STUDIA INFORMATICA 2012
Volume 33 Number 2A (105)

Jolanta KAWULOK
Politechnika Śląska, Instytut Informatyki

WYSZUKIWANIE PRZYBLIŻONE SEKWENCJI DNA Z UŻYCIEM INDEKSU FM

Streszczenie. Artykuł opisuje nowy algorytm wyszukiwania przybliżonego zadanych fragmentów DNA w długich sekwencjach. Zgodnie z zaproponowanym podejściem poszukiwany fragment jest dzielony na nakładające się słowa, których pozycje w badanej sekwencji są wyznaczane przez użycie indeksu FM. Wykorzystując tak otrzymaną listę pozycji słów w sekwencji, poszukuje się połączeń spełniających założenie o dopuszczalnej maksymalnej liczbie różnic. Stworzony algorytm został poddany walidacji eksperymentalnej, której wyniki przedstawiono w artykule.

Słowa kluczowe: dopasowanie sekwencji, transformata Burrowsa-Wheelera, drzewo falkowe, indeks FM

APPROXIMATE MATCHING OF DNA SEQUENCES USING FM-INDEX

Summary. This paper presents an algorithm for searching fragments of sequences in previously prepared DNA bases. The pattern is divided into words which overlap themselves. Their positions are found using FM-index, and they are used to search connections with each other under the assumption about a permissible maximum number of distinction.

Keywords: pattern matching, Burrows-Wheeler transform, Wavelet Tree, FM-index

1. Wprowadzenie

Metody sekwencjonowania DNA, czyli odczytywania par nukleotydów w cząsteczce, są obecnie na tyle zautomatyzowane, że z dnia na dzień rośnie liczba poznanych sekwencji. Jedną z większych i znanych baz danych przechowującą informacje genetyczne jest GenBank [3]. Obecnie baza ta zawiera około14,8 miliarda par zasad dla samego człowieka i dodatkowo

gromadzi informacje o innych organizmach. Przy tak dużej ilości danych istotne staje się zagadnienie ekstrahowania z nich konkretnych informacji, np. poszukiwania zadanych fragmentów w sekwencjach, analiza podobieństw, różnic czy też fragmentów powtarzających się. Metody dopasowania sekwencji są zatem fundamentalną procedurą prowadzenia badań w biologii i medycynie. Dlatego coraz bardziej są rozwijane algorytmy, które porównują jedną sekwencję do zawartości całej bazy DNA.

1.1. Istniejące algorytmy wyszukiwania dopasowań

Jedną z pierwszych metod porównywania dwóch sekwencji było budowanie tzw. macierzy *Dot-matrix*. Przykładowym programem obecnie wykorzystującym tę metodę jest dotter [18]. Narzędzie to umożliwia wizualizację wyników porównań pomiędzy dwoma sekwencjami. Realizowana jest ona przez interpretację porównywanych sekwencji jako współrzędnych układu dwuwymiarowego, na którym zaznaczane są punkty odpowiadające pozycjom identycznym. Jednak metoda ta nie jest optymalna ze względu na konieczność analizy dużej liczby ścieżek. Lepszym rozwiązaniem jest wykorzystanie programowania dynamicznego wykorzystującego macierze dopasowania, których poszczególne komórki są uzupełniane na podstawie porównywania badanych miejsc w obu sekwencjach z uwzględnieniem poprzednio wyliczonych wartości. Optymalna ścieżka dopasowania jest natomiast wyznaczana przez działanie w odwrotnym kierunku. Dopasowanie może być realizowane globalnie lub lokalnie. Bazowym algorytmem dla dopasowań globalnych jest algorytm Needlemana-Wunscha [14], a dla lokalnych – algorytm Smitha-Watermana [17].

