Karol Bonenberg

Lokalne dopasowanie wielu sekwencji w biologii obliczeniowej

Spis treści

1. Wprowadzenie

1.1. Dopasowywanie sekwencji

Dopasowywanie sekwencji to w bioinformatyce sposób porównywania sekwencji pierwszorzędowej DNA, RNA bądź białek w celu identyfikacji regionów podobnych, które mogą być wynikiem funkcjonalnych, strukturalnych bądź ewolucyjnych związków pomiędzy sekwencjami.[1] Jeśli dwie porownywane sekwencje pochodzą od wspólnego przodka niezgodności mogą być interpretowane jako mutacje punktowe, natomiast przerwy jako mutacje insercji bądź delecji w jednej z sekwencji, poziom podobieństwa sekwencji natomiast świadczy o tym, jak bardzo konserwatywne pod względem mutacji są porównywane sekwencje bądź domeny. Małe zmiany w danym rejonie mogą świadczyć o wysokiej wadze danej domeny dla zachowania funkcji białka. Porównywanie sekwencji może być używane również poza biologią - algorytmy stosowane do rozwiązywania powyższych problemów znajdują zastosowanie także w lingwistyce oraz analizie języka naturalnego.

1.2. Lokalne dopasowanie

Obliczeniowa natura problemu wymusiła rozróżnienie dwóch rodzajów dopasowań : lokalnego i globalnego. Poszukiwanie globalnego podobieństwa ma na celu dopasowanie sekwencji na całej długości w jak największym stopniu do zbioru sekwencji wejściowych. Najbardziej znaną metodą rozwiązywania tego problemu jest algorytm Needlemana-Wunscha, opierający się na programowaniu dynamicznym. Poszukiwanie lokalnego podobieństwa sprowadza się natomiast do wyszukania pewnych regionów podobieństwa w sekwencjach niekoniecznie do siebie podobnych czy też mających podobną długość. W przypadku tego problemu, najczęściej stosowanym algorytmem jest algorytm Smitha-Watermana, również wykorzystujący metodę programowania dynamicznego.

[1] „Bioinformatics, Sequence and Genome Analysis", *David W. Mount*, 2001 Cold Spring Harbor Laboratory Press

2. Opis projektu

Zadaniem jest opracować i zaimplementować heurystykę działającą w czasie wielomianowym dla problemu znajdywania lokalnego dopasowania wielu sekwencji.[1] Wymieniony wcześniej algorytm Smitha-Watermana porównuje dwie sekwencje, znajdując dla nich najlepsze lokalne podobieństwo. W naszym projekcie rozwiązujuemy problem dla wielu sekwencji, zatem konieczne jest stworzenie nowego rozwiązania, bedącego heurystyką.

Nasz algorytm zaimplementowaliśmy w Javie, wykorzystując środowisko Netbeans oraz pakiet JAligner. [2] Zgodnie z opisem zadania, dane wejściowe to zbiór sekwencji nukleodytowych, na wyjściu otrzymujemy lokalne dopasowanie wszystkich sekwencji. Kryterium oceny rozwiązania to maksymalizacja wartości dopasowania przy zastosowaniu punktacji +1 za zgodność pary nukleodytów, -1 za niezgodność, i -1 za każdą wprowadzoną spację.

[1] Strona laboratorium zastosowań informatyki w biologii obliczeniowej dr hab. Marty Kasprzak *http://www.cs.put.poznan.pl/mkasprzak/bio/lab.html*

[2] Pakiet JAligner jest implementacją algorytmu Smitha-Watermana zoptymalizowanego pod względem szybkości działania (Gotoh's improvement) na licencji BSD oraz GPL textithttp://jaligner.sourceforge.net/

