Dynamic Programming: Edit Distance
Outline

- DNA Sequence Comparison: First Success Stories
- Change Problem
- Manhattan Tourist Problem
- Longest Paths in Graphs
- Sequence Alignment
- Edit Distance
- Longest Common Subsequence Problem
- Dot Matrices
DNA Sequence Comparison: First Success Story

- Finding sequence similarities with genes of known function is a common approach to infer a newly sequenced gene’s function.

- In 1984 Russell Doolittle and colleagues found similarities between cancer-causing gene and normal growth factor (PDGF) gene.
Cystic Fibrosis

- **Cystic fibrosis** (CF) is a chronic and frequently fatal genetic disease of the body's mucus glands (abnormally high level of mucus in glands). CF primarily affects the respiratory systems in children.
- Mucus is a slimy material that coats many epithelial surfaces and is secreted into fluids such as saliva.
Cystic Fibrosis: Inheritance

• In early 1980s biologists hypothesized that CF is an autosomal recessive disorder caused by mutations in a gene that remained unknown till 1989

• Heterozygous carriers are asymptomatic

• Must be homozygously recessive in this gene in order to be diagnosed with CF
Finding Similarities between the Cystic Fibrosis Gene and ATP binding proteins

- ATP binding proteins are present on cell membrane and act as transport channel
- In 1989 biologists found similarity between the cystic fibrosis gene and ATP binding proteins
- A plausible function for cystic fibrosis gene, given the fact that CF involves sweet secretion with abnormally high sodium level
Cystic Fibrosis: Mutation Analysis

If a high % of cystic fibrosis (CF) patients have a certain mutation in the gene and the normal patients don’t, then that could be an indicator of a mutation that is related to CF

A certain mutation was found in 70% of CF patients, convincing evidence that it is a predominant genetic diagnostics marker for CF
Cystic Fibrosis and CFTR Gene:
Cystic Fibrosis and the CFTR Protein

• CFTR (Cystic Fibrosis Transmembrane conductance Regulator) protein is acting in the cell membrane of epithelial cells that secrete mucus
• These cells line the airways of the nose, lungs, the stomach wall, etc.
Mechanism of Cystic Fibrosis

- The **CFTR protein** (1480 amino acids) regulates a chloride ion channel.
- Adjusts the “wateriness” of fluids secreted by the cell.
- Those with cystic fibrosis are missing one single amino acid in their CFTR.
- Mucus ends up being too thick, affecting many organs.
Bring in the Bioinformaticians

• Gene similarities between two genes with known and unknown function alert biologists to some possibilities
• Computing a similarity score between two genes tells how likely it is that they have similar functions
• **Dynamic programming** is a technique for revealing similarities between genes
• The **Change Problem** is a good problem to introduce the idea of dynamic programming
The Change Problem

**Goal**: Convert some amount of money $M$ into given denominations, using the fewest possible number of coins

**Input**: An amount of money $M$, and an array of $d$ denominations $c = (c_1, c_2, \ldots, c_d)$, in a decreasing order of value ($c_1 > c_2 > \ldots > c_d$)

**Output**: A list of $d$ integers $i_1, i_2, \ldots, i_d$ such that

$$c_1i_1 + c_2i_2 + \ldots + c_di_d = M$$

and $i_1 + i_2 + \ldots + i_d$ is minimal
Change Problem: Example

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min # of coins</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Only one coin is needed to make change for the values 1, 3, and 5
Change Problem: Example (cont’d)

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min # of coins</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

However, two coins are needed to make change for the values 2, 4, 6, 8, and 10.
Change Problem: Example (cont’d)

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

<table>
<thead>
<tr>
<th>Value</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min # of coins</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Lastly, three coins are needed to make change for the values 7 and 9.
Change Problem: Recurrence