Jak wspomniano wcześniej, rozmiary baz danych zawierających sekwencje DNA stale się zwiększają. Powoduje to, że proste aplikacje, które stosują dynamiczne programowanie, stają się powoli niewystarczające. Konieczne zatem jest zastosowanie metod heurystycznych, które przez wykorzystanie aproksymacji przyspieszają proces poszukiwania dopasowań. Jedną z takich metod jest dzielenie sekwencji na krótsze łańcuchy kolejnych symboli, tzw. słowa [2, 19]. Główna zasada działania polega na założeniu, że dwie sekwencje powinny zawierać przynajmniej jedno wspólne słowo. Programy z grupy BLAST (*Basic Local Alignment Search Tool*) wyszukują słowa za pomocą algorytmu Smitha-Watermana, a następnie przeprowadzają analizę sąsiedztwa znalezionych słów. W programie BLAST słowa nie muszą być dopasowane dokładnie w przeciwieństwie do programu FASTA [1]. Podobnym narzędziem do BLAST-a jest BLAT (*Blast Like Alignment Tool*), który dla analizowanej, długiej sekwencji buduje odpowiedni indeks, wskazujący położenie słów [12]. Pozwala to na zwiększenie szybkości przeszukiwania. Wadą narzędzia jest natomiast niska skuteczność wyszukiwania przybliżonego dla sekwencji krótszych niż 40 par zasad.

Kompresja tekstu przez indeksowanie jest powszechnie stosowana w algorytmach wyszukiwania wzorca w tekście, jednak w większości przypadków nie jest to wyszukiwanie dokładne. Wśród niewielu istniejących metod wyszukiwania przybliżonego, działających na podstawie zaindeksowanych sekwencji, warto wymienić dwa rozwiązania. Välimäki i wsp. [20] ze zbioru *r* sekwencji nukleotydowych poszukują takich grup, których sufiksy lub prefiksy są dopasowane do siebie z ustaloną maksymalną odległością oraz mają zadaną długość. Do poszukiwania wzorca w tekście z ustaloną maksymalną odległością Russo i wsp. [16] tworzą modele błędów, które definiują, w jaki sposób należy podzielić poszukiwaną sekwencję, a także określają maksymalną liczbę błędów w danym fragmencie.

1.2. Zaproponowany algorytm

Przedstawiony w niniejszym artykule algorytm nie nakłada ograniczeń na minimalną długość sekwencji zapytania. Zostało to osiągnięte przez podział sekwencji na nakładające się słowa, które są wyszukiwane w bazie poddanej transformacji Burrowsa-Wheelera i zapisanej za pomocą falkowego drzewa binarnego. Generowanie nakładających się słów zwiększa szanse na znajdowanie blisko położonych od siebie różnic pomiędzy sekwencjami. Wyszukiwanie słów odbywa się na podstawie wyznaczania indeksu FM, któremu poświęcony jest rozdział 2. Na podstawie wszystkich pozycji wyznaczonych dla danego słowa, wyszukiwane są takie połączenia, aby dopasowanie nie miało większej odległości Levenshteina niż ustalona wartość (rozdział 3). W rozdziale 4 przedstawiono przykładowe wyniki, a w rozdziale 5 przedstawiono podsumowanie wraz z nakreśleniem kierunków przyszłych prac.

2. Wstępne przygotowanie sekwencji

Przyspieszenie procesu wyszukiwania zadanego wzorca w sekwencji *T* może zostać zrealizowane przez stworzenie odpowiedniego indeksu. Jednym ze sposobów zapisu tekstu, który został wykorzystany w niniejszym artykule, jest indeks FM (*FM-index*) [7]. Pozwala on odpowiedzieć na pytanie, ile zadanych wzorców jest w tekście, a także wskazać miejsce ich występowania w oryginalnej sekwencji. Indeks ten bazuje na transformacji Burrowsa-Wheelera, opisanej w dalszej części rozdziału.

2.1. Transformata Burrowsa-Wheelera

Transformata Burrowsa-Wheelera (BW) jest algorytmem zaproponowanym w 1994 roku, który przekształca dane wejściowe w celu ułatwienia ich dalszej kompresji [4]. Niech $T[0, n-1]$

będzie sekwencją składającą się ze znaków z alfabetu Σ o długości σ. Na końcu sekwencji dodawany jest dodatkowy znak specjalny nienależący do Σ i mniejszy od pozostałych znaków alfabetu (np. #). Transformacja BW składa się z trzech kroków, zobrazowanych na rys. 1 dla przykładowej sekwencji: (1) utworzenie macierzy M, której wiersze są cyklicznymi przesunięciami sekwencji T; (2) posortowanie tablicy M względem pierwszej kolumny (F); (3) zdefiniowanie ostatniej kolumny macierzy M jako wyniku transformacji – T^{BWT}. Jako że każda kolumna stanowi permutację znaków sekwencji T, przy sortowaniu według alfabetu sekwencji T^{BWT} zostanie wygenerowana pierwsza kolumna. Dla macierzy M można wykazać zależność pomiędzy kolumnami ostatnią a pierwszą. Warto wprowadzić następujące oznaczenia:

- $C[\cdot]$ – oznacza wektor długości σ, taki, że $C[c]$ zawiera liczbę znaków w T, mniejszą od znaku c,
- $OCC(c,q)$ – oznacza liczbę wystąpień znaku c w prefiksie $T^{BWT}[0,q]$.

