Spis rzeczy

2 Podobieństwo dwóch sekwencji **25**
2.2 Lokalne uliniowienie 25
2.3 Przerwy w uliniowieniach 27
2.3.1 Dowolna wartość kary za przerwę 29
2.3.2 Afiniczna funkcja kary za przerwę 30
2.4 Porównywanie podobnych sekwencji 31

2 Podobieństwo dwóch sekwencji

2.2 Lokalne uliniowienie

W wielu zastosowaniach w biologii mamy do czynienia z sytuacją, gdy sekwencje S_1 i S_2 nie są podobne ale interesuje nas znalezienie par fragmentów (po jenym z S_1 i S_2) wykazujących duże podobieństwo. Chcemy znaleźć fragmenty sekwencji S_1 i S_2 tak, aby podobieństwo pomiędzy nimi było maksymalne, spośród wszystkich innych par fragmentów. Problem poszukiwania takich fragmentów nazywa się problemem lokalnego uliniowienia.

Typowe zastosowanie globalnego uliniowienia to porównywanie białek (jako ciągów aminokwasów) z tej samej rodziny białek (np. globiny), chociaż niekoniecznie pochodzących z organizmów tego samego gatunku. Lokalne uliniowienie jest bardziej znaczące dla porównywania białek z różnych rodzin, lub porównywania dużych fragmentów DNA pochodzących z różnych organizmów. Lokalne uliniowienia, w przypadku białek, pozwala odkryć powtarzające się ważne fragmenty (motywy, domeny) mające istotny wpływ na funkcjonalność białka.[1] Biolodzy uważają, że badanie lokalnego uliniowienia często daje biologicznie bardziej znaczące wyniki niż badanie globalnego uliniowienia.

Załóżmy, że mamy pewną ustaloną funkcję podobieństwa $s : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \longrightarrow \mathbb{R}$. Niech $S_1, S_2 \in \Sigma^*$. *Lokalne uliniowienie* dla S_1, S_2 jest to każde globalne uliniowienia dla pewnych podsłów słów S_1, S_2.

[1] Fragmenty te są często poprzedzielane długimi obszarami nie wykazującymi żadnego podobieństwa, których role biologiczna nie jest tak istotna, np. fragmenty zwane pętlami, łączące ze sobą domeny.

Przykład 2.2.1 Niech $S_1 = pqr\underline{axabcs}tvq$ oraz $S_2 = xy\underline{axbacs}ll$. Podkreślone fragmenty oznaczają wybrane podsłowa. wówczas
axab–cs
ax–bacs
jest lokalnym uliniowieniem dla S_1, S_2. □

Optymalne lokalne uliniowienie dla słów S_1, S_2, to taka para podsłów: T_1 w S_1 oraz T_2 w S_2, że $sim(T_1, T_2)$ jest wartością maksymalną spośród wszystkich wartości $sim(T_1', T_2')$, gdzie T_1' przebiega wszystkie podsłowa słowa S_1, T_2' przebiega wszystkie podsłowa słowa S_2.

Będziemy się zajmować problemem znajdowania optymalnego lokalnego uliniowienia (oraz wyznaczania jego wartości).

Zauważmy, że dla każdego słowa S, liczba podsłów w S jest $O(|S|^2)$. Zatem liczba możliwych par podsłów dla S_1 i S_2 jest $O(|S_1|^2|S_2|^2)$. Zatem, na pierwszy rzut oka może się wydawać, że problem lokalnego uliniowienia jest algorytmicznie trudniejszy od problemu globalnego uliniowienia. Zauważmy też, że ponieważ zawsze możemy wziąć $T_1 = T_2 = \varepsilon$, to wartść optymalnego uliniowienia jest liczbą nieujemną. Po trzecie zauważmy, że o ile optymalne globalne uliniowienia można wyrazić używając pojęcia odległości edycyjnej (zamiast podobieństwa), to dla optymalnego lokalnego uliniowienia trudniej jst podać taką definicje, opierając się na pojęciu odległości (bo zawsze wybierając równe podsłowa, np. ε, otrzymujemy odległość 0).