This example is expressed by the following recurrence relation:

\[
\text{minNumCoins}(M) = \min\left\{ \begin{array}{l}
\text{minNumCoins}(M-1) + 1 \\
\text{minNumCoins}(M-3) + 1 \\
\text{minNumCoins}(M-5) + 1 \\
\end{array} \right. 
\]
Change Problem: Recurrence (cont’d)

Given the denominations \( c: c_1, c_2, \ldots, c_d \), the recurrence relation is:

\[
\text{minNumCoins}(M) = \min\left\{ \text{minNumCoins}(M-c_1) + 1, \text{minNumCoins}(M-c_2) + 1, \ldots, \text{minNumCoins}(M-c_d) + 1 \right\}
\]
Change Problem: A Recursive Algorithm

1. **RecursiveChange**(\(M, c, d\))
2. if \( M = 0 \)
3. return 0
4. \( bestNumCoins \leftarrow \) infinity
5. for \( i \leftarrow 1 \) to \( d \)
6. if \( M \geq c_i \)
7. \( numCoins \leftarrow RecursiveChange(M - c_i, c, d) \)
8. if \( numCoins + 1 < bestNumCoins \)
9. \( bestNumCoins \leftarrow numCoins + 1 \)
10. return \( bestNumCoins \)
RecursiveChange Is Not Efficient

- It recalculates the optimal coin combination for a given amount of money repeatedly.

  - i.e., \( M = 77, c = (1,3,7) \):
    - Optimal coin combo for 70 cents is computed 9 times! \((77 - 7), (77 - 3 - 3 - 1), (77 - 3 - 1 - 3), \ldots\)
    - the optimal combination for 20 cents will be recomputed billions of times → this algorithm is impractical.
We Can Do Better

- Save results of each computation for 0 to $M$
- We can do a reference call to find an already computed value, instead of re-computing each time
- Running time $M \times d$, where $M$ is the value of money and $d$ is the number of denominations
The Change Problem: Dynamic Programming

1. \( \text{DPChange}(M, c, d) \)
2. \( \text{bestNumCoins}_0 \leftarrow 0 \)
3. for \( m \leftarrow 1 \) to \( M \)
4. \( \text{bestNumCoins}_m \leftarrow \text{infinity} \)
5. for \( i \leftarrow 1 \) to \( d \)
6. \( \text{if } m \geq c_i \)
7. \( \text{if } \text{bestNumCoins}_{m - c_i + 1} < \text{bestNumCoins}_m \)
8. \( \text{bestNumCoins}_m \leftarrow \text{bestNumCoins}_{m - c_i + 1} \)
9. return \( \text{bestNumCoins}_M \)
DPChange: Example

\[ c = (1, 3, 7) \]
\[ M = 9 \]
Manhattan Tourist Problem (MTP)

Imagine seeking a path (from source to sink) to travel (only eastward and southward) with the most number of attractions (*) in the Manhattan grid.
Imagine seeking a path (from source to sink) to travel (only eastward and southward) with the most number of attractions (*) in the Manhattan grid.
Manhattan Tourist Problem: Formulation

Goal: Find the longest path in a weighted grid.

Input: A weighted grid $G$ with two distinct vertices, one labeled “source” and the other labeled “sink”

Output: A longest path in $G$ from “source” to “sink”
MTP: Greedy Algorithm Is Not Optimal

promising start, but leads to bad choices!
MTP: Simple Recursive Program

\[
\text{MT}(n,m) =
\begin{align*}
\text{if } n &= 0 \text{ or } m = 0 \\
& \quad \text{return } MT(n,m) \\
x & \leftarrow MT(n-1,m) + \\
& \quad \text{length of the edge from } (n-1,m) \text{ to } (n,m) \\
y & \leftarrow MT(n,m-1) + \\
& \quad \text{length of the edge from } (n,m-1) \text{ to } (n,m) \\
& \text{return } \max\{x,y\}
\end{align*}
\]
MTP: Simple Recursive Program

\[ MT(n, m) \]
\[ x \leftarrow MT(n-1, m) + \text{length of the edge from } (n-1, m) \text{ to } (n, m) \]
\[ y \leftarrow MT(n, m-1) + \text{length of the edge from } (n, m-1) \text{ to } (n, m) \]

return \( \min\{x, y\} \)

