Principles of Computer Science II

Dynamic Programming and Sequence Similarity

Marco Zecchini

Sapienza University of Rome

Lecture 9

Sequence Similarity

- We want to look for repeating patterns within DNA sequences (e.g., detecting duplicated genes, repeated subsequences).
- Now we want to compare *different* sequences:
  - Are they similar? How many positions differ? Can we *align* them to highlight similarities and differences?

Similarity of ATATATAT vs TATATATA

```
A T A T A T A T
: : : : : : :
T A T A T A T A
```

Sequence Similarity

- We want to look for repeating patterns within DNA sequences (e.g., detecting duplicated genes, repeated subsequences).
- Now we want to compare *different* sequences:
  - Are they similar? How many positions differ? Can we *align* them to highlight similarities and differences?

Similarity of ATATATAT vs TATATATA

```
A T A T A T A T
: : : : : : :
T A T A T A T A
```

- At first glance the sequences seem very similar, but not identical.
- We need a formal way to **measure their similarity**.

Sequence Similarity

- We want to look for repeating patterns within DNA sequences (e.g., detecting duplicated genes, repeated subsequences).
- Now we want to compare *different* sequences:
  - Are they similar? How many positions differ? Can we *align* them to highlight similarities and differences?

Similarity of ATATATAT vs TATATATA

```
A T A T A T A T
: : : : : : :
T A T A T A T A
```

- At first glance the sequences seem very similar, but not identical.
- We need a formal way to **measure their similarity**.
- This leads us to the concept of an **alignment** and to **dynamic programming** algorithms that compute it efficiently.

Coin Change Problem

United States Change Problem:
Convert some amount of money into the fewest number of coins.

Input: An amount of money, M, in cents.

Output: The smallest number of quarters q, dimes d, nickels n, and pennies p whose values add to M (i.e., $25q + 10d + 5n + p = M$ and $q + d + n + p$ is as small as possible).

- We know that the greedy solution is not good..
- ... we are only left with a brute-force approach that is impractical

Finding the best combination

Suppose you need to make change for 77 cents and the only coin denominations available are 1, 3, and 7 cents.
The best combination for 77 cents will be one of the following:

- the best combination for $77 - 1 = 76$ cents, plus a 1-cent coin;

Finding the best combination

Suppose you need to make change for 77 cents and the only coin denominations available are 1, 3, and 7 cents.
The best combination for 77 cents will be one of the following:

- the best combination for $77 - 1 = 76$ cents, plus a 1-cent coin;
- the best combination for $77 - 3 = 74$ cents, plus a 3-cent coin;

Finding the best combination

Suppose you need to make change for 77 cents and the only coin denominations available are 1, 3, and 7 cents.
The best combination for 77 cents will be one of the following:

- the best combination for $77 - 1 = 76$ cents, plus a 1-cent coin;
- the best combination for $77 - 3 = 74$ cents, plus a 3-cent coin;
- the best combination for $77 - 7 = 70$ cents, plus a 7-cent coin.

Finding the best combination

Suppose you need to make change for 77 cents and the only coin denominations available are 1, 3, and 7 cents.
The best combination for 77 cents will be one of the following:

- the best combination for $77 - 1 = 76$ cents, plus a 1-cent coin;
- the best combination for $77 - 3 = 74$ cents, plus a 3-cent coin;
- the best combination for $77 - 7 = 70$ cents, plus a 7-cent coin.

$$bestNumCoins_M = \min \begin{cases} bestNumCoins_{M-1} + 1 \\ bestNumCoins_{M-3} + 1 \\ bestNumCoins_{M-7} + 1 \end{cases}$$

Same for 76 and, then, 75...

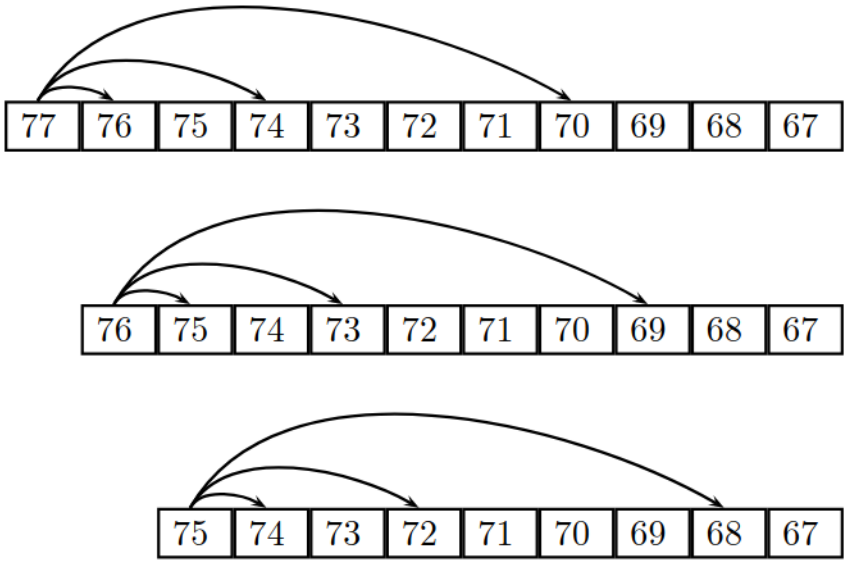


Figure 6.1 The relationships between optimal solutions in the Change problem. The smallest number of coins for 77 cents depends on the smallest number of coins for 76, 74, and 70 cents; the smallest number of coins for 76 cents depends on the smallest number of coins for 75, 73, and 69 cents, and so on.

RecursiveChange

```
RECURSIVECHANGE(M, c, d)
1  if M = 0
2      return 0
3  bestNumCoins ← ∞
4  for i ← 1 to d
5      if M ≥ c_i
6          numCoins ← RECURSIVECHANGE(M − c_i, c, d)
7          if numCoins + 1 < bestNumCoins
8              bestNumCoins ← numCoins + 1
9  return bestNumCoins
```

RecursiveChange

```
RECURSIVECHANGE(M, c, d)
1  if M = 0
2      return 0
3  bestNumCoins ← ∞
4  for i ← 1 to d
5      if M ≥ c_i
6          numCoins ← RECURSIVECHANGE(M − c_i, c, d)
7          if numCoins + 1 < bestNumCoins
8              bestNumCoins ← numCoins + 1
9  return bestNumCoins
```

- However, it computes the optimum amount of coin repeatedly

RecursiveChange

```
RECURSIVECHANGE(M, c, d)
1  if M = 0
2      return 0
3  bestNumCoins ← ∞
4  for i ← 1 to d
5      if M ≥ c_i
6          numCoins ← RECURSIVECHANGE(M − c_i, c, d)
7          if numCoins + 1 < bestNumCoins
8              bestNumCoins ← numCoins + 1
9  return bestNumCoins
```

- However, it computes the optimum amount of coin repeatedly
- The optimal coin combination for 70 cents is recomputed repeatedly nine times over and over as $(77-7), (77-3-3-1), (77-3-1-3), (77-1-3-3), (77-3-1-1-1-1), (77-1-3-1-1-1), (77-1-1-3-1-1), (77-1-1-1-3-1)$, $(77-1-1-1-1-3)$ and $(77-1-1-1-1-1-1-1)$.