Zastosujemy metodę dynamicznego programowania, która pozwoli szybko obliczać (i konstruować) optymalne lokalne uliniowienia. Oczywiście potrzebne są modyfikacje metody dla globalnego uliniowienia. Niech $|S_1| = m$, $|S_2| = n$. Dla $0 \leq i \leq m$ oraz $0 \leq j \leq n$, niech $v(i, j)$ będzie wartością maksymalnego globalnego uliniowienia pomiędzy pewnym sufiksem $S_1[1..i]$ oraz pewnym sufiksem $S_2[1..j]$. Oczywiście mamy $v(i, 0) = v(0, j) = 0$, dla $i \leq m$, $j \leq n$ (bo spacje wnoszą wartości ujemne).

Twierdzenie 2.2.1 *Wartość optymalnego lokalnego uliniowienia dla S_1 i S_2 jest równa*

$$max\{v(i, j) \mid 0 \leq i \leq m,\ 0 \leq j \leq n\},$$

gdzie $v(i, j)$ jest zdefiniownane wyżej.

Dowód: Niech v^* będzie wartością optymalnego lokalnego uliniowienia dla S_1, S_2. Oczywiście $v^* \geq v(i, j)$ zachodzi dla wszystkich i, j (bo sufiksy w $S_1[1..i]$ i w $S_2[1..j]$ wyznaczają podsłowa w S_1 i S_2). Z drugiej strony, jeśli weźmiemy optymalne lokalne uliniowienie T_1, T_2 dla słów S_1 i S_2, to T_1 można

przedstawić jako sufiks $S_1[1..i]$ (dla pewnego i) oraz podobnie dla T_2. To dowodzi równości

$$v^* = max\{v(i,j) \mid i \leq m,\ j \leq n\}.$$

■

Poniższe twierdzenie daje sposób szybkiego liczenia $v(i,j)$.

Twierdzenie 2.2.2 *Dla* $0 < i \leq m$ *oraz* $0 < j \leq n$ *mamy*

$$v(i,j) = max[0,\ v(i-1,j-1)+s(S_1(i),S_2(j)),\ v(i-1,j) + s(S_1(i),-),\\ v(i,j-1) + s(-,S_2(j))].$$

Dowód: Jest podobny do dowodu Twierdzenia 2.1.1. ■

Twierdzenie 2.2.3 *Metoda dynamicznego programowania oparta na Twierdzeniach 2.2.1 oraz 2.2.2 wyznacza wartość optymalnego lokalnego uliniowienia dla słów* S_1, S_2 *w czasie* $O(|S_1| \cdot |S_2|)$ *oraz w pamięci* $O(|S_1| \cdot |S_2|)$.

Dowód: Wynika natychmiast z Twierdzeń 2.2.1 oraz 2.2.2. ■

Zadanie 2.2.1 Dopracować szczegóły pozwalające odtworzyć optymalne lokalne uliniowienie dla dowolnej pary słów S_1, S_2.

Zadanie 2.2.2 Czy lokalne uliniowienie z Przykładu 2.2.1 jest optymalne dla funkcji s z Przykladu 2.1.1?

Zadanie 2.2.3 Znaleźć optymalne lokalne uliniowienie dla słów $S_1 = pawheae$ oraz $S_2 = heagawghee$, przy funkcji podobieństwa z Przykładu 2.1.1.