What’s wrong with this approach?
**MTP: Dynamic Programming**

- Calculate optimal path score for each vertex in the graph
- Each vertex’s score is the maximum of the prior vertices score plus the weight of the respective edge in between
MTP: Dynamic Programming (cont’d)

source

\[
\begin{align*}
S_{2,0} &= 8 \\
S_{1,1} &= 4 \\
S_{0,2} &= 3
\end{align*}
\]
MTP: Dynamic Programming (cont’d)

source

S_{3,0} = 8
S_{2,1} = 9
S_{1,2} = 13

S_{3,0} = 8
S_{2,1} = 9
S_{1,2} = 13
MTP: Dynamic Programming (cont’d)

source

![Diagram of MTP: Dynamic Programming](image)

- $S_{1,3} = 8$
- $S_{2,2} = 12$
- $S_{3,1} = 9$

greedy alg. fails!
MTP: Dynamic Programming (cont’d)

Source

\begin{array}{llllllll}
& 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 5 & 3 & 1 & 4 & 7 & 9 & 10 & 11 & 12 \\
2 & 8 & 6 & 5 & 8 & 11 & 13 & 14 & 15 & 16 \\
3 & 8 & 6 & 5 & 8 & 11 & 13 & 14 & 15 & 16 \\
\end{array}

**S_{2,3} = 15**

**S_{3,2} = 9**
MTP: Dynamic Programming
(cont’d)

(source)

Done!

(showing all back-traces)

\[ S_{3,3} = 16 \]
MTP: Recurrence

Computing the score for a point \((i,j)\) by the recurrence relation:

\[
s_{i,j} = \max \left\{ s_{i-1,j} + \text{weight of the edge between } (i-1, j) \text{ and } (i, j),
                     s_{i,j-1} + \text{weight of the edge between } (i, j-1) \text{ and } (i, j) \right\}
\]

The running time is \(n \times m\) for a \(n\) by \(m\) grid

\((n = \# \text{ of rows}, m = \# \text{ of columns})\)
Manhattan Is Not A Perfect Grid

What about diagonals?

- The score at point B is given by:

\[ s_B = \max \left\{ s_{A_1} + \text{weight of the edge } (A_1, B), s_{A_2} + \text{weight of the edge } (A_2, B), s_{A_3} + \text{weight of the edge } (A_3, B) \right\} \]
Computing the score for point $x$ is given by the recurrence relation:

$$s_x = \max_{y \in \text{Predecessors}(x)} \left( s_y + \text{weight of vertex } (y, x) \right)$$

- Predecessors $(x)$ – set of vertices that have edges leading to $x$

- The running time for a graph $G(V, E)$ ($V$ is the set of all vertices and $E$ is the set of all edges) is $O(E)$ since each edge is evaluated once
Traveling in the Grid

• The only hitch is that one must decide on the order in which visit the vertices

• By the time the vertex $x$ is analyzed, the values $s_y$ for all its predecessors $y$ should be computed – otherwise we are in trouble.

• We need to traverse the vertices in some order

• Try to find such order for a directed cycle

???
A numbering of vertices of the graph is called topological ordering of the DAG if every edge of the DAG connects a vertex with a smaller label to a vertex with a larger label (i.e. $i < j$ if $v_i$ becomes before $v_j$).