Reverse the order (look at line 2)...

Leverage previously computed solutions to form solutions to larger problems and avoid all this recomputation

```
DPCHANGE(M, c, d)
1  bestNumCoins_0 ← 0
2  for m ← 1 to M
3      bestNumCoins_m ← ∞
4      for i ← 1 to d
5          if m ≥ c_i
6              if bestNumCoins_{m−c_i} + 1 < bestNumCoins_m
7                  bestNumCoins_m ← bestNumCoins_{m−c_i} + 1
8  return bestNumCoins_M
```

We compute a solution for each possible amount of money m from 1 to M and, since it takes d steps, to find the right coin the cost is $\mathcal{O}(Md)$.

Dynamic Programming

- Dynamic programming solves problems by combining the solutions to subproblems.
- "Programming" refers to a tabular method, not to writing computer code.
- Dynamic programming applies when the subproblems overlaps - that is, when subproblems share subsubproblems
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem

Sequence Similarity

- We looked for repeating patterns within DNA sequences.
- How can we measure the similarity between different sequences?

Similarity of ATATATAT vs TATATATA

A	T	A	T	A	T	A	T
:	:	:	:	:	:	:	
T	A	T	A	T	A	T	A

Hamming distance

31 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

This article includes a list of general references, but **it lacks sufficient corresponding inline citations**. Please help to improve this article by introducing more precise citations. *(May 2015)* *(Learn how and when to remove this message)*

In information theory, the **Hamming distance** between two strings or vectors of equal length is the number of positions at which the corresponding symbols are different. In other words, it measures the minimum number of *substitutions*

Hamming distance

same length!

Sequence Similarity

- We looked for repeating patterns within DNA sequences.
- How can we measure the similarity between different sequences?

Similarity of ATATATAT vs TATATATA

```
A T A T A T A T
: : : : : : :
T A T A T A T A
```

Alignment of ATATATAT vs TATATATA

```
A T A T A T A T -
  : : : : : : :
- T A T A T A T A
```

(An alignment places the two sequences one above the other, possibly inserting gaps "–", to show how their symbols correspond.)

Edit Distance

- We use the notion of Vladimir Levenshtein introduced in 1966
- Edit distance – the **minimum** number of editing operations needed to transform one string into another (insert/delete symbol or substitute one symbol for another).

Alignment of ATATATAT vs TATAAT

```
A T A T A T A T
  : : : : : : :
- T A T A - A T
```

Edit Distance

Alignment of TGCATAT vs ATCCGAT

TGCATAT
↓ delete last T
TGCATA
↓ delete last A
TGCAT
↓ insert A at the front
ATGCAT
↓ substitute C for G in the third position
ATCCAT
↓ insert a G before the last A
ATCCGAT

Five operations.

Edit Distance

Alignment of TGCATAT vs ATCCGAT

TGCATAT
↓ insert A at the front
ATGCATAT
↓ delete T in the sixth position
ATGCAAT
↓ substitute G for A in the fifth position
ATGCGAT
↓ substitute C for G in the third position
ATCCGAT

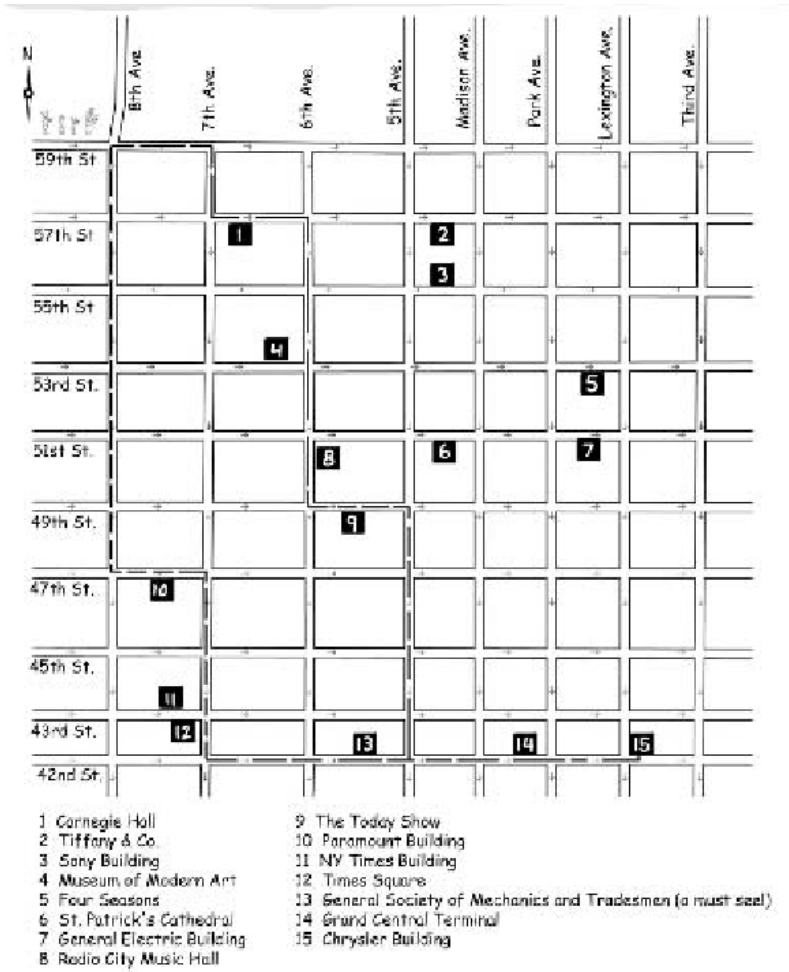
Four operations.

Edit Distance

- Vladimir Levenshtein defined the notion of Edit distance
- Did not provide an algorithm to compute it.