2.3 Przerwy w uliniowieniach

Przerwą w słowie $S \in (\Sigma \cup \{-\})^*$ nazwiemy każde niepuste podsłowo $T \in \{-\}^*$ słowa S, o maksymalnej długości. Pojęcie przerwy w sekwencji zostało wprowadzone jako wynik istnienia pewnych procesów biologicznych prowadzących do wstawień/usunięć dłuższych fragmentów (DNA):

- Zwykłe mutacje często mają efekt wstawienia bądź usunięcia dłuższego fragmentu.
- Translokacje DNA pomiędzy chromosomami.
- "Poślizg" DNA (DNA slippage) w trakcie replikacji (pewne fragmenty są powtarzane w kopii, gdy mechanizm replikacyjny zgubi aktualną pozycje replikacji na matrycy).
- Tzw. skaczące geny powodują wstawienia DNA.
- Wstawienia DNA mogą być też wynikiem działania retrowirusów.

Tak więc, zaistnienie przerwy (lub wstawienia dłuższego fragmentu) w wyniku jednostkowego procesu powinno być inaczej mierzone dla obliczania podobieństwa niż zwykłe dodawanie kar za każdą spację z osobna.

Pojęcie przerwy znajduje też oczywiste zastosowania gdy chcemy porównać genomowe DNA a komplementarnym DNA (powstałym z genomowego DNA przez usunięcie intronów). W takiej sytuacji chcemy mieć małe kary za przerwy, dość duże kary za niedopasowane symbole i dodatnie wartości za pasujące symbole.

Ogólne sformułowanie zadania wygląda następująco. Niech $w : \mathbb{N}^+ \longrightarrow \mathbb{R}$ będzie funkcją kary za przerwy,[2] tzn. $w(k)$ jest karą za przerwę długości k. Niech $s : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \longrightarrow \mathbb{R}$ będzie funkcją podobieństwa. Ponieważ spacje teraz stanowią część przerw, to oczywiście przyjmujemy, że $s(x,-) = s(-,x) = 0$. Niech $S_1, S_2 \in \Sigma^*$. Uliniowienie $(S_1^\#, S_2^\#)$ dla (S_1, S_2) definiujemy tak samo jak w Sekcji 2.1. Inaczej natomiast liczymy wartość podobieństwa dla takiego uliniowienia. Jest to liczba

$$\sum_{i=1}^{n} s(S_1^\#(i), S_2^\#(i)) + \sum_{j=1}^{m} w(k_j),$$

gdzie $n = |S_1^\#| = |S_2^\#|$, m jest liczbą przerw w $S_1^\#$ i w $S_2^\#$ oraz j-ta przerwa ma długość k_j.

Przykład 2.3.1 Przyjmijmy $s(x,x) = 1$ oraz $s(x,y) = -1$ dla $x, y \in \Sigma$, $x \neq y$. Natomiast funkcja kary za przerwy jest stała i przyjmuje wartość $w(k) = -5$, dla $k \in \mathbb{N}^+$. Wówczas uliniowienie
qq–––abab–rst
qsrrra––brr—
ma wartość podobieństwa -12 (bo mamy cztery przerwy w powyższym uliniowieniu.) □

[2]Przez $\mathbb{N}^+$ oznaczamy zbiór liczb naturalnych dodatnich.

2.3.1 Dowolna wartość kary za przerwę

Najpierw zajmiemy się algorytmem wyznaczania optymalnego uliniowienia, gdy nie mamy żadnych dodatkowych założeń o funkcji w. Zauważmy, że w ogólności mamy trzy rodzaje uliniowień $S_1^{\#}, S_2^{\#}$ dla słów S_1, S_2:

1. $S_1^{\#}$ kończy się przerwą.
2. $S_2^{\#}$ kończy się przerwą.
3. $S_1^{\#}$ i $S_2^{\#}$ kończą się literą.

Uliniowienie z Przykładu 2.3.1 jest rodzaju 2.

