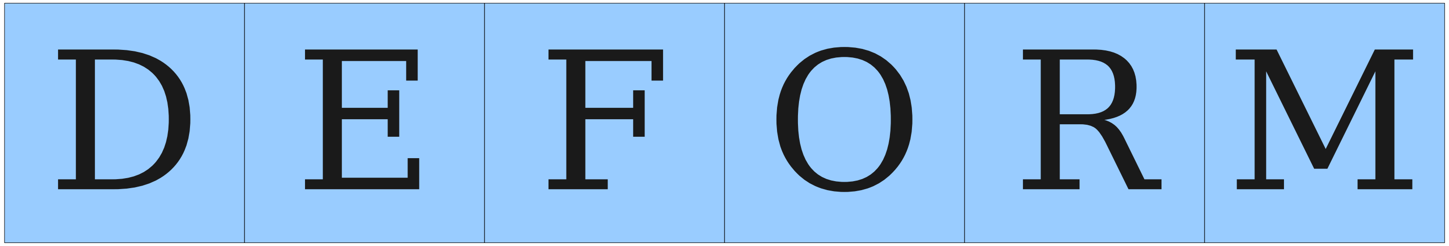


DERFORM

DEFORM



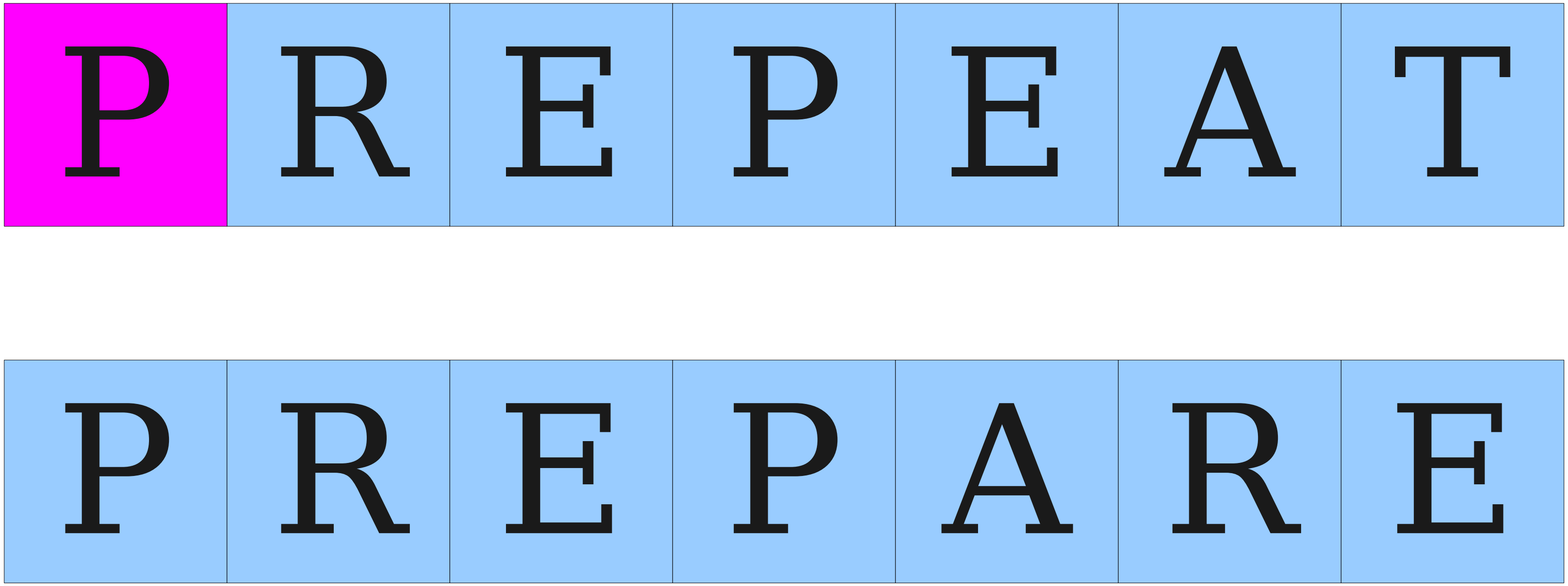
D	E	F	O	R	M



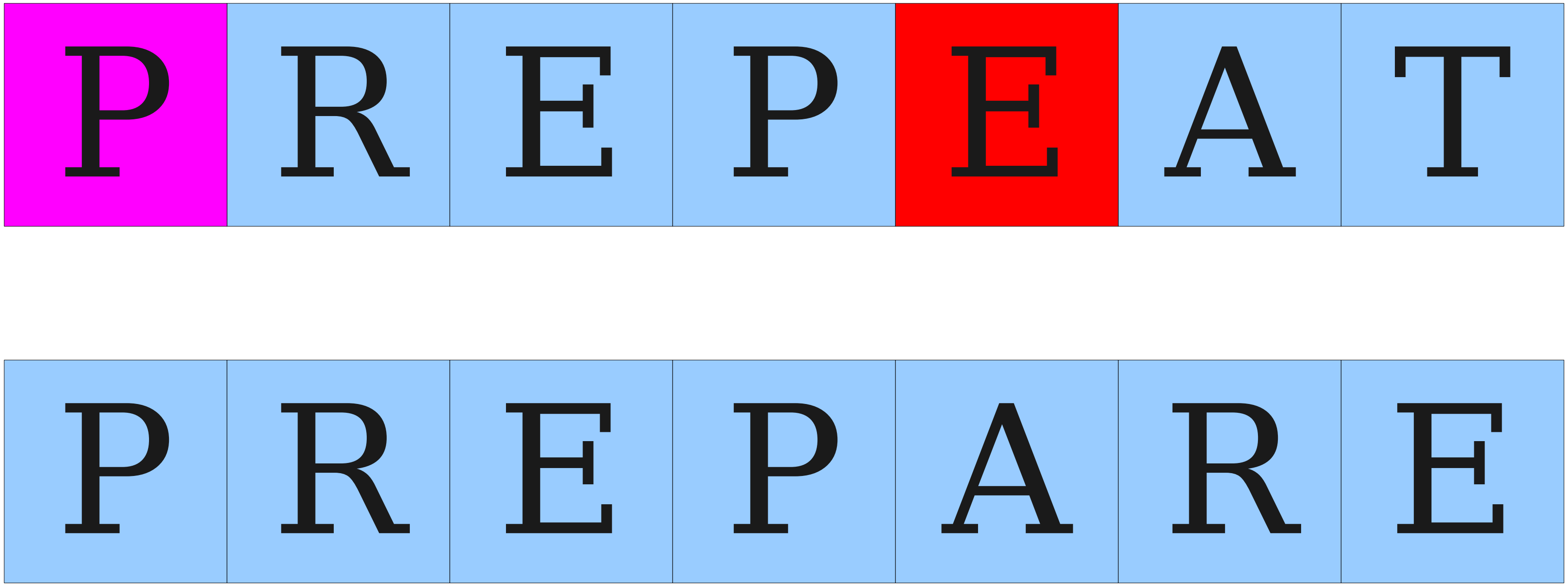
D	E	F	O	R	M

REPEAT

PREPARE



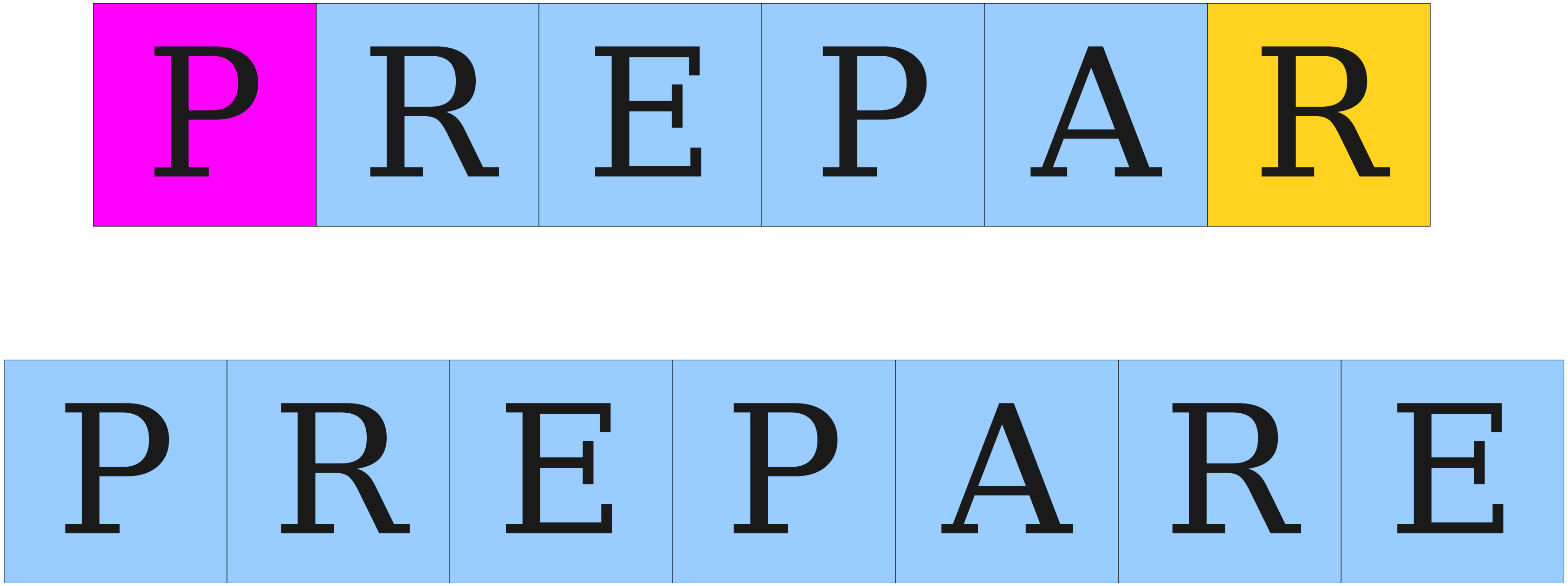
P	R	E	P	E	A	T
P	R	E	P	A	R	E