In other words, if vertices are positioned on a line in an increasing order of labels then all edges go from left to right.
Traversing the Manhattan Grid

• 3 different strategies:
  • a) Column by column
  • b) Row by row
  • c) Along diagonals (useful for parallel computation)
Longest Path in DAG Problem

• **Goal**: Find a longest path between two vertices in a weighted DAG

• **Input**: A weighted DAG $G$ with source and sink vertices

• **Output**: A longest path in $G$ from source to sink
Longest Path in DAG: Dynamic Programming

- Suppose vertex $v$ has indegree 3 and predecessors $\{u_1, u_2, u_3\}$
- Longest path to $v$ from source is:

\[
s_v = \max_{u} \left( s_u + \text{weight of edge from } u \text{ to } v \right)
\]

In General:

\[
s_v = \max_{u} \left( s_u + \text{weight of edge from } u \text{ to } v \right)
\]
Distance Metrics

• Last time, we introduced Hamming distance (number of mismatches between two sequences).
• Edit distance: number of operations needed to transform one sequence into another.
• Edit distance doesn’t need the sequences to be of the same length (unlike, Hamming distance).
Hamming distance always compares $i$-th letter of $v$ with $i$-th letter of $w$

$v = \text{ATATATAT}$

$w = \text{TATATATA}$

Hamming distance:

$d(v, w) = 8$

Computing Hamming distance is a trivial task.
Edit Distance vs Hamming Distance

Hamming distance always compares $i$-th letter of $v$ with $i$-th letter of $w$

$V = ATATATAT$  
$W = TATATATA$

Hamming distance:

$d(v, w) = 8$
Computing Hamming distance is a trivial task

Edit distance may compare $i$-th letter of $v$ with $j$-th letter of $w$

$V = -$  
$W = TATATATA$

Edit distance:

$d(v, w) = 2$
Computing edit distance is a non-trivial task
Hamming distance always compares
\( i \)-th letter of \( v \) with
\( i \)-th letter of \( w \)

\[ \begin{align*}
  v &= \text{ATATATAT} \\
  w &= \text{TATATATA}
\end{align*} \]

Hamming distance:
\[ d(v, w) = 8 \]

Edit distance may compare
\( i \)-th letter of \( v \) with
\( j \)-th letter of \( w \)

\[ \begin{align*}
  v &= -\text{ATATATAT} \\
  w &= \text{TATATATA}
\end{align*} \]

Edit distance:
\[ d(v, w) = 2 \]

(one insertion and one deletion)

How to find what \( j \) goes with what \( i \) ???
Edit Distance: Example

TGCATAT $\rightarrow$ ATCCGAT in 5 steps

1. TGCATAT $\rightarrow$ (delete last T)
2. TGCATA $\rightarrow$ (delete last A)
3. TGCAT $\rightarrow$ (insert A at front)
4. ATGCAT $\rightarrow$ (substitute C for 3rd G)
5. ATCCAT $\rightarrow$ (insert G before last A)

ATCCGAT (Done)
Edit Distance: Example

TGCATAT $\rightarrow$ ATCCCGAT in 5 steps

TGCATAT $\rightarrow$ (delete last T)
TGCATA $\rightarrow$ (delete last A)
TGCAT $\rightarrow$ (insert A at front)
ATGCAT $\rightarrow$ (substitute C for 3rd G)
ATCCCAT $\rightarrow$ (insert G before last A)
ATCCCGAT (Done)

What is the edit distance? 5?
Edit Distance: Example (cont’d)

TGCATAT $\rightarrow$ ATCCGAT in 4 steps

- TGCATAT $\rightarrow$ (insert A at front)
- ATGCATAT $\rightarrow$ (delete 6th T)
- ATGCATA $\rightarrow$ (substitute G for 5th A)
- ATGCATGTA $\rightarrow$ (substitute C for 3rd G)
- ATCCGAT (Done)
Edit Distance: Example (cont’d)

TGCATAT → ATCCGAT in 4 steps

TGCATAT → (insert A at front)
ATGCATAT → (delete 6th T)
ATGCATA → (substitute G for 5th A)
ATGCGTA → (substitute C for 3rd G)
ATCCGAT (Done)