Pozycja znaku $T^{BWT}[i]$ w pierwszej kolumnie jest wyznaczana za pomocą odwzorowania LF (*last-to-first mapping*):

$$LF(i) = C[c] + OCC(c,i) - 1, \tag{1}$$

gdzie $c=T^{BWT}[i]$. Przykładowo na podstawie rys. 1 $LF(4) = C['G'] + OCC('G',4) - 1 = 7$, co oznacza, że zarówno $T^{BWT}[4]$, jak i $F[LF[4]]=F[7]$ odpowiadają pierwszemu znakowi „G" w sekwencji T.

Macierz M
ATCATGCAG#
TCATGCAG#A
CATGCAG#AT
ATGCAG#ATC
TGCAG#ATCA
GCAG#ATCAT
CAG#ATCATG
AG#ATCATGC
G#ATCATGCA
#ATCATGCAG

→

Posortowana macierz M		
F		T^{BW}
#	ATCATGCA	G
A	G#ATCATG	C
A	TCATGCAG	#
A	TGCAG#AT	C
C	AG#ATCAT	G
C	ATGCAG#A	T
G	#ATCATGC	A
G	CAG#ATCA	T
T	CATGCAG#	A
T	GCAG#ATC	A

Rys. 1. Przykład transformaty Burrowsa-Wheelera dla sekwencji „ATCATGCAG". Macierz po prawej stronie zawiera wiersze posortowane według alfabetu. Wynikiem jest ostatnia kolumna T^{BTW}

Fig. 1. Example of Burrows-Wheeler transform for the sequence "ATCATGCAG". The matrix on the right has the rows sorted in lexicographic order. The output is the last column T^{BWT}

2.2. Kompresja tekstu i wykorzystanie indeksu FM

Głównym celem indeksu FM jest poszukiwanie wstecz występowania zadanego wzorca P w sekwencji T. Odbywa się to przez wyznaczenie przedziału wierszy w macierzy M, których prefiksy są równe poszukiwanemu fragmentowi. Na rys. 1 jest przedstawiony pseudokod al-

gorytmu FM_COUNT, zwracający pierwszy (`First`) i ostatni (`Last`) numery wiersza macierzy *M*, których prefiks jest równy *P* o długości *p* [7, 8]. Liczba wierszy w przedziale równa jest liczbie wzorców występujących w badanej sekwencji *T*. Można zauważyć, że czas wyszukiwania jest zależny od kosztu obliczania wartości $OCC(\cdot)$, przemnożonego przez $2p$.

```
Algorytm FM_COUNT(P[0, p-1])
1: i←p-1, First←0, Last←n-1;
2: while ((First≤Last) and (i≥0) do
3:        c←P[i];
4:        First←C[c]+OCC(c, First-1)+1;
5:        Last←C[c]+OCC(c, Last);
6:        i--;
7: if (First>Last) then return "not found"; else return (First, Last);
```

Rys. 2. Algorytm `FM_COUNT` wyznaczania wierszy występowania prefiksu $P[0, p\text{-}1]$) w macierzy *M*
Fig. 2. Algorithm `FM_COUNT` for finding the set of rows prefixed by $P[0, p\text{-}1]$ in the matrix M

Wykorzystując informację o zakresie występowania sufiksów w macierzy *M*, można określić ich położenie w oryginalnej sekwencji. Oznacza to, że dla każdej pozycji i=`First`, `First`+1, ..., `Last` w T^{BWT} należy osobno wyznaczyć jej odpowiednik w sekwencji *T*, który został oznaczony jako $Pos(i)$. Podstawową metodą do znajdowania wystąpień wzorca *P* jest zastosowanie lematu Ferraginy i Manziniego [7]. Zgodnie z nim, znając pozycję sufiksu z wektora *i*, można wyznaczyć pozycję dla indeksu z wektora *j*. W tym celu wykorzystuje się odwzorowanie *LF*, które zostało przedstawione w równaniu (1). $Pos(i)$ jest wyznaczane za pomocą iteracyjnego algorytmu `locate(i)`, przedstawionego na rys. 3. Przy transformacie BW zapamiętywany jest podzbiór wszystkich pozycji próbkowanych krokiem η. Im mniejszy jest ten krok, tym szybciej jest wyznaczana niezapisana wartość, jednak dzieje się to kosztem pamięci. Przy uwzględnieniu wyznaczania *LF* w każdej iteracji czas znalezienia poszukiwanej pozycji wyniesie $O(OCC\eta)$. Dlatego ważne jest, aby wybrać optymalny parametr kroku zapisu. Ferragina i Manzini proponują, aby krok był równy $\eta = \lceil \log^{1+\varepsilon} n \rceil$.