P	R	E	P	E	A	T
P	R	E	P	A	R	E

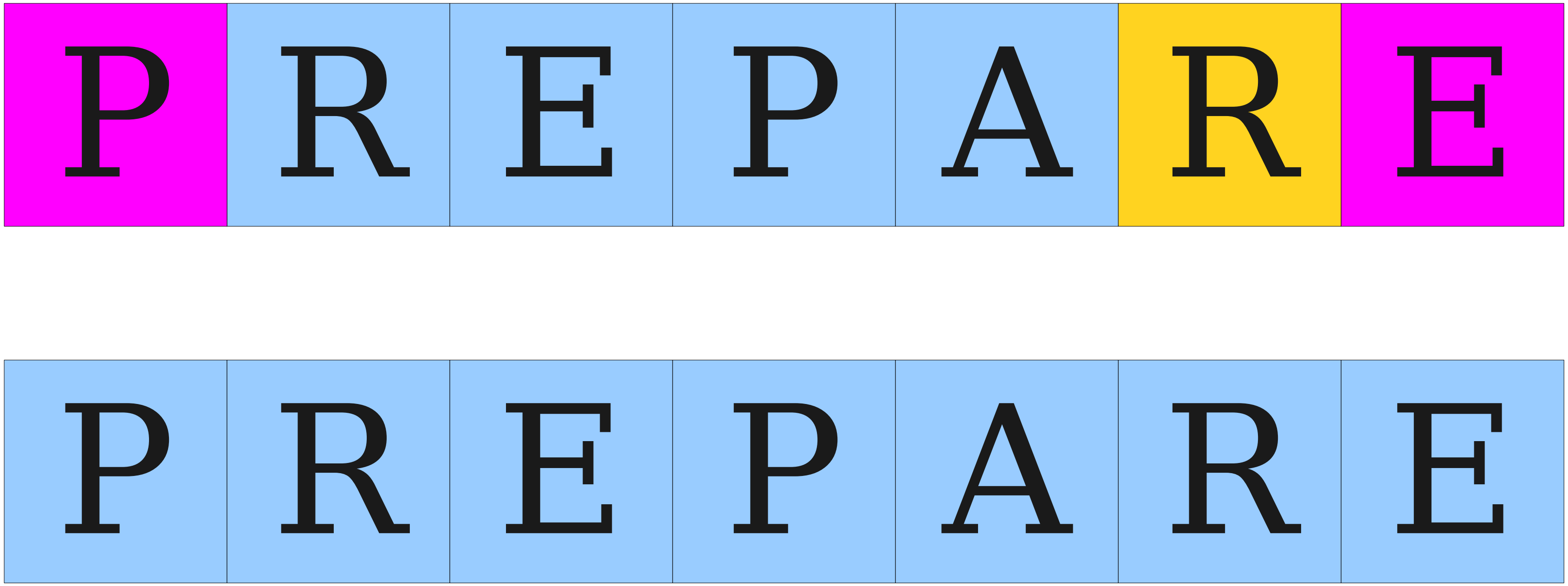
P	R	E	P	A	T

P	R	E	P	A	R	E



P	R	E	P	A	R

P	R	E	P	A	R	E



P	R	E	P	A	R	E

P	R	E	P	A	R	E

Transforming Strings

- Given a source string and target string, transform the source string into the target string by applying these edits:
  - **Insertion** of a new character,
  - **Deletion** of an existing character, or
  - **Replacement** of an existing character.
- The minimum number of edits required is called the **Levenshtein distance**.