Niech $m = |S_1|$ oraz $n = |S_2$. Niech $0 \leq i \leq m$, $0 \leq j \leq n$. Wprowadzimy następujące oznaczenia:

- $E(i,j)$ oznacza maksymalną wartością uliniowienia rodzaju 1 dla $S_1[1..i]$ oraz $S_2[1..j]$;
- $F(i,j)$ oznacza maksymalną wartością uliniowienia rodzaju 2 dla $S_1[1..i]$ oraz $S_2[1..j]$;
- $G(i,j)$ oznacza maksymalną wartością uliniowienia rodzaju 3 dla $S_1[1..i]$ oraz $S_2[1..j]$;
- $V(i,j)$ oznacza wartość optymalnego lokalnego uliniowienia dla $S_1[1..i]$ oraz $S_2[1..j]$.

Rekurencyjne zależności pomiędzy powyższymi wartościami opisuje następujące twierdzenie.

Twierdzenie 2.3.1 *Niech* $0 < i \leq m$ *oraz* $0 < j \leq n$. *Wówczas*

- $V(i,j) = max[E(i,j), F(i,j), G(i,j)]$;
- $E(i,j) = \max\limits_{0 \leq k \leq j-1}[max[F(i,k), G(i,k)] + w(j-k)]$;
- $F(i,j) = \max\limits_{0 \leq k \leq i-1}[max[E(k,j), G(k,j)] + w(i-k)]$;
- $G(i,j) = V(i-1, j-1) + s(S_1(i), S_2(j))$.

Dowód: Powód, dla którego w drugim równaniu stoi $max[F(i,k), G(i,k)]$ zamiast $V(i,k)$ jest taki, że rozważamy maksymalne uliniowienia pomiedzy $S_1[1..i]$ oraz $S_2[1..k]$ nie będące rodzaju 1 (aby nie dzielić przerwy na dwie

części). ∎

Mamy też $G(0,0) = 0$ oraz następujące wartości dla "brzegowych" argumentów: $V(i,0) = V(0,i) = E(i,0) = F(0,i) = w(i)$, dla $i > 0$.

Twierdzenie 2.3.2 *Jeśli $|S_1| = m$ oraz $|S_2| = n$, to wartość optymalnego uliniowienia dla S_1, S_2 (z dowolną funkcją kary za przerwy) można obliczyć w czasie $O(mn^2 + m^2n)$.*

Dowód: Aby wyznaczyć całą j-tą kolumnę w F musimy odwiedzić $\sum_{i=1}^{m} i = O(m^2)$ pozycji w E oraz G. Zatem do wyznaczenia całej tablicy F potrzebujemy $O(m^2n)$ kroków. Podobnie, aby wyznaczyć cały i-ty wiersz w E musimy odwiedzić $\sum_{j=1}^{n} i = O(n^2)$ pozycji w F oraz G. Zatem do wyznaczenia całej tablicy E potrzebujemy $O(mn^2)$ kroków. Łącznie, dostajemy wynik w czasie $O(mn^2 + m^2n)$. ∎

2.3.2 Afiniczna funkcja kary za przerwę

Funkcja kary $w : \mathbb{N}^+ \longrightarrow \mathbb{R}$ jest *afiniczna*,[3] gdy istnieją takie stałe A, B, że $w(k) = A + kB$, dla $k \in \mathbb{N}^+$. Oczywiście jeśli każdą spację karzemy tak samo, to odpowiada to funkcji kary $w(k) = kB$. Inny szczególny i ważny przypadek afinicznej funkcji to tzw. stała kara: $w(k) = A$, dla $k \in \mathbb{N}^+$.

Pokażemy, że uogólnienie funkcji kary za przerwy $w(k) = kB$ (co miało miejsce w Sekcji 2.2) na ogólny przypadek $w(k) = A + kB$ nie prowadzi do zwiększenia złożoności czasowej. Zauważmy, że przyjęcie afinicznej funkcji kary oznacza, że pierwsza spacja w przerwie (tzw. otwarcie przerwy kosztuje $A + B$, a każda następna spacja kosztuje B. Obserwacja ta będzie pomocna przy wyznaczaniu postaci równań pozwalających wyznaczać wartość optymalnego uliniowienia. Poniżej używamy oznaczeń z Sekcji 2.3.1.