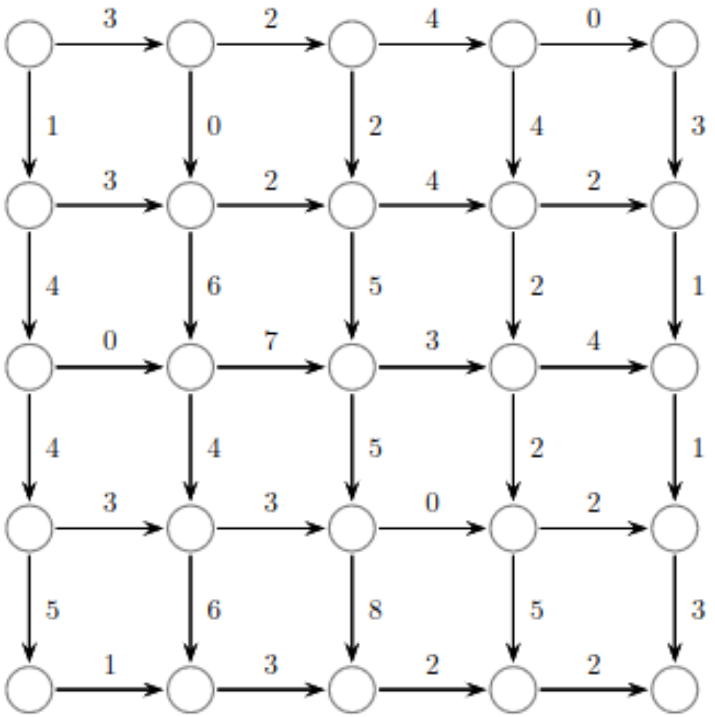
Manhattan Tourist Problem

- Sightseeing tour in Manhattan where a group of tourists wants to walk from the corner of 59th Street and 8th Avenue to the Chrysler Building.
- Many attractions along the way and the tourists want to see as many attractions as possible.
- The tourists can move either to the south or to the east, but even so, they can choose from many different paths.



Manhattan Tourist Problem as a graph

- We can represent this gridlike structure as a *graph*
- Intersections are *verteces*
- Streets are *edges* that are oriented either towards south ($\downarrow$) or east ($\rightarrow$) and have a *weight*
- A *path* is a continuous sequence of edges, and the *length of a path* is the sum of the edge weights in the path



Problem statement

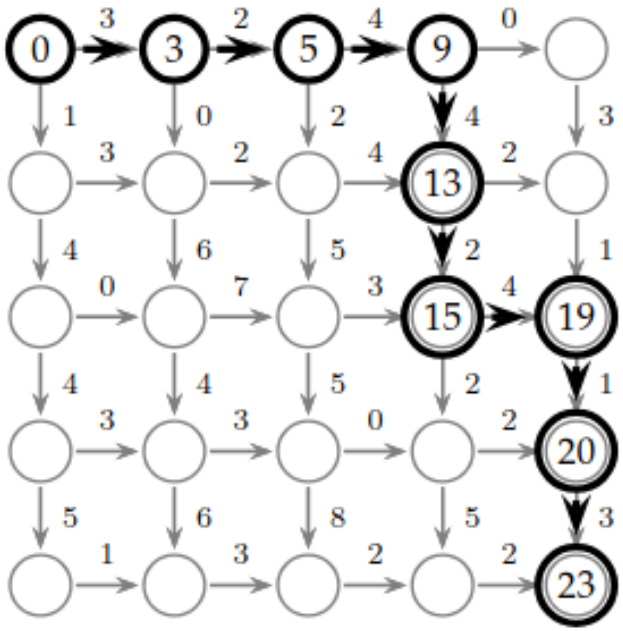
Manhattan Tourist Problem:
Find a longest path in a weighted grid.

Input: A weighted grid G with two distinguished vertices: a *source* and a *sink*.

Output: A longest path in G from *source* to *sink*.

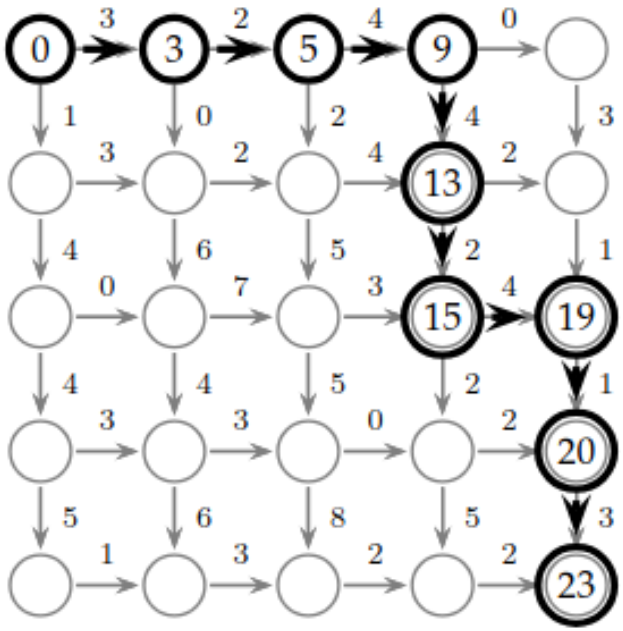
Greedy approach

- Always pick the maximum edge for each vertex



Greedy approach

- Always pick the maximum edge for each vertex
- Can we do better?

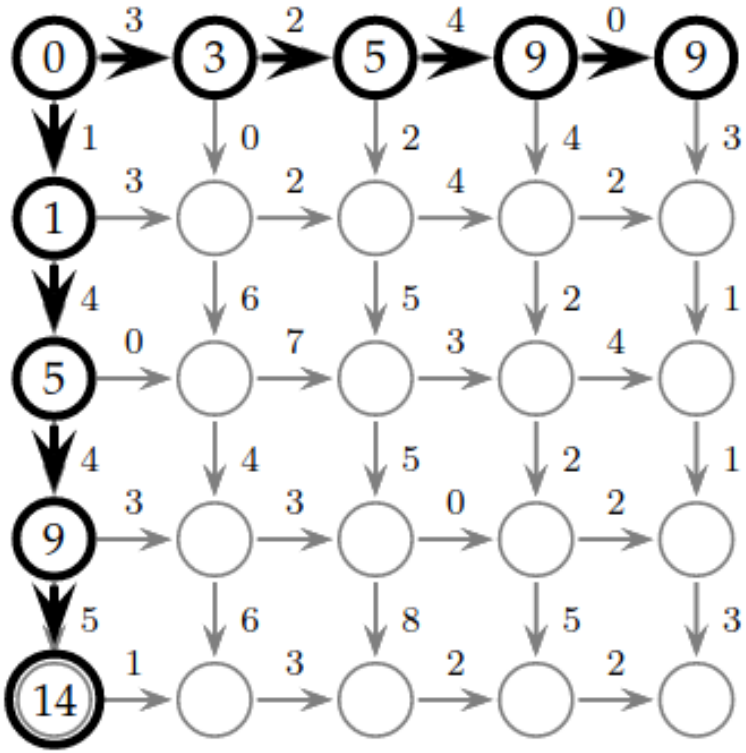


DP Approach: easier subproblems

- Let us solve a more general problem: find the longest path from source to an arbitrary vertex (i, j) with $0 \leq i \leq n, 0 \leq j \leq m$.