Thinking Recursively

R	E	P	E	A	T

P	R	E	P	A	R	E

Thinking Recursively

R	E	P	E	A	T	
P	R	E	P	A	R	E

Thinking Recursively

P	R	E	P	E	A	T
P	R	E	P	A	R	E

Thinking Recursively

R	E	P	E	A	T
R	E	P	A	R	E

Thinking Recursively

R	E	P	E	A	T

R	E	P	A	R	E

Thinking Recursively

E P E A T

E P A R E

Thinking Recursively

E P E A T

E P A R E

Thinking Recursively

PEAT

PARE

Thinking Recursively

P E A T

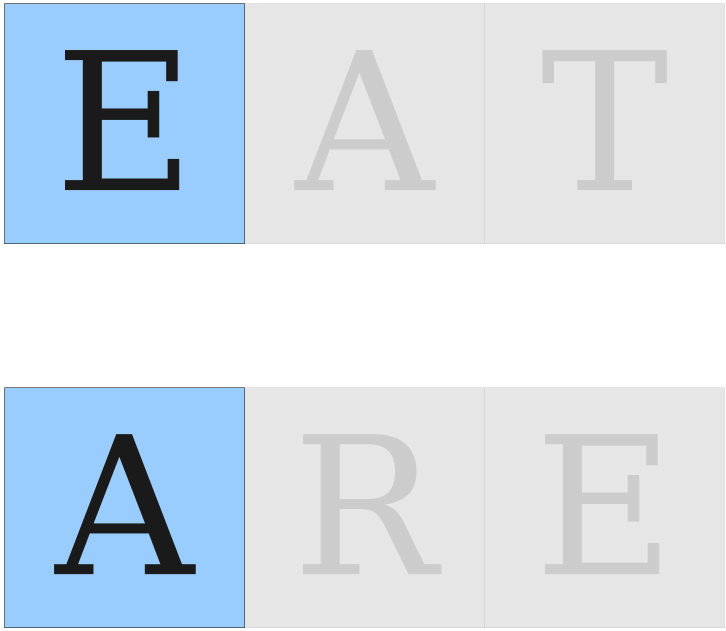
P A R E

Thinking Recursively

E A T

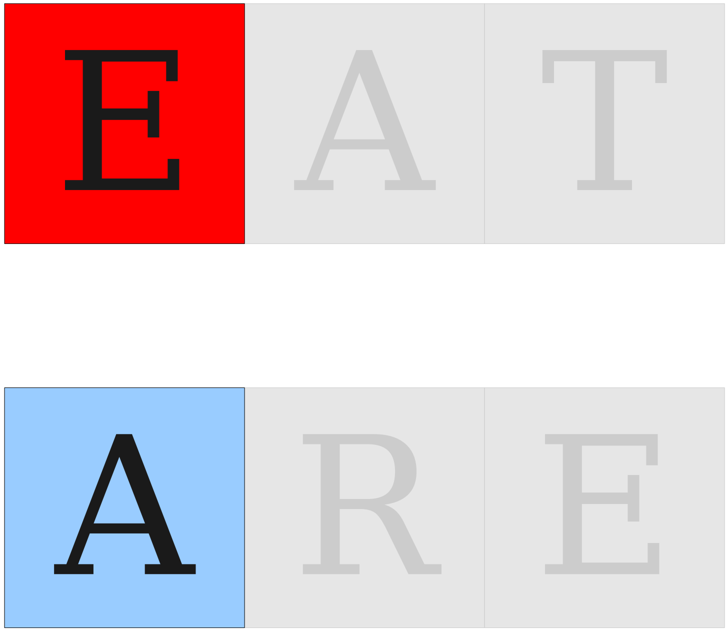
A R E

Thinking Recursively



E	A	T
A	R	E

Thinking Recursively



E A T

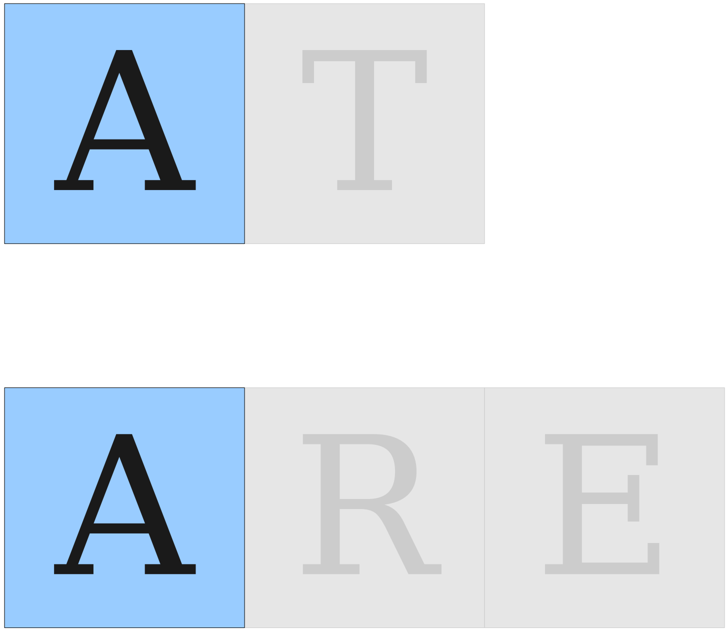
A R E

Thinking Recursively

A T

A R E

Thinking Recursively



A T

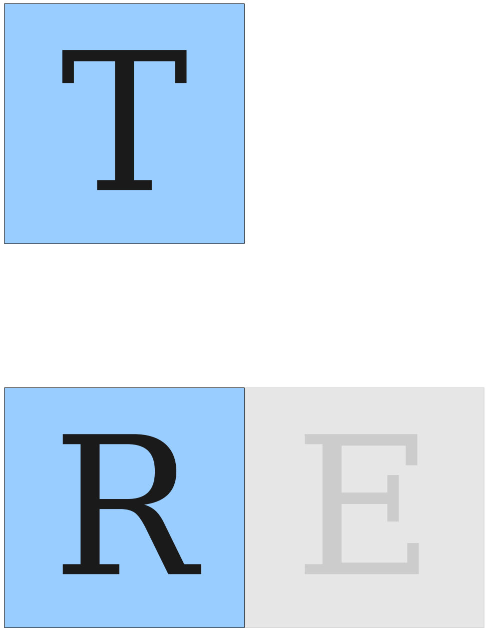
A R E

Thinking Recursively

T

R E

Thinking Recursively



T

R E

Thinking Recursively

R

R E

Thinking Recursively

Thinking Recursively

E

Thinking Recursively

E

We can either delete this character or add a matching E up top.

Our Options

- Look at the first characters of each string.
- We can either
  - Match them together, if they're the same character.
  - Add in a character to the top or bottom to match the other string's character.
  - Delete a character from the top or bottom.
  - Replace the top or bottom character to match the other character.
- When one string becomes empty, the options are to add the remaining characters or delete them from the other string, both of which have the same cost.

