WYKŁAD 8

Porównywanie Sekwencji i Odległości Edycyjne

Skompilowano: 2025/05/20 o 11:59:32

1 Wprowadzenie

Wielu problemów w informatyce i poza nią dotyczy danych, które mają naturalne uporządkowanie – są to **sekwencje**. Przykłady sekwencji obejmują:

- Ciągi DNA, RNA i białek (biologia molekularna)
- Teksty (przetwarzanie języka naturalnego, bioinformatyka)
- Szeregi czasowe (finanse, analiza danych sensorowych)
- Sekwencje zdarzeń (np. kliknięcia użytkownika na stronie internetowej)

Często zachodzi potrzeba porównania dwóch lub więcej takich sekwencji, aby określić, jak bardzo są do siebie podobne lub w jakich miejscach się różnią.

Dlaczego klasyczne wskaźniki podobieństwa są niewystarczające?

Wiele standardowych miar podobieństwa zostało zaprojektowanych do porównywania zbiorów, gdzie kolejność elementów nie ma znaczenia. Przykładem jest omawiane na wcześniejszych wykładach podobieństwo Jaccarda, które dla dwóch zbiorów A i B mierzy stosunek liczby wspólnych elementów do łącznej liczby unikalnych elementów w obu zbiorach:

$$\sigma_{Jacc}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Rozważmy jednak następujące dwie sekwencje na zbiorze liter:

- Sekwencja 1: (a, b, c)
- Sekwencja 2: (c, b, a)

Jeśli potraktujemy te sekwencje jako zbiory, otrzymamy $S_1 = \{a, b, c\}$ i $S_2 = \{c, b, a\}$.

$$\sigma_{Jacc}(S_1, S_2) = \frac{|\{a, b, c\} \cap \{c, b, a\}|}{|\{a, b, c\} \cup \{c, b, a\}|} = \frac{|\{a, b, c\}|}{|\{c, b, a\}|} = \frac{3}{3} = 1$$

Według współczynnika Jaccarda, te "zbiory" są identyczne, co sugerowałoby, że sekwencje są bardzo podobne. Jednakże, w kontekście sekwencji (np. w biologii molekularnej czy przetwarzaniu tekstu), kolejność elementów jest kluczowa. Sekwencja (a, b, c) jest często uważana za znacząco różną od (c, b, a), ponieważ kolejność jest odwrócona.

Jak sama nazwa wskazuje, w sekwencjach **kolejność elementów ma kluczowe znaczenie**. Położenie danego elementu w sekwencji i jego relacja do sąsiednich elementów często niesie fundamentalną informację. Zmiana kolejności, wstawienie lub usunięcie elementu może całkowicie zmienić "znaczenie" lub "właściwości" sekwencji (np. mutacje w DNA, przestawki w słowach).

Dlatego potrzebujemy miar podobieństwa, które uwzględniają to uporządkowanie, a nie tylko zbiór obecnych elementów. W kolejnych częściach przyjrzymy się kilku podstawowym sposobom porównywania sekwencji.

2 Podstawowe metody porównywania sekwencji

Istnieje wiele metod porównywania sekwencji, różniących się tym, na czym skupiają swoją uwagę (dokładne dopasowanie na początku/końcu, wspólne fragmenty, wspólne elementy w tej samej względnej kolejności, minimalna liczba operacji przekształcających jedną sekwencję w drugą itp.). Zacznijmy od tych, które koncentrują się na elementach wspólnych występujących z zachowaniem zależności kolejnościowych:

2.1 Najdłuższy Wspólny Prefiks (NWP)

Definicja 1. Prefiks sekwencji to początkowy fragment tej sekwencji. **Najdłuższy Wspólny Prefiks (NWP)** dwóch sekwencji S_1 i S_2 to najdłuższa sekwencja, która jest jednocześnie prefiksem S_1 i prefiksem S_2.

Aby znaleźć NWP, porównujemy elementy sekwencji od początku, element po elemencie, aż napotkamy pierwszą różnicę lub dojdziemy do końca krótszej sekwencji.