3. Opis algorytmu

3.1. Idea działania

Zaprojektowany przez nas algorytm przybliżony w ogólności opiera się na algorytmie Smith'a-Watermana, a jego działanie można podzielić na 4 zasadnicze etapy.W kroku pierwszym dokonywane jest dopasowanie za pomocą SM wszystkich sekwencji (każda z każdą). W rezultacie otrzymujemy macierz o wymiarach n * n zawierającą wyniki dopasowań.
W drugim etapie realizowane jest sumowanie wartości w poszczególnych wierszach macierzy gdzie suma m-tego wiersza reprezentuje jakość dopasowania m-tego ciągu z wszystkimi pozostałymi, przy czym zapamiętywana jest również reprezentacja graficzna dopasowania (dane ciągi sx i sy, które by dopasowywane). Informacje uzyskane z sumowania determinują, które z zapisanych dopasowań dwóch sekwencji zostaną dalej wykorzystane. Kluczowy jest również fakt, że sumy te wyznaczają też dalszą kolejność, w jakiej będą przetwarzane wybrane dopasowania.
Trzeci krok obejmuje przygotowanie zbioru wybranych sekwencji do ostatniego etapu, konstrukcji rozwiązania. Polega on na odpowiednim posortowaniu danych na podstawie sum z etapu drugiego. Realizację sortowania ilustruje poniższy przykład:

```
Sortowanie w oparciu o informacje o sumach

Należy pamiętać, że macierz na podstawie, której wyliczane były sumy przechowuje również
 informacje o samych dopasowaniach. Każda komórka macierzy reprezentuje,
 więc trójkę ( ( s1_dopasowane_do_s2, s2_dopasowane_do_s1 ), wynik_dopasowania. )

Ciąg - suma
S1 - 300
S4 - 240
S2 - 235
S6 - 204
S3 - 198
S5 - 130

DanePosortowane[0] = null;
...
DanePosortowane[5] = null;

Pierwszym rozpatrywanym elementem jest S1. Jednakże dane o dopasowaniach określane są
```

```
 przez dwa ciągi Sx i Sy. Należy, zatem przejść do następnego "najlepiej pasującego
 do pozostałych" elementu, w tym przypadku S4. Określana zostaje para (s1_z_s4,s4_z_s1).

DanePosortowane[0] = s1_z_s4;
DanePosortowane[1] = s4_z_s1;
DanePosortowane[2] = null;
...
DanePosortowane[5] = null;

Przechodzimy do kolejnego "najlepszego" elementu, czyli S2. W tym miejscu następuje
 sprawdzenie wartości "w tył" w przeciwieństwie do kroku poprzedniego gdzie sprawdzany
 był następny element. Określana jest para (s4_z_s2,s2_z_s4) i pobierana jest wartość s2_z_s4.

...
DanePosortowane[2] = s2_z_s4;
...

Tutaj następuje koniec etapu i procedura jest powtarzana, tzn. Wybieramy dwa kolejne
 "najlepsze" (S6 i S3), uzyskujemy s6_z_s3,s3_z_s6. Wybieramy następny najlepszy i
 porównujemy z poprzednim, co da s3_z_s5,s5_z_s3.
 Uzyskamy kolejną trójkę posortowanych danych

...
DanePosortowane[3] = s6_z_s3
DanePosortowane[4] = s3_z_s6
DanePosortowane[5] = s5_z_s3

Koniec sortowania
```

W kroku czwartym realizowana jest konstrukcja rozwiązania. Przygotowywana jest macierz wynikowa o wymiarach

```
liczba_elementow_wejsciowych   x   2*max_długosc_ciagu_wejsciowego+1
```

Dokładnie w połowie pierwszego wiersza osadzony jest pierwszy element z DanePosortowane. Następnie kolejne elementy są "przyklejane" w najkorzystniejszym miejscu. Poszukiwanie owego miejsca (przesunięcia przyklejanego ciągu względem początku wiersza) oraz przyklejanie kolejnych "klocków" realizują dwie funkcje rekurencyjne.

```
wybierz(String s1,String s2,int startOffset,int currOffset,int maxOffset,int maxValue)
```

Pierwsza z nich sprawdza, jaką wartość uzyskamy z dopasowania klocka s2 do s1, jeżeli s1 zaczyna się od startOffset, a sprawdzane bieżące przesunięcie s2 to currOffset.
Należy dodać, że currOffset reprezentuje względne przesunięcie początku s2 w stosunku do końca s1 osadzonego w macierzy wynikowej (startOffset+s1.length()). Wartość trueOffset reprezentuje przesunięcie początku s2 względem początku wiersza macierzy wynikowej.