Some Notation

- Suppose we want to transform the first i characters of A into the first j characters of B.
- Let OPT(i, j) denote the optimal cost of such an alignment.
- Let I_{ij} be 0 if $A[i] = B[j]$ and 1 otherwise.
- **Claim:** OPT(i, j) satisfies the following:

$$\mathrm{OPT}(i,j)=\begin{cases} j & if\ i=0 \\ i & if\ j=0 \\ min\begin{Bmatrix} 1+OPT(i,j-1), \\ 1+OPT(i-1,j), \\ \mathrm{I}_{ij}+OPT(i-1,j-1) \end{Bmatrix} & otherwise \end{cases}$$

Seem Familiar?

$$\mathrm{OPT}(i,j)=\begin{cases} j\delta & if\ i=0 \\ i\delta & if\ j=0 \\ min\begin{Bmatrix} \delta+OPT(i-1,j), \\ \delta+OPT(i,j-1), \\ \alpha_{A[i]B[j]}+OPT(i-1,j-1) \end{Bmatrix} & otherwise \end{cases}$$

$$\mathrm{OPT}(i,j)=\begin{cases} j & if\ i=0 \\ i & if\ j=0 \\ min\begin{Bmatrix} 1+OPT(i,j-1), \\ 1+OPT(i-1,j), \\ \mathrm{I}_{ij}+OPT(i-1,j-1) \end{Bmatrix} & otherwise \end{cases}$$

A Clever Reduction

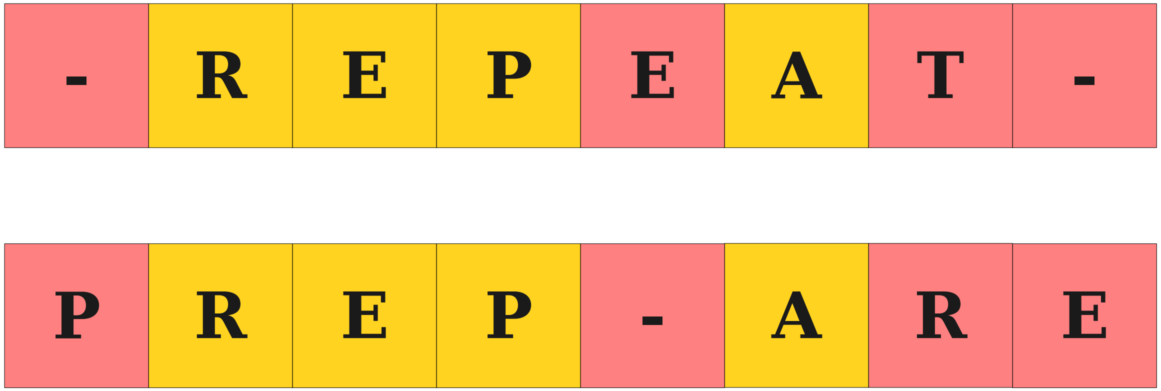
- ***Claim:*** The Levenshtein distance between two strings is equal to their alignment cost if we set
  - $\delta = 1$.
  - $\alpha_{ab} = 0$ if $a = b$ and is 1 otherwise.
- ***Proof Idea:*** First, prove that the previous recurrence holds for Levenshtein distance, then show the recurrence is identical to that of sequence alignment with the above parameterization.