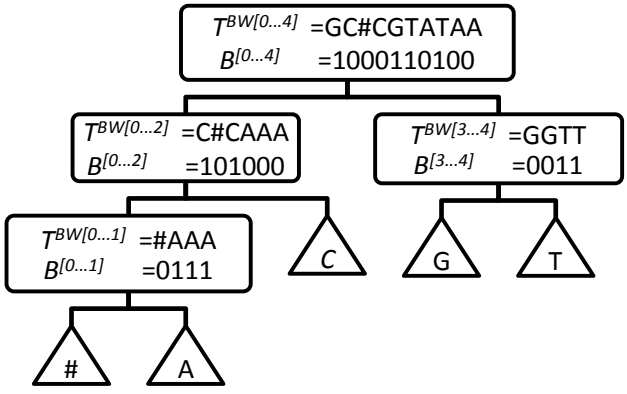
```
Algorytm locate(i)
1: i'←i, t←0;
2: while Pos(i') nie jest określone
3:        i'←LF[i'];
4:        t++;
5: return Pos(i')+t;
```

Rys. 3. Algorytm `locate` wyznaczający $Pos(i)$
Fig. 3. Algorithm `locate` for the computional $Pos(i)$

Szybkość wyznaczania liczby i miejsca występowania badanego wzorca w tekście zależy głównie od sposobu wyznaczania *OCC*. Istnieje kilka sposobów kompresji sekwencji w taki sposób, potem aby móc łatwo wyznaczać liczbę wystąpień danego znaku w prefiksie o zadanej długości. Przykładowo w tym celu budowane są tablice częstości występowań dla każdego znaku [7]. Możliwe jest także wykorzystanie kompresji Huffmana, a następnie przedstawienie sekwencji w sposób binarny [9] czy budowanie falkowych drzew binarnych (*Wavelet*

Tree) [10, 13]. W przedstawionym algorytmie otrzymaną sekwencję T^{BWT} zapisano za pomocą trzeciej metody – drzewa.

Drzewo falkowe dla sekwencji T^{BWT} zbudowanej z Σ alfabetu jest binarnym drzewem, które ma σ liści oraz wysokość równą $\lceil \log(\sigma) \rceil$. Każdy węzeł v odpowiada fragmentowi sekwencji składającej się ze znaków z danego przedziału alfabetu [l...r], a korzeń odpowiada całej sekwencji – [0, σ-1]. Jeśli l=r, wtedy węzeł v nie ma dzieci, w przeciwnym razie ma ich dwójkę. Lewe dziecko (v_l) ma nowy przedział równy [l...m], a prawe (v_r) – [m+1...r], gdzie $m = \lfloor (l+r)/2 \rfloor$. Przykładowe drzewo dla sekwencji T^{BWT}=GC#CGTATAA zostało przedstawione na rys. 4 (sekwencje znaków zostały pokazane tylko dla przejrzystości – w rzeczywistości nie są one przechowywane w drzewie). Znaki w węźle v są kodowane w ten sposób, że jeśli $c \in v_l$, to znak jest zapisywany jako 0, a w przeciwnym razie jako 1. Przy zastosowaniu drzewa poszukiwanie wartości $OCC(c, q)$ polega na przejściu ścieżki z korzenia do liścia z wykorzystaniem liczby jedynek w wektorach bitowych, znajdujących się w kolejnych węzłach ścieżki. Istnieje wiele struktur danych reprezentujących wektory bitowe, tworzonych w celu skrócenia czasu odpowiedzi na zapytanie [6, 15]. W przedstawionej pracy do szybszego wyznaczania liczby jedynek wykorzystano jedną z prostszych metod, dzieląc każdy wektor bitowy na dwa bloki o długościach $\ell = \lceil \log n \rceil$ i ℓ^2 i zapisując poszczególne zliczenia jedynek w wektorach [5, 11].