Can it be done in 3 steps???
Alignment: 2 row representation

Given 2 DNA sequences \( v \) and \( w \):

\[
\begin{align*}
v & : \quad \text{ATCTGAT} \quad m = 7 \\
w & : \quad \text{TGCATATA} \quad n = 6
\end{align*}
\]

Alignment: \( 2 \times k \) matrix ( \( k > m, n \) )

<table>
<thead>
<tr>
<th>Letters of ( v )</th>
<th>A</th>
<th>T</th>
<th>--</th>
<th>G</th>
<th>T</th>
<th>T</th>
<th>A</th>
<th>T</th>
<th>--</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letters of ( w )</td>
<td>A</td>
<td>T</td>
<td>C</td>
<td>G</td>
<td>T</td>
<td>--</td>
<td>A</td>
<td>--</td>
<td>C</td>
</tr>
</tbody>
</table>

- 4 matches
- 2 insertions
- 2 deletions
Longest Common Subsequence (LCS) – Alignment without Mismatches

• Given two sequences

  \[ v = v_1 v_2 \ldots v_m \] and \[ w = w_1 w_2 \ldots w_n \]

• The LCS of \( v \) and \( w \) is a sequence of positions in

  \( v: 1 \leq i_1 < i_2 < \ldots < i_t \leq m \)

and a sequence of positions in

  \( w: 1 \leq j_1 < j_2 < \ldots < j_t \leq n \)

such that \( i_t \)-th letter of \( v \) equals to \( j_t \)-letter of \( w \) and \( t \) is maximal
LCS: Example

\begin{array}{cccccccccccc}
  i \text{ coords:} & 0 & 1 & 2 & 2 & 3 & 3 & 4 & 5 & 6 & 7 & 8 \\
  elements of v & A & T & -- & C & -- & T & G & A & T & C \\
  elements of w & -- & T & G & C & A & T & -- & A & -- & C \\
  j \text{ coords:} & 0 & 0 & 1 & 2 & 3 & 4 & 5 & 5 & 6 & 6 & 7 \\
\end{array}

(0,0) \rightarrow (1,0) \rightarrow (2,1) \rightarrow (2,2) \rightarrow (3,3) \rightarrow (3,4) \rightarrow (4,5) \rightarrow (5,5) \rightarrow (6,6) \rightarrow (7,6) \rightarrow (8,7)

Every common subsequence is a path in 2-D grid
LCS: Dynamic Programming

• Find the LCS of two strings

Input: A weighted graph $G$ with two distinct vertices, one labeled “source” one labeled “sink”

Output: A longest path in $G$ from “source” to “sink”

• Solve using an LCS edit graph with diagonals replaced with +1 edges
Every path is a common subsequence.

Every diagonal edge adds an extra element to common subsequence

LCS Problem:
Find a path with maximum number of diagonal edges
Computing LCS

Let $v_i = \text{prefix of } v \text{ of length } i: \ v_1 \ldots v_i$
and $w_j = \text{prefix of } w \text{ of length } j: \ w_1 \ldots w_j$
The length of LCS($v_i,w_j$) is computed by:

$$s_{i, j} = \max \left\{ s_{i-1, j}, s_{i, j-1}, s_{i-1, j-1} + 1 \text{ if } v_i = w_j \right\}$$
Computing LCS (cont’d)

\[ s_{i,j} = \text{MAX} \begin{cases} 
    s_{i-1,j} + 0 \\
    s_{i,j-1} + 0 \\
    s_{i-1,j-1} + 1, \quad \text{if} \quad v_i = w_j
\end{cases} \]
Alignments in Edit Graph

- and → represent indels in \( v \) and \( w \) with score 0.
- \( \downarrow \) represent matches with score 1.
- The score of the alignment path is 5.
Alignment as a Path in the Edit Graph

Every path in the edit graph corresponds to an alignment:
Alignment as a Path in the Edit Graph

Old Alignment

\[ v = \text{AT\_GTTAT\_} \]
\[ w = \text{ATCGT\_A\_C} \]
\[ 012345677 \]