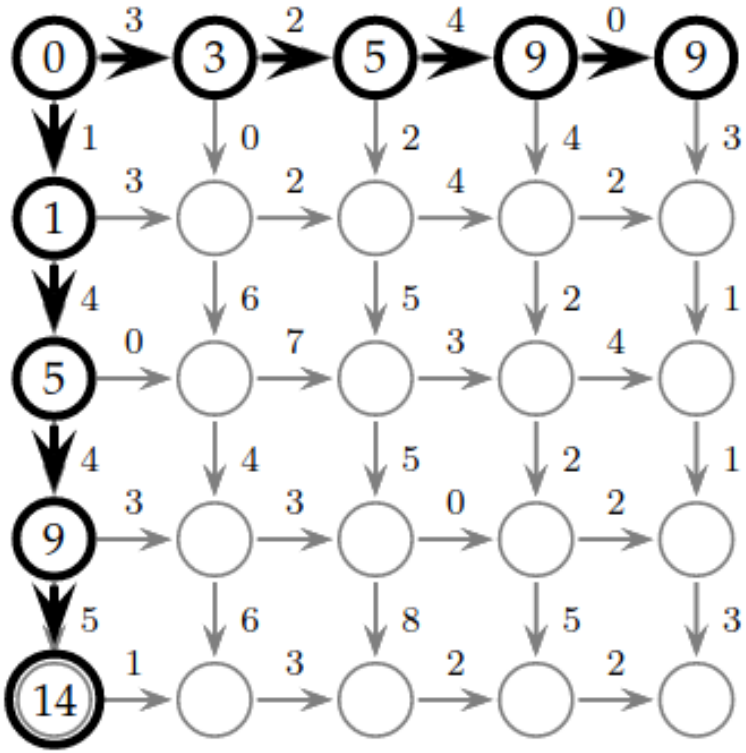
DP Approach: easier subproblems

- Let us solve a more general problem: find the longest path from source to an arbitrary vertex (i, j) with $0 \leq i \leq n, 0 \leq j \leq m$.
- Let $s_{i,j}$ denotes the optimal solution for the vertex (i, j).



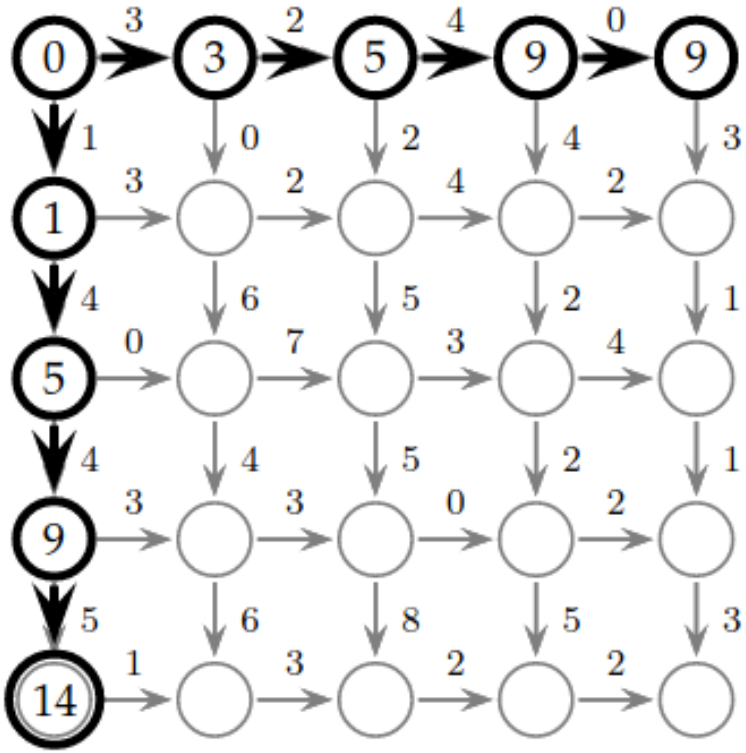
DP Approach: easier subproblems

- Let us solve a more general problem: find the longest path from source to an arbitrary vertex (i, j) with $0 \leq i \leq n, 0 \leq j \leq m$.
- Let $s_{i,j}$ denotes the optimal solution for the vertex (i, j).
- Finding $s_{0,j}$ (for $0 \leq j \leq m$) is not hard because tourists cannot choose an arbitrary path: they can only go east.