Warunek stopu rekursji przedstawia się, zatem następująco

```
trueOffset + s2.length() == startOffset + 1 --> STOP
```

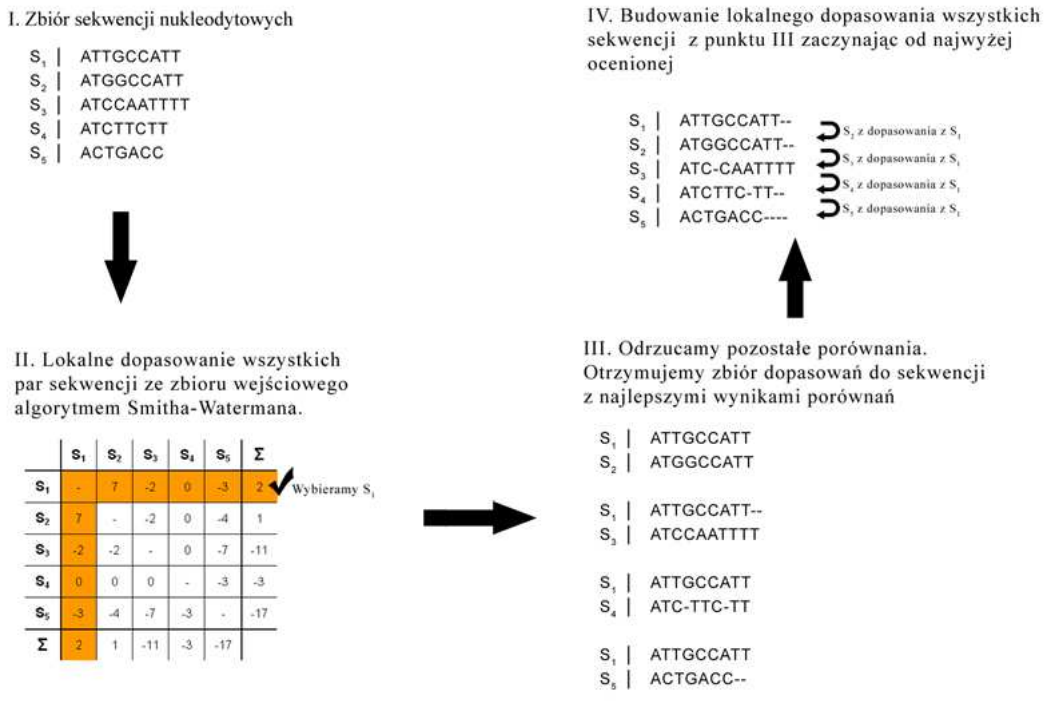
```
rozwiąż(String[] dane,int startOffset,Wyniki wyniki,int level)
```

Druga funkcja realizuje doklejenie kolejnych klocków (sekwencji) do kolejnych wierszy macierzy wynikowej.
Warunek stopu wygląda tak:

```
level == dane.length() - 1 --> STOP
```

Należy zaznaczyć, że poszukiwanie najkorzystniejszego miejsca do przyklejenia k-tego klocka polega na znalezieniu takiego umiejscowienia k-tego klocka względem k-1-szego klocka, aby wynik ich (k i k-1) dopasowania lokalnego był maksymalny. Klockowanie rozpoczynane jest od drugiego elementu z DanePosortowane, gdyż tak jak wspomniano wyżej, pierwszy element jest już osadzony w macierzy wynikowej. Po zakończeniu konstrukcji rozwiązania następuje podliczenie jego wartości liczbowej. Na wyjściu uzyskujemy ową wartość a także graficzną reprezentację dopasowania oraz kilka danych dotyczących długości oraz ilości trafień