The Intuition

R	**E**	**P**	**E**	**A**	**T**

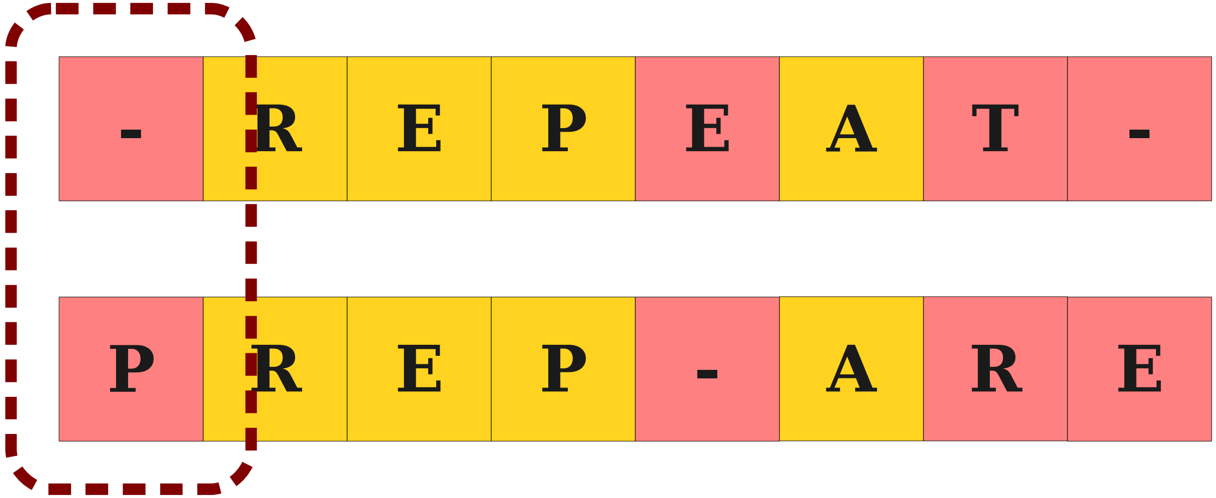
P	**R**	**E**	**P**	**A**	**R**	**E**

The Intuition



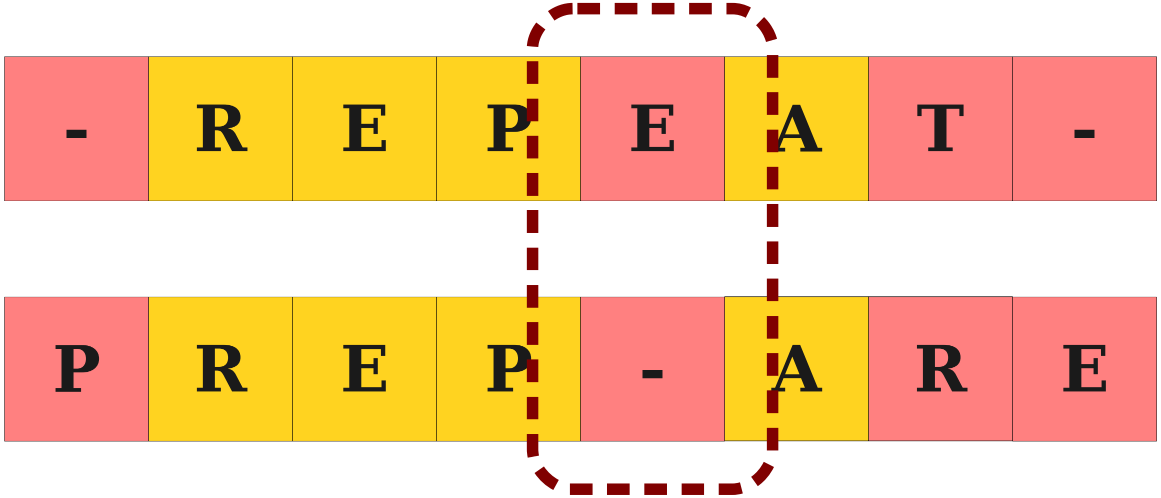
-	R	E	P	E	A	T	-
P	R	E	P	-	A	R	E

The Intuition



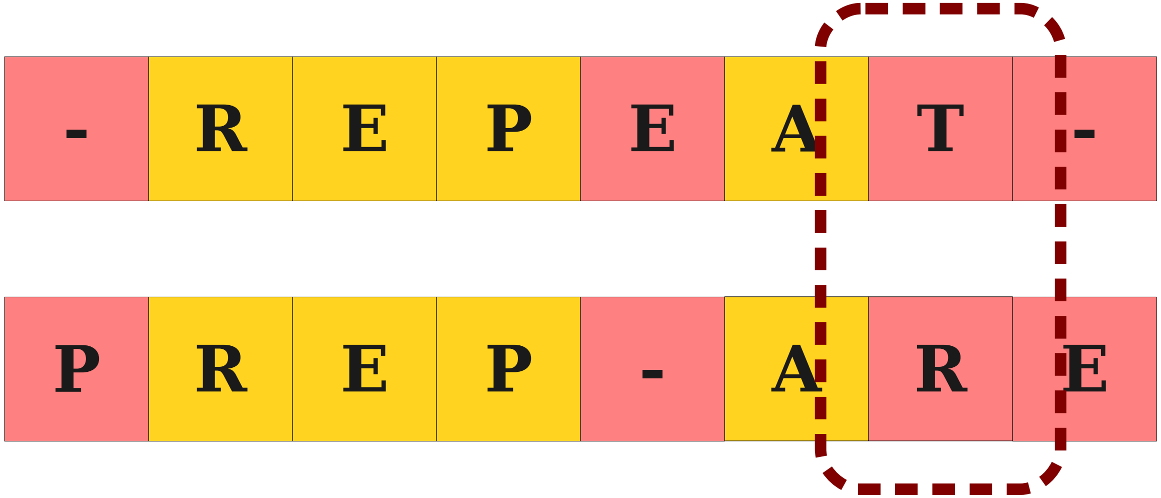
-	R	E	P	E	A	T	-
P	R	E	P	-	A	R	E

The Intuition



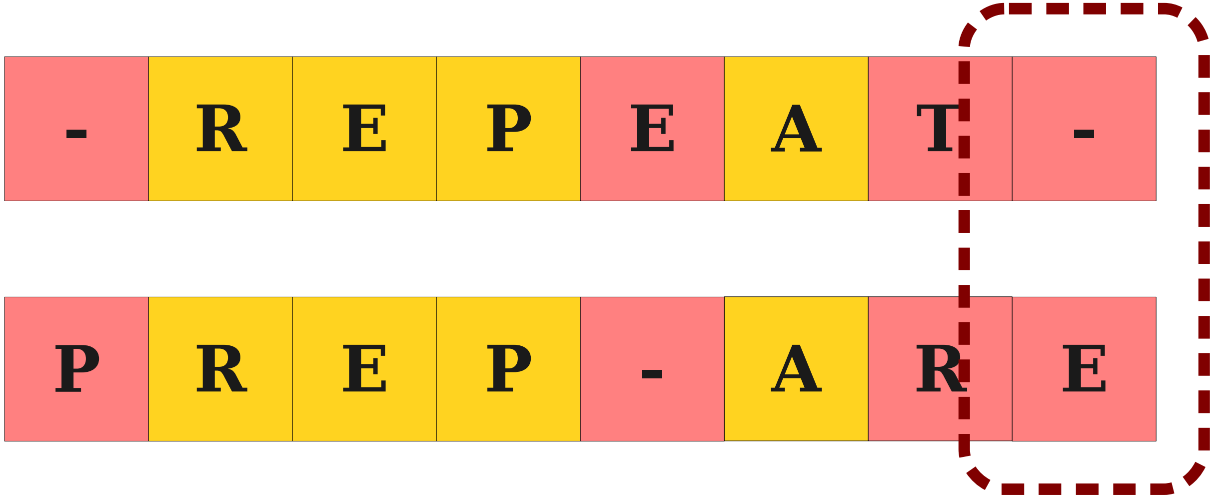
-	R	E	P	E	A	T	-
P	R	E	P	-	A	R	E

The Intuition



-	R	E	P	E	A	T	-
P	R	E	P	-	A	R	E

The Intuition



-	**R**	**E**	**P**	**E**	**A**	**T**	**-**
P	**R**	**E**	**P**	**-**	**A**	**R**	**E**

Another Intuition

- Run sequence alignment and do the following:
  - For any character matched against a blank, delete that character or insert a matching character into the other string.
  - For any character matched against a mismatched character, replace one character with the other.
- Therefore, can compute distance and transformation in $O(mn)$ time and $O(m + n)$ space, or can get value in $O(mn)$ time and $O(\min\{m, n\})$ space.

Next Time

- Shortest Paths Revisited
- The Bellman-Ford Algorithm
- Network Routing