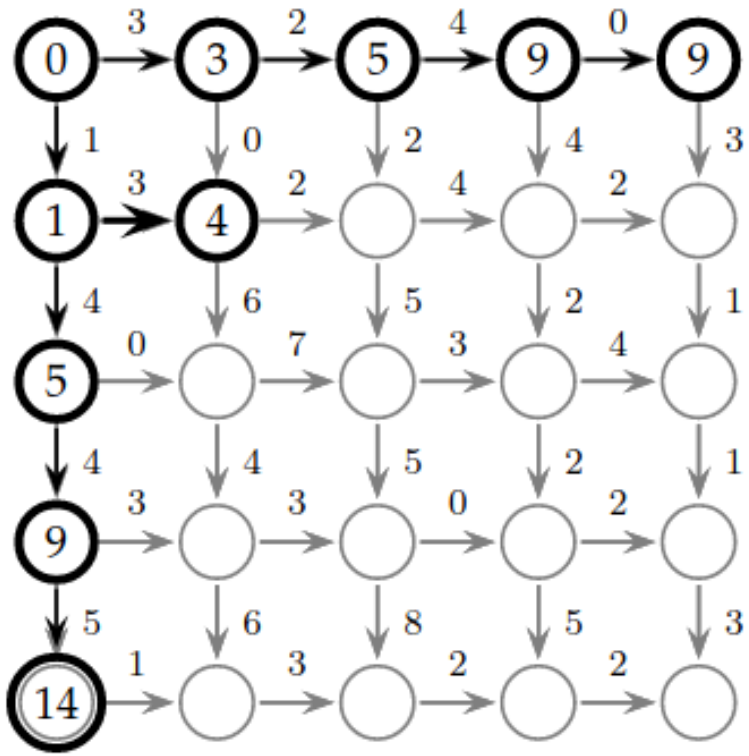


DP Approach: easier subproblems

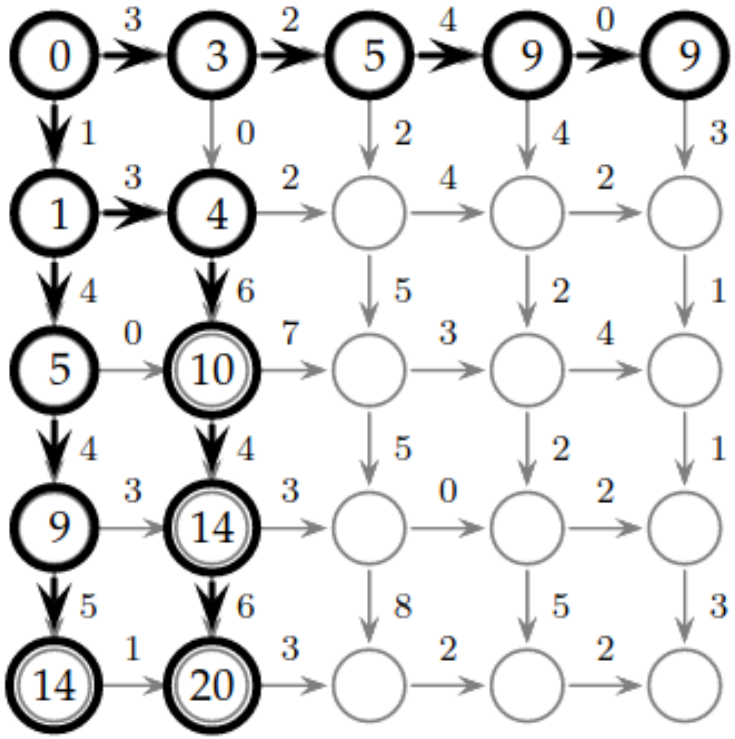
- Let us solve a more general problem: find the longest path from source to an arbitrary vertex (i, j) with $0 \leq i \leq n, 0 \leq j \leq m$.
- Let $s_{i,j}$ denotes the optimal solution for the vertex (i, j).
- Finding $s_{0,j}$ (for $0 \leq j \leq m$) is not hard because tourists cannot choose an arbitrary path: they can only go east.
- The same for $s_{i,0}$ (for $0 \leq i \leq n$).



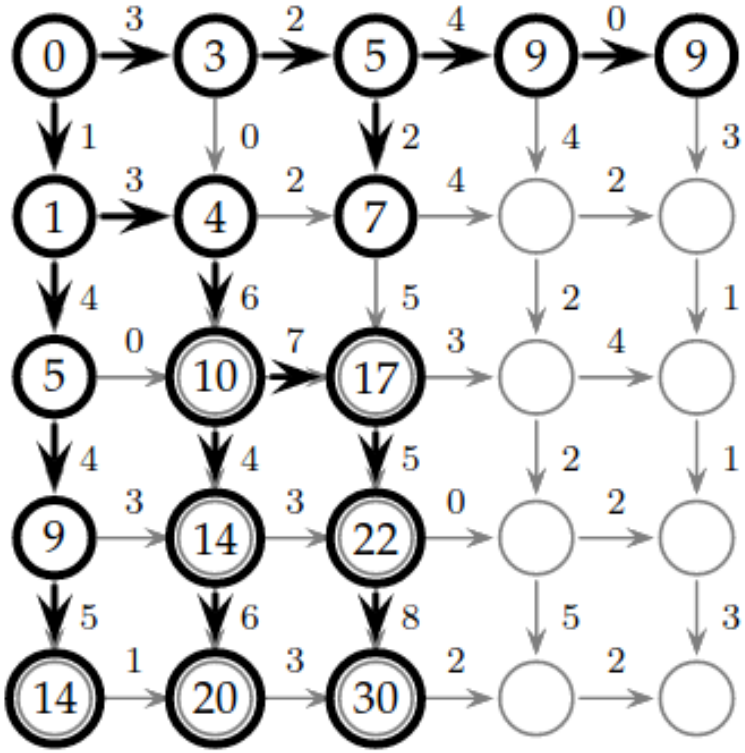
- Let us find the solution for $s_{1,1}$.
- Tourist can arrive to $(1,1)$ in only two ways: from $(0,1)$ or from $(1,0)$
- The best path will be: either (1) $s_{0,1}$ + the weight from $(0,1)$ to $(1,1)$ or (2) $s_{1,0}$ + the weight from $(1,0)$ to $(1,1)$.
- We take the largest value.



- Similar logic to $s_{2,1}$, $s_{3,1}$ and so on.

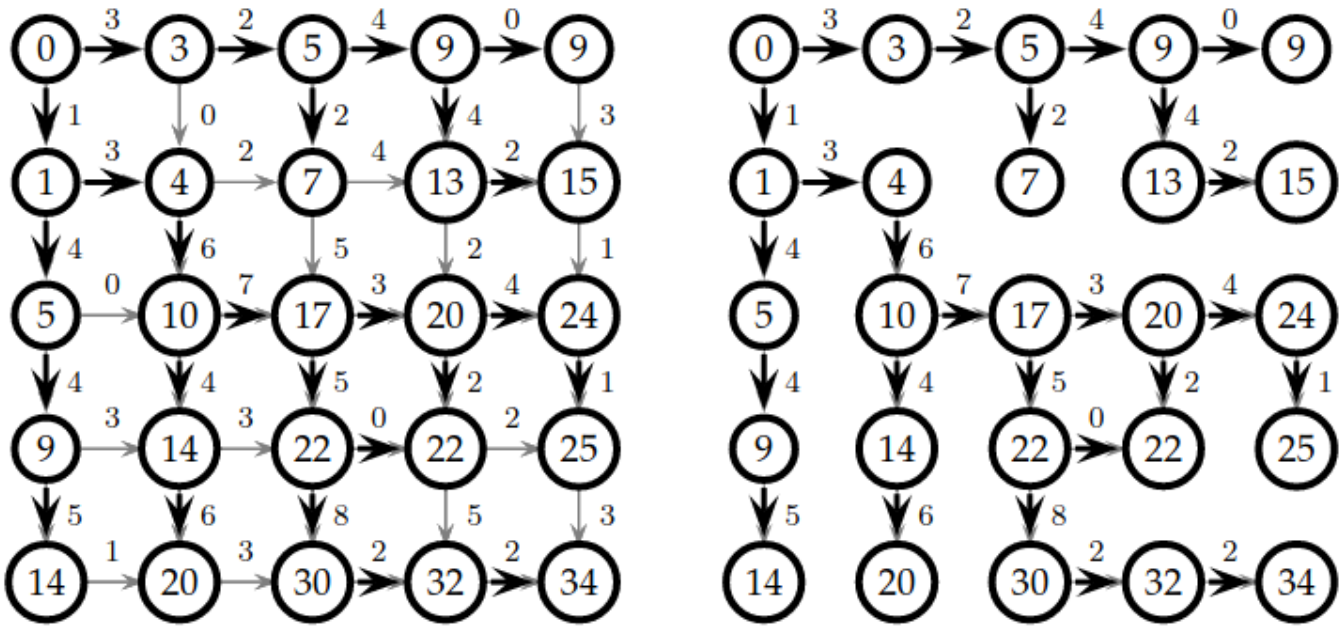


- Let us find the solution for $s_{1,2}$.
- Tourist can arrive to $(1, 2)$ in only two ways: from $(1, 1)$ or from $(2, 0)$
- The best path will be: either (1) $s_{1,1}$ + the weight from $(1, 1)$ to $(1, 2)$ or (2) $s_{0,2}$ + the weight from $(0, 2)$ to $(1, 2)$.
- We take the largest value.
- Similar logic to $s_{2,2}$, $s_{3,2}$ and so on.