Twierdzenie 2.3.3 *Niech $0 < i \leq m$ oraz $0 < j \leq N$. Wówczas*

- $V(i,j) = max[E(i,j), F(i,j), G(i,j)]$;
- $G(i,j) = V(i-1, j-1) + s(S_1(i), S_2(j))$;
- $E(i,j) = max[E(i,j-1), V(i,j-1) + A] + B$;

[3] Jest to nazwa zwyczajowo przyjęta w biologii. W matematyce takie funkcje naywa się liniowymi.

- $F(i,j) = max[F(i-1,j), V(i-1,j) + A] + B$.

Dowód: Równanie na $E(i,j)$ ujmuje dwie możliwości: kiedy przerwa kończąca (rodzaj 1 uliniowienia) zaczyna się w miejscu i (wówczas wartość wynosi $V(i,j-1) + w(1) = V(i,j-1) + A + B$); druga możliwość to kiedy przerwa ta jest kontynuowana w miejscu i (tzn. zaczęła się wcześniej, jej wartość wówczas wynosi $E(i,j-1) + B$). Podobne uwagi stosują się do równania na $F(i,j)$. ∎

Poniższe twierdzenie mówi, że koszt obliczania uliniowienia z afiniczną funkcją kary jest stosunkowo niski. Z tego względu takie funkcje kary są powszechnie stosowane w biologii.

Twierdzenie 2.3.4 *Jeśli $|S_1| = m$ oraz $|S_2| = n$, to wartość optymalnego uliniowienia dla S_1, S_2, z afiniczną funkcją kary za przerwy, można obliczyć w czasie $O(mn)$.*

Dowód: Wynika natychmiast z Twierdzenia 2.3.3. ∎

2.4 Porównywanie podobnych sekwencji

Teraz zajmiemy się problemem znalezienia globalnego podobieństwa dwóch słów, mając dodatkową informację, że słowa te nie różnią się zbytnio. Taki problem może pojawić się w pewnych zastosowaniach biologicznych:

- Umiejscowianie genów, których mutacje są odpowiedzialne za pewne choroby o podłożu genetycznym (porównuje się odpowiadające sobie geny osobników chorych i zdrowych — wówczas porównywane sekwencje są bardzo podobne).

- W procesie sekwencjonowania dużych genomów zwykle, dla poprawienia pewności prawidłowości odczyatnej sekwencji, zleca się sekwencjonownaie tego samego chromosomu kilku różnym laboratoriom. Następnie porównuje się wynikowe sekwencje, które oczywiście są bardzo podobne.

- Tworzenie nieredundantnych baz danych dla białek pochodzących od bakterii. Redukcja rozmiaru takiej bazy danych ma oczywisty wpływ na poprawę szybkości wyszukiwania, bądź też na lepsze oszacownaie statystycznej ważności znalezionych wyników poszukiwań. usuwanie

redundancji polega na znajdowaniu bardzo podobnych sekwecji i łączeniu ich w klasy podobieństw, lub nawet usuwaniu z bazy danych.

Będziemy sie zajmować globalnym uliniowieniem. Przyjmijmy, że słowa S_1, S_2 mają tę samą długość n. Przyjmijmy, że funkcja podobieństwa s ma tę cechę, że kara za spację nie jest dużo większa niż kara za niedopasowanie liter. Wówczas jeśli droga wyznaczająca optymalne uliniowienie dla S_1, S_2 leży w pobliżu przekątnej tablicy V wyznaczonej przez algorytm optymalnego uliniowienia, to możemy uznać, że S_1 i S_2 są podobne. Zajmiemy się następującym problemem: dane są słowa S_1, S_2 o tej samej długości oraz $k > 0$. Szukamy optymalnego globalnego uliniowienia (oraz wartości ich podobieństwa), przy dodatkowym założeniu, że droga w V wyznaczająca to uliniowienie nie wykracza poza pas $[-k, k+1]$ wokół przekątnej tablicy V.[4] Oczywiście w zadaniu tym chodzi o to by zminimalizować czas. Poniższy algorytm rozwiązuje to zadanie w czasie $O(kn)$, co jest dużym zyskiem wobec ogólnej metody działającej w czasie $O(n^2)$ (oczywiście, gdy k jest dużo mniejsze od n).