Przykład – Najdłuższy Wspólny Prefiks

$S_1 = (a, b, c, d, e)$ $S_2 = (a, b, d, f, g)$

Porównujemy: $S_1[1] = a, S_2[1] = a$ (pasuje), $S_1[2] = b, S_2[2] = b$ (pasuje), $S_1[3] = c, S_2[3] = d$ (nie pasuje). Najdłuższy wspólny prefiks S_1 i S_2 to (a, b). Jego długość wynosi 2.

$S_3 = (p, q, r)$ $S_4 = (x, y, z)$

$S_3[1] = p, S_4[1] = x$ (nie pasuje). Najdłuższy wspólny prefiks S_3 i S_4 jest sekwencją pustą. Jego długość wynosi 0.

NWP jest prostą miarą, która informuje nas jedynie o tym, jak bardzo podobny jest początek sekwencji. Przejdźmy do analogicznej miary, skupiającej się na końcowych fragmentach sekwencji.

2.2 Najdłuższy Wspólny Sufiks (NWS)

Definicja 2. Sufiks sekwencji to końcowy fragment tej sekwencji. **Najdłuższy Wspólny Sufiks (NWS)** dwóch sekwencji S_1 i S_2 to najdłuższa sekwencja, która jest jednocześnie sufiksem S_1 i sufiksem S_2.

Aby znaleźć NWS, porównujemy elementy sekwencji od końca, element po elemencie, aż napotkamy pierwszą różnicę lub dojdziemy do początku jednej z sekwencji.

Przykład – Najdłuższy Wspólny Sufiks

$S_1 = (a, b, c, d, e)$ $S_2 = (x, y, c, d, e)$

Porównujemy od końca: $S_1[5] = e, S_2[5] = e$ (pasuje), $S_1[4] = d, S_2[4] = d$ (pasuje), $S_1[3] = c, S_2[3] = c$ (pasuje), $S_1[2] = b, S_2[2] = y$ (nie pasuje). Najdłuższy wspólny sufiks sekwencji S_1 i S_2 to (c, d, e). Jego długość wynosi 3.

$S_3 = (p, q, r)$ $S_4 = (p, q, s)$

$S_3[3] = r, S_4[3] = s$ (nie pasuje). Najdłuższy wspólny sufiks sekwencji S_3 i S_4 jest sekwencją pustą. Jego długość wynosi 0.

NWS informuje nas o podobieństwie na końcu sekwencji. Podobnie jak NWP, jest miarą lokalną, skupiającą się tylko na jednym krańcu sekwencji.

2.3 Najdłuższy Wspólny Podciąg (NWPod)

Definicja 3. Podciąg (ang. *subsequence*) sekwencji powstaje poprzez usunięcie z niej zera lub więcej elementów, **bez zmieniania kolejności** pozostałych elementów. **Najdłuższy Wspólny Podciąg (NWPod)** dwóch sekwencji S_1 i S_2 to najdłuższa sekwencja, która jest jednocześnie podciągiem S_1 i podciągiem S_2.

W odróżnieniu od prefiksu czy sufiksu, podciąg nie musi być spójnym fragmentem sekwencji. Np. w sekwencji (a, b, c, d, e), (a, c, e) jest podciągiem (usunęliśmy b i d), ale nie jest ani prefiksem, ani sufiksem, ani spójnym podciągiem (czyli takim podciągiem "bez przerw", który powstał z oryginalnego ciągu przez usunięcie pewnego prefiksu i pewnego sufiksu).

NWPod jest znacznie bardziej elastyczną miarą podobieństwa, ponieważ pozwala na dopasowywanie elementów w tej samej względnej kolejności, nawet jeśli są one oddalone od siebie w oryginalnych sekwencjach. Znajdowanie NWPod jest bardziej złożonym problemem niż NWP czy NWS i zazwyczaj rozwiązuje się go za pomocą programowania dynamicznego (co omówimy szczegółowo w kolejnych częściach wykładu).

Przykład - NWPod

$S_1 = (a, b, c, d, e)$ $S_2 = (a, x, b, y, c, z, d)$

Nietrudno dostrzec, że najdłuższy wspólny podciąg (elementy wspólne, które pojawiają się w tej samej względnej kolejności w obu sekwencjach) to (a, b, c, d). Jego długość wynosi 4. Powstał przez usunięcie części znaków w poszczególnych sekwencjach: W S_1: $(a, b, c, d, \not{e})$, W S_2: $(a, \not{x}, b, \not{y}, c, \not{z}, d)$.