We can compute the optimal solution for each vertex with this approach.

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{ weight of the edge between } (i-1,j) \text{ and } (i,j) \\ s_{i,j-1} + \text{ weight of the edge between } (i,j-1) \text{ and } (i,j) \end{cases}$$



DP Approach: the algorithm

- $\overset{\downarrow}{\mathbf{w}}$ two-dimensional array representing the weights of the grid's edges that run north to south
- $\overrightarrow{\mathbf{w}}$ is a two-dimensional array representing the weights of the grid's edges that run west to east.

```
MANHATTANTOURIST(↓w, →w, n, m)
1  s_{0,0} ← 0
2  for i ← 1 to n
3      s_{i,0} ← s_{i-1,0} + ↓w_{i,0}
4  for j ← 1 to m
5      s_{0,j} ← s_{0,j-1} + →w_{0,j}
6  for i ← 1 to n
7      for j ← 1 to m
8          s_{i,j} ← max { s_{i-1,j} + ↓w_{i,j}
                         { s_{i,j-1} + →w_{i,j}
9  return s_{n,m}
```

Initial Conditions

Recurrence

DP Approach: the algorithm

- $\overset{\downarrow}{\mathbf{w}}$ two-dimensional array representing the weights of the grid's edges that run north to south
- $\overrightarrow{\mathbf{w}}$ is a two-dimensional array representing the weights of the grid's edges that run west to east.
- The cost is $\mathcal{O}(nm)$

```
MANHATTANTOURIST(↓w, →w, n, m)
1  s_{0,0} ← 0
2  for i ← 1 to n
3      s_{i,0} ← s_{i-1,0} + ↓w_{i,0}
4  for j ← 1 to m
5      s_{0,j} ← s_{0,j-1} + →w_{0,j}
6  for i ← 1 to n
7      for j ← 1 to m
8          s_{i,j} ← max { s_{i-1,j} + ↓w_{i,j}
                         { s_{i,j-1} + →w_{i,j}
9  return s_{n,m}
```

Initial Conditions

Recurrence

Steps to design a DP algorithm.

Dynamic programming typically applies to *optimization problems*: each solution has a value (i.e., it is a number) and you want to find a solution with the optimal (minimum or maximum) value. To develop a dynamic-programming algorithm, follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

When do we use DP in bioinformatics?

Edit Distance

- We use the notion of Vladimir Levenshtein introduced in 1966
- Edit distance – the **minimum** number of editing operations needed to transform one string into another (insert/delete symbol or substitute one symbol for another).

Alignment of ATATATAT vs TATAAT

A	T	A	T	A	T	A	T
	:	:	:	:	:	:	:
-	T	A	T	A	-	A	T

Edit Distance Algorithm using Dynamic Programming

- Assume two strings:
  - v (of n characters)
  - w (of m characters)
- The alignment of v, w is a two-row matrix such that
  - first row: contains the characters of v (in order)
  - second row: contains the characters of w (in order)
  - spaces are interspersed throughout the table, *no replaces*
- Characters in each string appear in order, though not necessarily adjacently.

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

- No column contains spaces in both rows.
- At most $n + m$ columns.

Edit Distance Algorithm using Dynamic Programming

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

- Matches – columns with the same letter,
- Mismatches – columns with different letters.
- Columns containing one space are called indels
  - Space on top row: insertions
  - Space on bottom row: deletions

$$\#\,\text{matches} + \#\,\text{mismatches} + \#\,\text{indels} < n + m$$

Representing the rows

v	A	T	-	G	T	T	A	T	-
w	A	T	C	G	T	-	A	-	C

- One way to represent v
  - AT-CGTAT-
- One way to represent w
  - ATCGT-A-C
- Another way to represent v
  - AT-CGTAT-
  - 122345677
  - number of symbols of v present up to a given position
- Similarly, to represent w
  - ATCGT-A-C
  - 123455667

Representing the rows

v	A	T	-	G	T	T	A	T	-
w	A	T	C	G	T	-	A	-	C

v	1	2	2	3	4	5	6	7	7
w	1	2	3	4	5	5	6	6	7

can be viewed as a coordinate in 2-dimensional $n \times m$ grid:

$$\binom{0}{0} \binom{1}{1} \binom{2}{2} \binom{2}{3} \binom{3}{4} \binom{4}{5} \binom{5}{5} \binom{6}{6} \binom{7}{6} \binom{7}{7}$$