Działanie poniższego algorytmu polega na tym, że wypełnia on, stosując te same reguły co ogólny algorytm programowania dynamicznego, jedynie pas $[-k, k]$ wokół przekątnej. Oczywiście nie bierze on pod uwagę wartości, które w V zbajdują się poza pasem $[-k, k]$. Jeśli optymalne uliniowienie znajduje się w tym pasie, to algorytm go znajdzie — wystarczy zapamiętywać w tablicy strzałki pokazujące skąd wzięła się wartość w aktualnej pozycji w tablicy a. Ponieważ rozmiar tego pasa jest $O(kn)$, to taki jest czas działania tego algorytmu. Algorytm ten można bez trudu uogólnić dla słów S_1, S_2 o długościach m, n, pod warunkiem, że $|m - n| \leq k$ (bo tylko wtedy (m, n) należy do pasa $[-k, k]$).

Zadanie 2.4.1 Znaleźć algorytm, który dla danego $k \in \mathbb{N}^+$ oraz słów S_1, S_2, takich że $||S_1| - |S_2|| \leq k$, znajduje optymalne uliniowienie (a nie tylko wartość jego podobieństwa) dla S_1, S_2 leżące w pasie $[-k, k]$, w czasie $O(k \cdot max[m, n])$.

[4] Czyli (i, j) należy do takiej drogi, gdy $|i - j| \leq k$.

Wejście: słowa $S_1, S_2 \in \Sigma^*$, $(|S_1| = |S_2|)$, $k \in \mathbb{N}^+$
Wynik: wartość najlepszego globalnego uliniowienia dla S_1, S_2, leżącego w pasie $[-k, k]$ wokół głównej przekątnej w V.

$n := |S_1|$;
$a(0,0) := 0$;
for $i = 1$ **to** k **do**
    $a(i,0) := a(i-1,0) + s(S_1(i), -)$;
for $j = 1$ **to** k **do**
    $a(0,j) := a(0,j-1) + s(-, S_2(j))$;
for $i = 0$ **to** k **do**
    for $d = -k$ **to** k **do**
        if $1 \leq i + d \leq n$ **then**
            $\{j := i + d$;
            $a(i,j) := a(i-1,j-1) + s(S_1(i), S_2(j))$;
            if $|i - j - 1| \leq k$ **then** $//(i-1,j)$ należy do pasa $[-k,k]$
                $a(i,j) := max[a(i,j), a(i-1,j) + s(S_1(i), -)]$;
            if $|i - j + 1| \leq k$ **then** $//(i,j-1)$ należy do pasa $[-k,k]$
                $a(i,j) := max[a(i,j), a(i,j-1) + s(-, S_2(j))]$;
            $\}$
return $a(n,n)$;

Algorytm 2.4.1: Obliczanie globalnego podobieństwa dla podobnych słów.

Zauważmy, że jeśli optymalne globalne uliniowienie dla S_1, S_2 nie leży w pasie $[-k, k]$, to Algorytm 2.4.1 znajdzie uliniowienie (w w/w pasie), którego wartość podobieństwa jest mniejsza od $sim(S_1, S_2)$. Jeśli nie znamy wartości k, to możemy iterować Algorytm 2.4.1, aproksymując wartość optymalnego uliniowienia: zaczynamy dla $k = 1$, niech a_1 będzie wartością zwróconą przez algorytm. Następnie podwajamy k i ponownie wykonujemy algorytm, itd. Niech a_p będzie wartością zwracaną przez algorytm dla $k = p$. Oczywiście mamy

$$a_1 \leq a_2 \leq a_4 \leq \ldots .$$

Jeśli dla pewnego p mamy $a_p < a_{2p}$, to na pewno a_p nie jest wartością optymalną. Jeśli jednak $a_p = a_{2p}$, to nie możemy stwierdzić czy a_p jest wartością optymalną, o czym świadczy poniższy przykład.[5]