NWSP daje lepsze wyobrażenie o ogólnym podobieństwie dwóch sekwencji pod względem wspólnych elementów występujących w tej samej kolejności, nawet jeśli pomiędzy nimi występują "niezgodności" (czyli elementy obecne tylko w jednej sekwencji).

2.3.1 Wzór rekurencyjny na długość Najdłuższego Wspólnego Podciągu

Niech S_1 będzie sekwencją długości m oraz S_2 będzie sekwencją długości n. Oznaczmy $S_1[1..i]$ jako prefiks sekwencji S_1 składający się z pierwszych i elementów, a $S_2[1..j]$ jako prefiks sekwencji S_2 składający się z pierwszych j elementów. Element na i-tej pozycji w sekwencji S_1 oznaczmy przez $S_1[i]$, a w sekwencji S_2 przez $S_2[j]$.

Niech $L(i, j)$ oznacza długość najdłuższego wspólnego podciągu prefiksów $S_1[1..i]$ i $S_2[1..j]$. Wzór rekurencyjny na $L(i, j)$ definiuje się następująco:

Dla $i = 0$ lub $j = 0$, długość NWSP jest równa 0 (ponieważ jeden z prefiksów jest pusty):

$$L(i, 0) = 0 \quad \text{dla } i \geq 0$$

$$L(0, j) = 0 \quad \text{dla } j \geq 0$$

Dla $i > 0$ i $j > 0$, rozważamy ostatnie elementy prefiksów:

- Jeśli ostatnie elementy prefiksów są takie same ($S_1[i] = S_2[j]$), to ten wspólny element należy do najdłuższego wspólnego podciągu prefiksów $S_1[1..i]$ i $S_2[1..j]$. Długość NWSP zwiększa się o 1 w stosunku do długości NWSP prefiksów bez tych pasujących elementów:

$$L(i,j) = L(i-1,j-1) + 1 \quad \text{jeśli } S_1[i] = S_2[j]$$

- Jeśli ostatnie elementy prefiksów są różne ($S_1[i] \neq S_2[j]$), to ostatni element S_1 ($S_1[i]$) nie może jednocześnie należeć do NWSP razem z ostatnim elementem S_2 ($S_2[j]$). W takim przypadku, najdłuższy wspólny podciąg prefiksów $S_1[1..i]$ i $S_2[1..j]$ musi być albo NWSP prefiksów $S_1[1..i-1]$ i $S_2[1..j]$ (gdzie ignorujemy $S_1[i]$), albo NWSP prefiksów $S_1[1..i]$ i $S_2[1..j-1]$ (gdzie ignorujemy $S_2[j]$). Wybieramy dłuższą z tych dwóch opcji:

$$L(i,j) = \max(L(i-1,j), L(i,j-1)) \quad \text{jeśli } S_1[i] \neq S_2[j]$$

Łącząc te przypadki, otrzymujemy wzór rekurencyjny w pełnej formie:

$$L(i,j) = \begin{cases} 0 & \text{jeśli } i = 0 \text{ lub } j = 0 \\ L(i-1,j-1) + 1 & \text{jeśli } i > 0, j > 0 \text{ i } S_1[i] = S_2[j] \\ \max(L(i-1,j), L(i,j-1)) & \text{jeśli } i > 0, j > 0 \text{ i } S_1[i] \neq S_2[j] \end{cases}$$

Długość najdłuższego wspólnego podciągu całych sekwencji S_1 i S_2 wynosi $L(m,n)$. Ten wzór stanowi podstawę do skonstruowania algorytmu programowania dynamicznego, który efektywnie oblicza $L(i,j)$ dla wszystkich $0 \leq i \leq m$ i $0 \leq j \leq n$, zwykle przy użyciu tablicy (macierzy) do przechowywania pośrednich wyników i unikania wielokrotnego obliczania tych samych wartości.

2.4 Najdłuższy Wspólny Spójny Podciąg (Substring)

Zanim zdefiniujemy Najdłuższy Wspólny Spójny Podciąg, sprecyzujmy, czym jest spójny podciąg.

Definicja 4. Spójny podciąg (ang. substring) sekwencji to ciąg kolejnych elementów tej sekwencji. Innymi słowy, spójny podciąg jest fragmentem sekwencji bez "przerw".