New Alignment

\[ v = \text{AT\_GTTAT\_} \]
\[ w = \text{ATCG\_TA\_C} \]
\[ 0123445667 \]
Alignment as a Path in the Edit Graph

\[
\begin{align*}
v &= \text{AT\_GT\_TAT}\_ \\
w &= \text{ATCGT\_A\_C} \\
(0,0), (1,1), (2,2), (2,3), (3,4), (4,5), (5,5), (6,6), (7,6), (7,7)
\end{align*}
\]
Dynamic Programming Example

Initialize 1st row and 1st column to be all zeroes.
Dynamic Programming Example

\[ S_{i,j} = \max \left\{ \begin{array}{c} S_{i-1, j-1} \quad \text{value from NW +1, if } v_i = w_j \\ S_{i-1, j} \quad \text{value from North (top)} \\ S_{i, j-1} \quad \text{value from West (left)} \end{array} \right. \]
Dynamic Programming Example

Find a match in row and column 2.

\( i=2, j=2,5 \) is a match (T).

\( j=2, i=4,5,7 \) is a match (T).

Since \( v_i = w_j \), \( s_{i,j} = s_{i-1,j-1} + 1 \)

\[
\begin{align*}
    s_{2,2} & = [s_{1,1} = 1] + 1 \\
    s_{2,5} & = [s_{1,4} = 1] + 1 \\
    s_{4,2} & = [s_{3,1} = 1] + 1 \\
    s_{5,2} & = [s_{4,1} = 1] + 1 \\
    s_{7,2} & = [s_{6,1} = 1] + 1
\end{align*}
\]
Backtracking Example

Continuing with the dynamic programming algorithm gives this result.
Backtracking pointers

\[ s_{i,j} = \begin{cases} 
  \max \left( s_{i-1, j-1} + 1 \text{ if } v_i = w_j, \\
  s_{i-1, j}, \\
  s_{i, j-1} \right) 
\end{cases} \]
Backtracking

- Follow the arrows backwards from sink
LCS Algorithm

1. \textbf{LCS}(v, w)
2. \hspace{1em} for \( i \leftarrow 1 \) to \( n \)
3. \hspace{2em} \( s_{i,0} \leftarrow 0 \)
4. \hspace{1em} for \( j \leftarrow 1 \) to \( m \)
5. \hspace{2em} \( s_{0,j} \leftarrow 0 \)
6. \hspace{1em} for \( i \leftarrow 1 \) to \( n \)
7. \hspace{2em} \hspace{1em} for \( j \leftarrow 1 \) to \( m \)
8. \hspace{4em} \( s_{i,j} \leftarrow \max \left\{ s_{i-1,j}, s_{i,j-1}, s_{i-1,j-1} + 1, \text{ if } v_i = w_j \right\} \)
9. \hspace{4em} b_{i,j} \leftarrow \begin{cases} \text{“↑” if } s_{i,j} = s_{i-1,j} \\ \text{“←” if } s_{i,j} = s_{i,j-1} \\ \text{“↖” if } s_{i,j} = s_{i-1,j-1} + 1 \end{cases}
10. \hspace{1em} \text{return } (s_{n,m}, b)
Printing LCS: Backtracking

1. PrintLCS(b, v, i, j)
2. if \( i = 0 \) or \( j = 0 \)
3. return
4. if \( b_{i,j} = \) \( \downarrow \) \( \leftarrow \)
5. PrintLCS(b, v, i-1, j-1)
6. print \( v_i \)
7. else
8. if \( b_{i,j} = \) \( \uparrow \) \( \rightarrow \)
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.

• Why $O(nm)$? The pseudocode consists of a nested “for” loop inside of another “for” loop to set up a $nxm$ matrix.
Back to Edit Distance

- Initialization: $d_{i,0} = i$, and $d_{j,0} = j$

- Recurrence:

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