[5]Wbrew temu co twierdzą na str. 68 autorzy podręcznika "Introduction to Computational Molecular Biology" (Meidanes, Setubal).

Przykład 2.4.1 Niech $S_1 = aaacc$, $S_2 = ccttt$. Jako funkcję podobieństwa weźmiemy $s(x,x) = 5$, $s(x,y) = -1$ (dla $x \neq y$) oraz $s(x,-) = -2$, gdzie $x, y \in \{a, c, t\}$.

Wartość uliniowienia S_1, S_2 bez spacji jest równa -5 (jest to sytuacja odpowiadająca $k = 0$). Jeśli wstawimy do S_1 i S_2 po jednej spacji (czyli $k = 1$), liczba porównań ze spacją wyniesie 2, natomiast przy każdym wyborze pozycji na spacje nie mamy żadnej pary odpowiadających sobie równych symboli. Zatem wartość takiego uliniowienia wyniesie -8. Zatem algorytm dla $k = 1$ zwróci uliniowienie bez spacji Tak wiec, $a_1 = -5$.

Jeśli wstawimy do S_1 i S_2 po dwie spacje ($k = 2$), to liczba porównań ze spacją wyniesie 4, natomist liczba porównań symbolu z symbolem jest równa 3. Mamy dwie możliwości: możemy tak wstawić spacje, że będziemy mieli dokładnie jedną parę identycznych symboli na odpowiadających sobie pozycjach, bądź też takiej pary wcale nie będzie. Tak wiec optymalne uliniowienie dla wstawionych dwóch spacji wyniesie $-8 + 5 - 2 = -5$. Zatem $a_2 = -5$ Czyli dla $k = 2$ mamy dwa uliniowienia realizujące wartość -5 i leżące w pasie $[-2, 2]$.

Natomiast jeśli wstawimy do S_1 i S_2 po trzy spacje, to jak łatwo się przekonać, możemy dostać wynik $-12+10 = -2$. Jest to optymalna wartość. □

Zadanie 2.4.2 * Czy istnieje algorytm obliczający dla danych słów S_1, S_2 o takiej samej długości najmniejsze k, takie że istnieje optymalne globalne uliniowienie dla S_1, S_2 leżące w pasie $[-k, k]$. Algorytm ten ma działać w czasie $O(k \cdot |S_1|)$.

Następujące zadanie pokazuje, że odległość edycyjną możemy obliczać szybciej niż by to wynikało z ogólnego algorytmu opartego na dynamicznym programowaniu (por. Zadanie 2.1.5).

Zadanie 2.4.3 Niech $|S_1| = |S_2| = n$. Metoda dynamicznego programowania obliczająca $\delta(S_1, S_2)$ działa w czasie $O(n^2)$. Znaleźć algorytm działający w czasie $O(n \cdot \delta(S_1, S_2))$, który oblicza $\delta(S_1, S_2)$, dla dowolnych słów S_1, S_2. Tak więc, jeśli słowa S_1 i S_2 są podobne (w sensie małej odległości edycyjnej, np. $\delta(S_1, S_2) \leq k$ dla pewnej małej stałej k), to algorytm będzie w stanie obliczyć $\delta(S_1, S_2)$ w czasie $O(kn)$. Algorytm będzie również w stanie stwierdzić poprawnie w czasie $O(kn)$, że $\delta(S_1, S_2) > k$, jeśli nierówność taka naprawdę zachodzi.