Na przykład, dla sekwencji (a, b, c, d, e), spójnym podciągiem jest (b, c, d), ale (a, c, e) nie jest, ponieważ elementy c oraz e nie sąsiadują ze sobą bezpośrednio w oryginalnej sekwencji. O spójnym podciągu możemy myśleć, jak o podciągu, która powstał przez odcięcie pewnego prefiksu i/lub pewnego sufiksu z oryginalnego ciągu.

Teraz możemy zdefiniować najdłuższy wspólny spójny podciąg.

Definicja 5. Najdłuższy Wspólny Spójny Podciąg (NWSP), znany również jako *Longest Common Substring* (LCS) lub *Longest Common Continuous Subsequence*, dwóch sekwencji S_1 i S_2, to najdłuższa sekwencja, która jest jednocześnie spójnym podciągiem (substringiem) S_1 i spójnym podciągiem (substringiem) S_2.

W odróżnieniu od Najdłuższego Wspólnego Podciągu (Subsequence), gdzie elementy mogły pochodzić z dowolnych pozycji w oryginalnych sekwencjach (pod warunkiem zachowania ich względnej kolejności), w przypadku NWSP wymagana jest ciągłość elementów w obu oryginalnych sekwencjach.

3 Odległości edycyjne

Alternatywnym podejściem do mierzenia podobieństwa między sekwencjami (lub szerzej, dowolnymi obiektami, które można transformować za pomocą określonych operacji) jest określenie **odległości edycyjnej**. Zamiast szukać wspólnych fragmentów czy podciągów, pytamy: ile "pracy" trzeba włożyć, aby przekształcić jedną sekwencję w drugą?

Definicja 6. Odległość edycyjna (ang. edit distance) między dwiema sekwencjami to minimalna liczba operacji edycyjnych (takich jak wstawienie, usunięcie, zamiana elementu) potrzebnych do przekształcenia jednej sekwencji w drugą.

Różne definicje odległości edycyjnej mogą uwzględniać różne zestawy dozwolonych operacji edycyjnych oraz przypisywać im różne koszty. Najczęściej spotykane operacje to:

- **Wstawienie** (ang. insertion): dodanie elementu.
- **Usunięcie** (ang. deletion): usunięcie elementu.
- **Zamiana** (ang. substitution/replacement): zmiana jednego elementu na inny.
- Czasem także **transpozycja** (ang. transposition): zamiana miejscami dwóch sąsiadujących elementów (np. w odległości Damerau-Levenshteina).

Zazwyczaj każdej operacji przypisany jest koszt (np. 1), a odległość to minimalny łączny koszt sekwencji operacji transformujących pierwszą sekwencję w drugą.

3.1 Odległość Hamminga

Jedną z najprostszych odległości edycyjnych jest odległość Hamminga.

Definicja 7. Odległość Hamminga δ_H (ang. Hamming distance) między dwiema sekwencjami o **jednakowej długości** to liczba pozycji, na których te sekwencje się różnią.

Odległość Hamminga można traktować jako specyficzny przypadek odległości edycyjnej, w którym:

- Dopuszczalną operacją jest wyłącznie **zamiana** (substytucja) z kosztem 1.
- Operacje **wstawienia** i **usunięcia** mają nieskończony koszt (lub są w ogóle niedozwolone), co wymusza porównywanie sekwencji tej samej długości i korygowanie różnic wyłącznie przez zamiany.

Jest to zatem najprostsza forma odległości edycyjnej, ograniczona do porównywania sekwencji równej długości. Odległość Hamminga to minimalna liczba **zamian** pojedynczych elementów potrzebnych do przekształcenia jednej sekwencji w drugą.

Odległość Hamminga ma zastosowanie tylko wtedy, gdy porównywane sekwencje są tej samej długości. Dozwoloną operacją jest tylko zamiana (substytucja). Formalnie, dla sekwencji $S_1 = (s_1, \ldots, s_n)$ i $S_2 = (t_1, \ldots, t_n)$ o długości n:

$$\delta_H(S_1, S_2) = \sum_{i=1}^{n} [s_i \neq t_i],$$

gdzie $[s_i \neq t_i]$ wynosi 1, jeśli $s_i \neq t_i$, i 0, jeśli $s_i = t_i$.