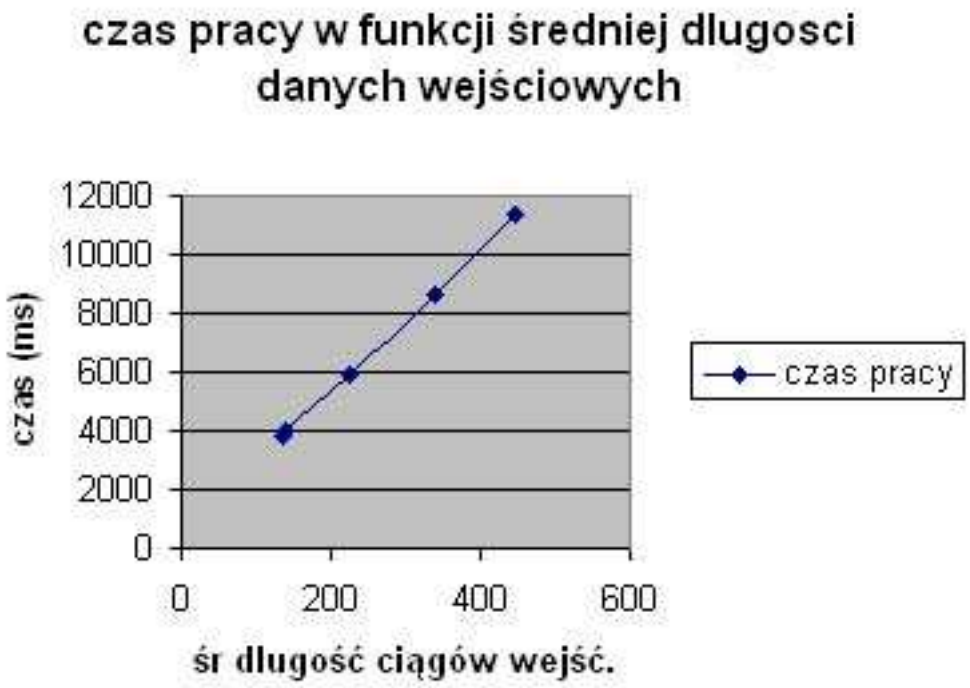
3.2. Schemat

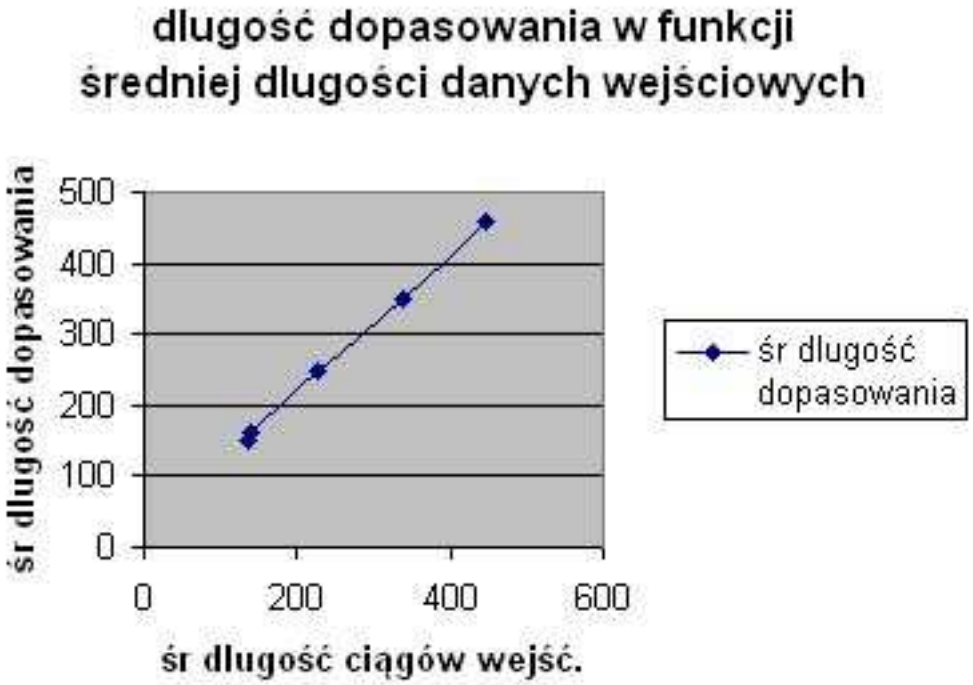


4. Testy

4.1. Test czasowy

W wyniku przeprowadzonych testów udało się wyznaczyć następujące zależności:




dlugość dopasowania w funkcji
średniej dlugości danych wejściowych
śr dlugość dopasowania
500
400
300
200
100
0
0
200
400
600
śr dlugość ciągów wejść.
śr dlugość dopasowania


czas dzialania w funkcji rozmiaru
instancji
czas (ms)
25000
20000
15000
10000
5000
0
0
10
20
rozmiar instancji
czas dzialania

5. Podsumowanie

Algorytmy heurystyczne z założenia nie są optymalne. Warto jednak przeanalizować zaproponowaną przez nas heurezę w kilku aspektach:

- zastosowane pomysły a ich wpływ na jakość rozwiązania
- jakość czasowo-pamięciowa algorytmu
- gdzie można by dokonać ulepszeń

Wnioski płynące z powyższych rozważań najtrafniej ująć w punkty:

- (krok 1) Porównanie zbioru sekwencji wejściowych "każdy z każdym" jest dość czasochłonne, jednak uzyskane informacje są dokładne a nieprzybliżone.
- (krok 2) Można by uzyskać korzyści czasowe w jakiś sposób ograniczając liczbę porównań, np. probabilistyczny wybór komórek par ciągów, które mogą dać dobry wynik. Niestety taka modyfikacja prawdopodobnie pogorszy jakość rozwiązania.
- (krok 3) Ponieważ dokonywane są tylko obliczenia arytmetyczne, jedynie jakość implementacji wpływa na czas realizacji tychże obliczeń a w konsekwencji na czas działania algorytmu.
- (krok 4) Preselekcja odbywająca się w tym kroku ma zdecydowany wpływ na jakość rozwiązania ostatecznego. Zarówno kolejność jak i postać wybieranych ciągów bezpośrednio przekładają się na jakość wyniku końcowego Jest to najlepszy pod względem jakości wyników sposób selekcji jaki udało się nam opracować.
- (krok 5) Jest to najistotniejszy etap algorytmu z punktu widzenia kryteriów zarówno jakościowych jak i czasowo-zasobowych. Tutaj następuje najsilniejsze rozmycie jakościowe rozwiązania, jak również największy narzut pamięciowy związany z przetwarzaniem. Związane jest to z zastosowaniem rekurencji, której głębokość jest bezpośrednio związana z długością ciągów wejściowych. Obliczenia w kroku 4 zajmują znaczną część czasu działania algorytmu. Jeśli chodzi o jakość rozwiązania, zdecydowane polepszenie uzyskałoby w przypadku, gdy najkorzystniejsze miejsce do przyklejenia kolejnego klocka (ciągu) wyznaczane byłoby w oparciu o

konfiguracje wszystkich dotychczas przyklejonych ciągów w macierzy wynikowej, a nie tylko ad hoc, względem poprzedniego przyklejanego ciągu. Takie rozwiązanie obarczone jest niestety ogromnym kosztem pamięciowym, ponieważ na każdym poziomie rekurencji musimy pamiętać i przekazywać całą macierz wynikową. (pamiętajmy, że jest ona definiowana na sztywno i posiada dość znaczne wymiary). Z powyższych względów nie zdecydowaliśmy się na takie podejście. Mimo to napotkaliśmy problemy w postaci ograniczenia dopuszczalnej długości ciągów wejściowych przez rozmiar stosu (dopuszczalną głębokość rekursji).

Osiągi czasowe algorytmu jak również ilość niezbędnej do działania pamięci zależą naturalnie również od jakości implementacji poszczególnych elementów (kroków) algorytmu.