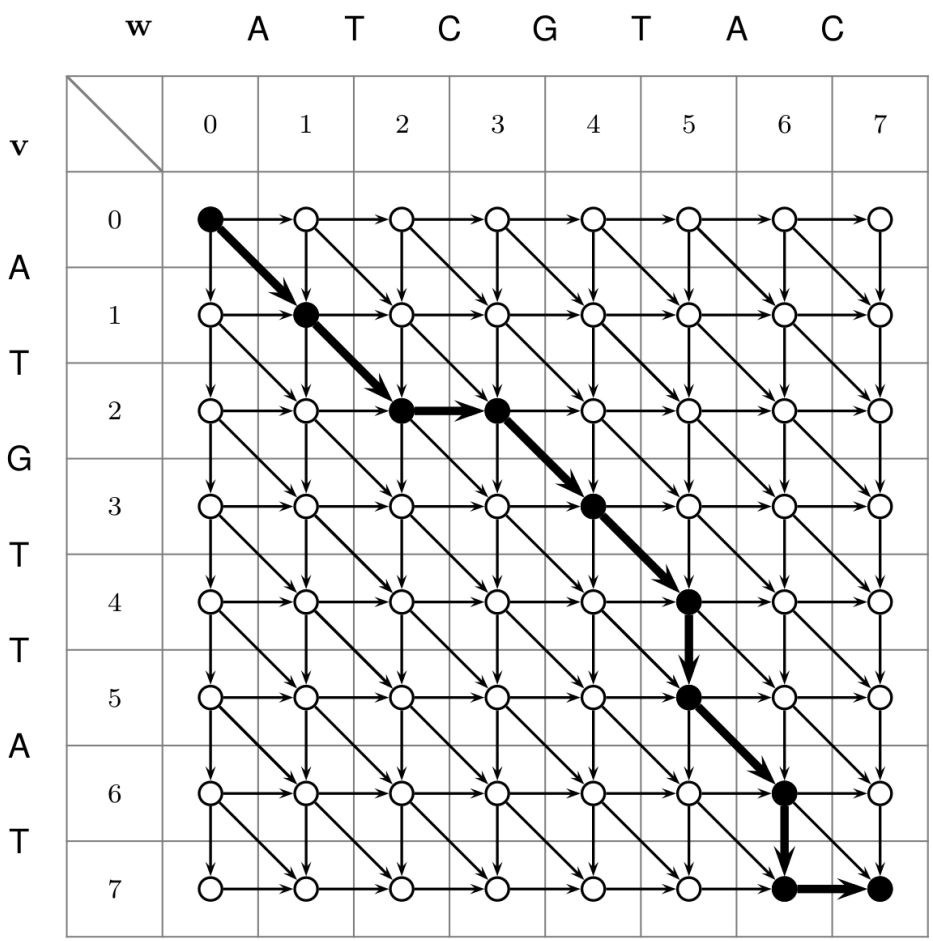
The entire alignment is simply a path:

$(0,0) \to (1,1) \to (2,2) \to (2,3) \to (3,4) \to (4,5) \to (5,5) \to (6,6) \to (7,6) \to (7,7)$

Edit distance graph

- Edit graph: a grid of n, m size.
- The edit graph will help us in calculating the edit distance.
- Alignment: a **path** from $(0, 0)$ to (n, m).
- Every alignment corresponds to a **path** in the edit graph.
- Diagonal movement at point i, j correspond to column $\begin{pmatrix} v_i \\ w_j \end{pmatrix}$
- Horizontal movement correspond to column $\begin{pmatrix} - \\ w_j \end{pmatrix}$
- Vertical movement correspond to column $\begin{pmatrix} v_i \\ - \end{pmatrix}$

Edit distance graph



Edit Distance Recurrence

- Let $s_{i,j}$ be the number of transformations for the i-prefix of v_i and j-prefix of w_i
- $s_{i,0} = i$ and $s_{0,j} = j$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$
- $s_{i,j}$ satisfies the following recurrence:

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

Edit Distance Python program

```
def edit_distance(s1, s2):
    m=len(s1)+1
    n=len(s2)+1

    tbl = {}
    for i in range(m): tbl[i,0]=i
    for j in range(n): tbl[0,j]=j
    for i in range(1, m):
        for j in range(1, n):
            cost = 0 if s1[i-1] == s2[j-1] else 1
            tbl[i,j] = min(tbl[i, j-1]+1,
                           tbl[i-1, j]+1,
                           tbl[i-1, j-1]+cost)

    return tbl[i,j]
```

Scoring the alignment

- In the Edit Distance problem we minimized the **number of operations** (insertions, deletions, substitutions) needed to transform one sequence into another.
- This can also be viewed as assigning a **cost** to each column of the alignment: matches cost 0, edits cost 1.
- More generally, instead of minimizing a cost we can **maximize a score**.

Scoring the alignment

- In the Edit Distance problem we minimized the **number of operations** (insertions, deletions, substitutions) needed to transform one sequence into another.
- This can also be viewed as assigning a **cost** to each column of the alignment: matches cost 0, edits cost 1.
- More generally, instead of minimizing a cost we can **maximize a score**.
- We introduce the notion of a **scoring function**, which assigns a value to each column (match, mismatch, or gap) and sums them up.

Scoring the alignment

- In the Edit Distance problem we minimized the **number of operations** (insertions, deletions, substitutions) needed to transform one sequence into another.
- This can also be viewed as assigning a **cost** to each column of the alignment: matches cost 0, edits cost 1.
- More generally, instead of minimizing a cost we can **maximize a score**.
- We introduce the notion of a **scoring function**, which assigns a value to each column (match, mismatch, or gap) and sums them up.
- *Example:* $+1$ for a match, 0 otherwise. The total score is the sum of column scores.

Scoring the alignment

- In the Edit Distance problem we minimized the **number of operations** (insertions, deletions, substitutions) needed to transform one sequence into another.
- This can also be viewed as assigning a **cost** to each column of the alignment: matches cost 0, edits cost 1.
- More generally, instead of minimizing a cost we can **maximize a score**.
- We introduce the notion of a **scoring function**, which assigns a value to each column (match, mismatch, or gap) and sums them up.
- *Example:* $+1$ for a match, 0 otherwise. The total score is the sum of column scores.
- By choosing different scoring functions we obtain different string comparison problems (e.g., LCS, global alignment, etc.).

Longest Common Subsequence (LCS)

...if $+1$ *if in* v_i *and* w_i *same letter,* 0 *otherwise...*

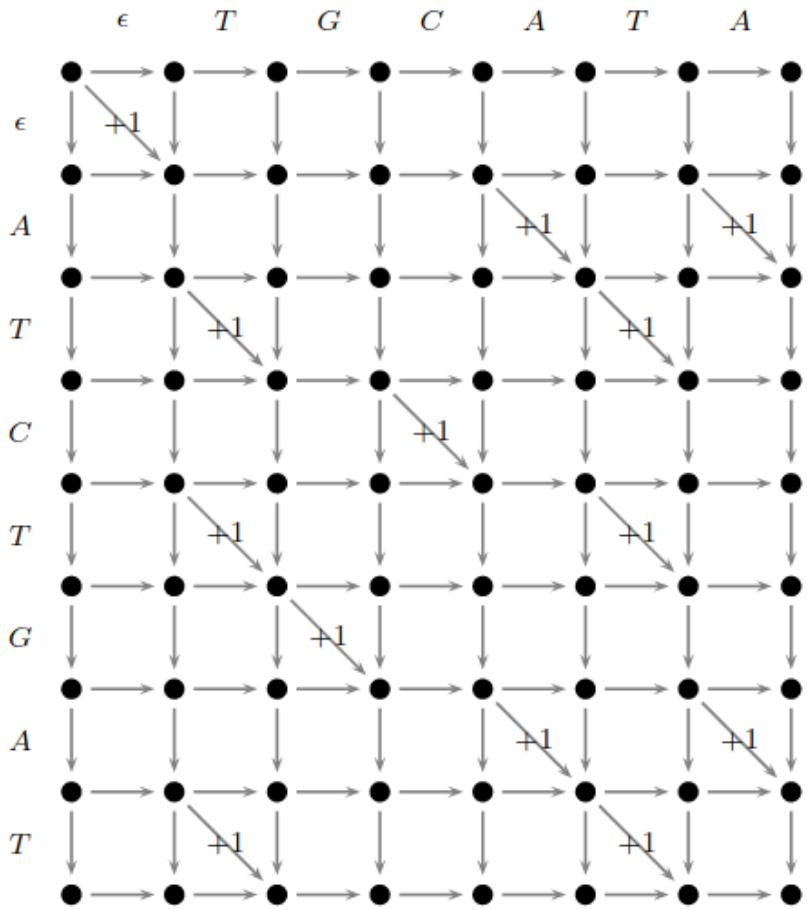
- **Similar problem to Edit Distance**
- A *subsequence* is an ordered sequence of characters (not necessarily, consecutive).
- For ATTGCTA, AGCA is a subsequence, TGTT is not.
- A subsequence is *common* to two strings if it is a subseq of them both
- TCTA is a common to both A**TCT**G**A**T and **T**G**C**A**TA**