Przykład – odległość Hamminga

$\delta_H((k,o,t),(p,s,a)) = 3$ – sekwencje różnią się na wszystkich 3 pozycjach.

$\delta_H((g,r,a,m),(g,r,e,s)) = 2$ –sekwencje różnią się na dwóch ostatnich pozycjach.

$\delta_H((A,T,G,C),(A,T,G,C)) = 0$ – sekwencje DNA są identyczne.

3.2 Odległość Levenshteina

Bardziej ogólną i szerzej stosowaną miarą odległości edycyjnej jest odległość Levenshteina, która pozwala na operacje wstawienia i usunięcia, dzięki czemu można porównywać sekwencje o różnej długości.

Definicja 8. Odległość Levenshteina (ang. Levenshtein distance) między dwiema sekwencjami to minimalna liczba pojedynczych operacji: **wstawienia**, **usunięcia** lub **zamiany** elementu, potrzebnych do przekształcenia jednej sekwencji w drugą. Każda z tych operacji ma zazwyczaj koszt równy 1.

Odległość Levenshteina jest metryką, co oznacza, że spełnia warunki: nieujemności, zerowej odległości między identycznymi sekwencjami, symetrii oraz nierówności trójkąta.

Wyznaczenie odległości Levenshteina opiera się na podejściu programowania dynamicznego, podobnie jak w przypadku NWSP. Możemy zdefiniować wzór rekurencyjny. Niech S_1 ma długość m, a S_2 ma długość n. Niech $Lev(i,j)$ oznacza odległość Levenshteina między prefiksami $S_1[1..i]$ i $S_2[1..j]$.

Dla warunków brzegowych ($i = 0$ lub $j = 0$):

- Przekształcenie pustej sekwencji w $S_2[1..j]$ wymaga j wstawień: $Lev(0,j) = j$.
- Przekształcenie $S_1[1..i]$ w pustą sekwencję wymaga i usunięć: $Lev(i,0) = i$.

Dla $i > 0$ i $j > 0$ rozważamy ostatnie elementy $S_1[i]$ i $S_2[j]$:

- Jeśli $S_1[i] = S_2[j]$, te elementy pasują i nie wymagają operacji. Minimalna odległość to odległość między pozostałymi prefiksami: $Lev(i,j) = Lev(i-1,j-1)$.
- Jeśli $S_1[i] \neq S_2[j]$, musimy wykonać jedną z trzech operacji na końcach prefiksów, aby dopasować te elementy, a następnie rozwiązać problem dla pozostałych części sekwencji:
  - Zamiana $S_1[i]$ na $S_2[j]$ (koszt 1): Problem redukuje się do $Lev(i-1,j-1)$. Całkowity koszt: $Lev(i-1,j-1)+1$.
  - Usunięcie $S_1[i]$ (koszt 1): Problem redukuje się do $Lev(i-1,j)$ (przekształcamy $S_1[1..i-1]$ w $S_2[1..j]$). Całkowity koszt: $Lev(i-1,j)+1$.
  - Wstawienie $S_2[j]$ (koszt 1): Problem redukuje się do $Lev(i,j-1)$ (przekształcamy $S_1[1..i]$ w $S_2[1..j-1]$, a następnie wstawiamy $S_2[j]$). Całkowity koszt: $Lev(i,j-1)+1$.

  Wybieramy opcję o minimalnym koszcie: $Lev(i,j) = \min(Lev(i-1,j-1)+1, Lev(i-1,j)+1, Lev(i,j-1)+1)$.

Łącząc wszystkie przypadki, otrzymujemy wzór rekurencyjny dla $i, j \geq 0$:

$$Lev(i,j) = \begin{cases} j & \text{jeśli } i = 0 \\ i & \text{jeśli } j = 0 \\ Lev(i-1, j-1) & \text{jeśli } i > 0, j > 0 \text{ i } S_1[i] = S_2[j] \\ \min(Lev(i-1,j-1)+1, Lev(i-1,j)+1, Lev(i,j-1)+1) & \text{jeśli } i > 0, j > 0 \text{ i } S_1[i] \neq S_2[j] \end{cases}$$

Odległość Levenshteina między całymi sekwencjami S_1 i S_2 wynosi $Lev(m, n)$.

Przykład – odległość Levenshteina

Obliczmy odległość Levenshteina między sekwencjami: $S_1 = (k, o, t)$, $S_2 = (p, i, e, s)$.
Spróbujmy kilku sekwencji edycji:

a) kot $\xrightarrow{\text{zamień k na p}}$ pot $\xrightarrow{\text{zamień o na i}}$ pit $\xrightarrow{\text{zamień t na e}}$ pie $\xrightarrow{\text{wstaw s}}$ pies – łącznie 4 operacje.

b) kot $\xrightarrow{\text{wstaw p}}$ pkot $\xrightarrow{\text{zamień k na i}}$ piot $\xrightarrow{\text{zamień o na e}}$ piet $\xrightarrow{\text{usuń t}}$ pie $\xrightarrow{\text{wstaw s}}$ pies – 5 operacji.

c) kot $\xrightarrow{\text{zamień k na p}}$ pot $\xrightarrow{\text{zamień o na i}}$ pit $\xrightarrow{\text{wstaw e}}$ piet $\xrightarrow{\text{zamień t na s}}$ pies – 4 operacje.

Znaleźliśmy kilka sposobów przekształcenia S_1 w S_2 w czterech krokach. Nietrudno uzasadnić, że krócej się nie da, bo S_2 ma długość 4 i żadna z liter tej sekwencji nie występuje w S_1, więc każdą z nich trzeba kolejno dodać przez wstawienie lub podmianę istniejącej litery.

Nawet w powyższym przykładzie widać, że odległości edycyjne mogą sprawić duże problemy obliczeniowe. Istnieje wiele ciągów przekształceń, które nie przybliżają nas do sukcesu i nawet próba przeszukiwania drzewa wszystkich możliwych przekształceń wszerz nadal jest problemem o złożoności wykładniczej.

Dlatego zazwyczaj stosuje się iteracyjny algorytm opierający się na programowaniu dynamicznym. Można go znaleźć pod wieloma nazwami, np. jako **algorytm Wagnera-Fischera**.

Standardowa odległość Levenshteina, uwzględniająca operacje wstawienia, usunięcia i zamiany z kosztem 1, może być efektywnie obliczona za pomocą algorytmu programowania dynamicznego w czasie i pamięci rzędu $O(mn)$, gdzie m i n to długości porównywanych sekwencji. Ta efektywność wynika bezpośrednio ze struktury problemu i natury dozwolonych operacji.

Algorytm programowania dynamicznego dla odległości Levenshteina opiera się na budowaniu tabeli $Lev(i, j)$, gdzie $Lev(i, j)$ to odległość między prefiksem $S_1[1..i]$ a prefiksem $S_2[1..j]$. Wartość w każdej komórce $Lev(i, j)$ zależy tylko od wartości w sąsiednich, "wcześniej" obliczonych komórkach: $Lev(i-1, j-1)$, $Lev(i-1, j)$ i $Lev(i, j-1)$. Te zależności odpowiadają ostatniej operacji (zamiana/brak, usunięcie, wstawienie) w optymalnej ścieżce edycji i mają charakter **lokalny**.

3.3 Inne odległości edycyjne i problemy obliczeniowe

Istnieją również inne definicje odległości edycyjnych, uwzględniające dodatkowe operacje lub przypisujące operacjom inne koszty, np.:

- **Odległość Damerau-Levenshteina**: Dodaje operację transpozycji (zamiany miejscami dwóch sąsiednich znaków), np. "kat" do "akt" wymaga 1 transpozycji. Jest to przydatne przy uwzględnianiu błędów literowych polegających na przestawieniu liter.

- Odległości z różnymi kosztami operacji (np. usunięcie samogłoski kosztuje mniej niż usunięcie spółgłoski, lub koszt zamiany zależy od tego, jakie znaki są zamieniane).

Główne niebezpieczeństwo obliczeniowe związane z próbą rozszerzenia zestawu dopuszczalnych operacji edycyjnych wynika z potencjalnego wprowadzenia zależności **nielokalnych** do rekurencji programowania dynamicznego.