Longest Common Subsequence Problem:
Find the longest subsequence common to two strings.

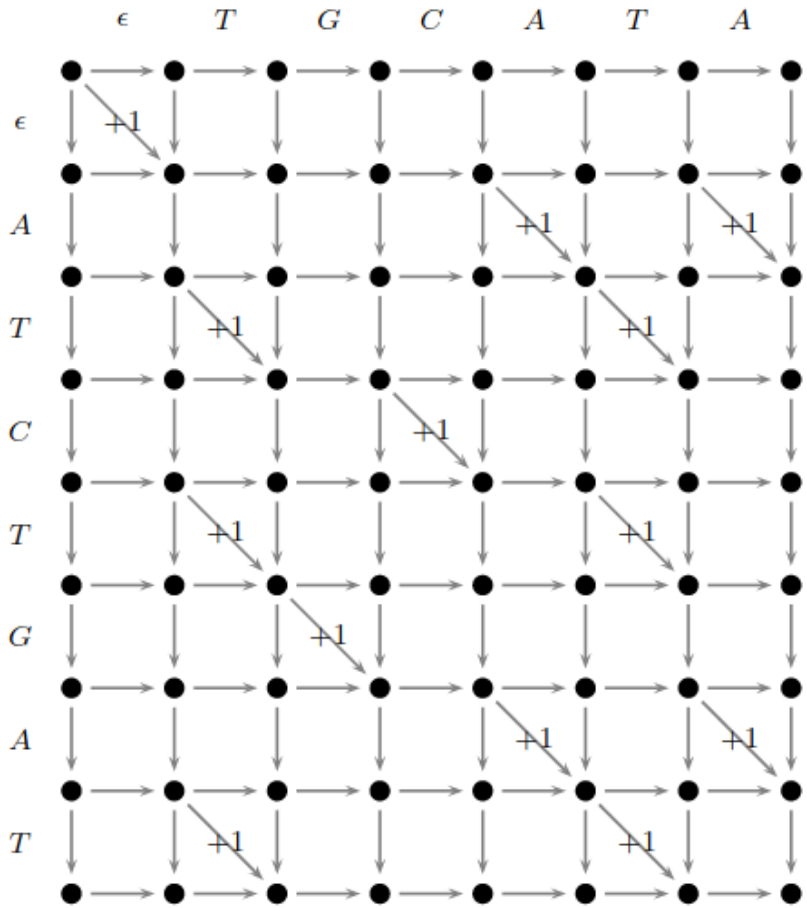
Input: Two strings, **v** and **w**.

Output: The longest common subsequence of **v** and **w**.

LCS Edit Graph



LCS Edit Graph



Similar to Manhattan Tourist problem

Recurrence in LCS

- Let $s_{i,j}$ be the length of LCS for the i-prefix of v_i and j-prefix of w_i

Recurrence in LCS

- Let $s_{i,j}$ be the length of LCS for the i-prefix of v_i and j-prefix of w_i
- $s_{i,0} = s_{0,j} = 0$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$

Recurrence in LCS

- Let $s_{i,j}$ be the length of LCS for the i-prefix of v_i and j-prefix of w_i
- $s_{i,0} = s_{0,j} = 0$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$
- $s_{i,j}$ satisfies the following recurrence:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$$

LCS Algorithm

LCS(**v**, **w**)

1 **for** $i \leftarrow 0$ **to** n

2 $\quad s_{i,0} \leftarrow 0$

3 **for** $j \leftarrow 1$ **to** m

4 $\quad s_{0,j} \leftarrow 0$

5 **for** $i \leftarrow 1$ **to** n

6 $\quad$ **for** $j \leftarrow 1$ **to** m

7 $$s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$$

8 **return** $(s_{n,m}, \mathbf{b})$

Global Sequence Alignment

- The LCS problem corresponds to a rather restrictive scoring that awards 1 for matches and does not penalize indels.

Global Sequence Alignment

- The LCS problem corresponds to a rather restrictive scoring that awards 1 for matches and does not penalize indels.
- To generalize scoring, we extend the k-letter alphabet to include the gap character "-", and consider an arbitrary $(k+1) \times (k+1)$ scoring matrix δ.

Global Sequence Alignment

- The LCS problem corresponds to a rather restrictive scoring that awards 1 for matches and does not penalize indels.
- To generalize scoring, we extend the k-letter alphabet to include the gap character "-", and consider an arbitrary $(k+1) \times (k+1)$ scoring matrix δ.
- The score of the column (x, y) in the alignment is $\delta(x, y)$ and the alignment score is defined as the sum of the scores of the columns. The recurrence will be:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \\ s_{i-1,j-1} + \delta(v_i, w_j) \end{cases}$$

LCS, Edit Distance and Global Alignment

LCS Longest Common Subsequence.

- Ops: Match / Skip (no substitution)
- Score: $+1$ for match, 0 otherwise
- Goal: maximize length
- Use: find common patterns

Edit Distance Levenshtein distance.

- Ops: Insert, Delete, Substitute
- Cost: $+1$ per edit, 0 for match
- Goal: minimize cost
- Use: measure dissimilarity

Global Alignment Needleman–Wunsch algorithm.

- Ops: Insert, Delete, Substitute (weighted)
- Score: $\delta(x, y)$ from scoring matrix
- Goal: maximize total score
- Use: DNA / protein alignment

All rely on Dynamic Programming on an $n \times m$ grid, differing only by scoring and optimization objective.