Rozważmy przykład dodania operacji **transpozycji** (zamiany sąsiadujących elementów), jak w odległości Damerau-Levenshteina. Jeśli dopuścimy zamianę miejscami $S_1[i-1]$ i $S_1[i]$ (kosztem 1), co dopasuje je do $S_2[j-1]$ i $S_2[j]$, gdy $S_1[i-1] = S_2[j]$ i $S_1[i] = S_2[j-1]$, to minimalny koszt przekształcenia $S_1[1..i]$ w $S_2[1..j]$ może zależeć od kosztu przekształcenia $S_1[1..i-2]$ w $S_2[1..j-2]$, powiększonego o koszt transpozycji. Oznacza to, że komórka $Lev(i,j)$ może teraz zależeć od komórki $Lev(i-2, j-2)$.

Uwaga 1. Nawet z dodatkową zależnością od $Lev(i-2, j-2)$ (dla transpozycji sąsiadujących elementów), odległość Damerau-Levenshteina (w wersji optymalnej, uwzględniającej transpozycje tylko raz) nadal może być obliczona w czasie $O(mn)$, choć algorytm dynamiczny staje się nieco bardziej złożony (np. wymaga dodatkowej tablicy lub bardziej skomplikowanego przejścia). Złożoność asymptotyczna pozostaje taka sama, ale stały czynnik może wzrosnąć.

3.3.1 Główne wyzwanie – operacje nielokalne lub złożone

Prawdziwe problemy obliczeniowe pojawiają się, gdy dodajemy operacje, które nie dotyczą tylko sąsiednich elementów lub operacje o bardziej złożonym charakterze, np.:

- **Transpozycje niesąsiadujących elementów**: Zamiana elementów na dowolnych pozycjach i i k. Koszt przekształcenia $S_1[1..m]$ w $S_2[1..n]$ poprzez taką operację na końcu wymagałby znajomości optymalnej odległości dla sekwencji z usuniętymi dwoma elementami i wstawionymi w innych miejscach, co drastycznie komplikuje zależności w tabeli programowania dynamicznego.

- **Operacje blokowe**: Wstawienie, usunięcie lub zamiana całego bloku elementów. Koszt operacji zależałby od długości bloku, a rekurencja musiałaby uwzględniać dopasowanie $S_1[1..i-k]$ do $S_2[1..j]$ (usunięcie bloku długości k) lub $S_1[1..i]$ do $S_2[1..j-k]$ (wstawienie bloku długości k), co wymagałoby sprawdzenia wszystkich możliwych długości bloków k.

- **Operacje zależne od kontekstu**: Koszt operacji zależy od elementów *wokół* miejsca edycji (np. odległość znaku na klawiaturze odwołująca się także do sąsiednich znaków w sekwencji), a nie tylko od zmienianego elementu.

Dodanie takich operacji może sprawić, że:

1. Prosta, lokalna struktura zależności w tabeli programowania dynamicznego zostanie zniszczona. Komórka $Lev(i,j)$ mogłaby potencjalnie zależeć od dużej liczby, a nawet od wszystkich poprzednich komórek w tabeli, co zwiększyłoby czas obliczeń dla każdej komórki z $O(1)$ (stały czas, sprawdzamy 3-4 sąsiadów) do $O(i \cdot j)$ lub nawet $O(mn)$. Całkowita złożoność algorytmu mogłaby wzrosnąć z $O(mn)$ do $O((mn)^2)$ lub więcej.

2. Problem przestanie spełniać warunki pozwalające na efektywne rozwiązanie za pomocą "klasycznego" programowania dynamicznego w wielomianowym czasie o niskim stopniu. W niektórych przypadkach problem obliczenia odległości edycyjnej z rozszerzonymi zestawami operacji (np. obejmującymi złożone przestawienia czy duplikacje) może stać się **problemem NP-trudnym**.

Podsumowując, głównym niebezpieczeństwem obliczeniowym przy rozszerzaniu zestawu operacji edycyjnych jest ryzyko naruszenia lokalnej struktury zależności, która gwarantuje efektywność standardowych algorytmów programowania dynamicznego. Operacje nielokalne lub bardziej złożone mogą prowadzić do wykładniczego wzrostu złożoności obliczeniowej lub sprawić, że problem stanie się nierozwiązywalny w praktyce dla dużych sekwencji. Projektowanie algorytmów dla takich rozszerzonych odległości edycyjnych wymaga często zupełnie innych technik i jest znacznie bardziej wymagające.