Rys. 4. Falkowe drzewo dla sekwencji T^{BWT}=GC#CGTATAA
Fig. 4. The wavelet tree of the string T^{BWT}=GC#CGTATAA

3. Wyszukiwanie sekwencji z maksymalną odległością Levenshteina

Celem realizowanych badań było znajdowanie fragmentu sekwencji P o długości m, która może zawierać trzy typy zmian:

- substytucję (*mismatch*) – w danym miejscu jest inny nukleotyd;
- delecję (*deletion*) – przesunięcie wynikające z braku nukleotydu;

- insercję (*insertion*) – przesunięcie wynikające z dodatkowego nukleotydu.

Sekwencja P	=	A	A	T	C	G	C	C	T	T	C	G	T	T	C
S_0	=	A	A	T	C	G								
	S_1 =	T	C	G	C	C								
		S_2 =	G	C	C	T	T							
			S_3 =	C	T	T	C	G						
				S_4 =	T	C	G	T	T					

Rys. 5. Ustalenie listy podziału na słowa (s=5, $\upsilon = 2$)
Fig. 5. The method to establish words list (s=5, $\upsilon = 2$)

Poszukiwany wzorzec P został podzielony na krótsze, nakładające się fragmenty-słowa o długości s, z przesunięciem $\upsilon \in < 2, s >$. Przez taki podział powstało $r = \lfloor (m-s)/\upsilon \rfloor + 1$ krótkich fragmentów. Na rys. 5 pokazano przykładowy podział sekwencji P (m=14) na słowa o długości 5, z przesunięciem równym 2. W ten sposób otrzymano 5 krótkich odcinków. Wszystkie słowa mają taką samą, ustaloną długość, zatem w niektórych przypadkach ostatnie znaki sekwencji P nie będą brane pod uwagę w żadnym słowie (w przypadku przedstawionym na rys. 5 jest to ostatni znak C). Znaki te zostaną poddane analizie na późniejszym etapie przetwarzania.

Zaproponowany algorytm poszukiwania fragmentów z zadaną maksymalną odległością Levenshteina (liczbą zmian) składa się z następujących etapów:

1. Stworzenie listy wystąpienia każdego słowa (S_i, $i = 0,...,r-1$). W tym celu wykorzystano przedstawione w poprzednim rozdziale algorytmy wyszukiwania dokładnego przy wykorzystaniu drzew falkowych z sekwencji po transformacie Burrowsa-Wheelera. Po stworzeniu listy pozycji dla każdego S_i została ona posortowana malejąco przy użyciu sortowania pozycyjnego (*radix sort*).
2. Poszukiwanie połączeń pomiędzy słowami z wykorzystaniem znajomości miejsca ich występowania w sekwencji.
3. Sprawdzenie brakujących fragmentów w znalezionych połączeniach.

3.1. Wyszukiwanie połączeń

Do poszukiwania kandydatów, czyli takich połączeń słów S, które nie będą zawierać więcej różnic niż zadana maksymalna wartość *MaxD*, posłużono się algorytmem przedstawionym na rys. 6. Zakłada on, że w jednym miejscu może wystąpić pojedyncze przesunięcie (insercja lub delecja) oraz że każdy rodzaj zmiany jest traktowany z identyczną wagą. Algorytm wykorzystuje r list pozycji (*ListPos*[i], i=0,...r-1), które wcześniej zostały posortowane malejąco. Każdy nowy kandydat jest zapisywany jako wektor par połączeń *PS*={*nrP*, *nrT*}, gdzie *nrP* jest miejscem pozycji w badanej sekwencji P, a *nrT* w sekwencji bazowej T. Nowe połą-

czenie ma zatem postać [PS_0,..., PS_k] i jest brane dalej pod uwagę tylko wtedy, gdy są nie mniej niż dwie pary (linia 24). Po utworzeniu nowego kandydata wszystkie pozycje dalszych słów, które należą do zbudowanego połączenia, są usuwane (linia 27).

```
Algorytm Connecting(ListPos)
1: δ←s/ν; p←0;
2: for (i←0, r-1, i++) //dla każdego Si
3:    While (ListPos[i] nie jest pusta)
4:         ed←ceil(i/ceil(s/ν));
5:         If (ed>MaxD)
6:              usunięcie ostatniego elementu dla ListPos[i];
7:         else
8:              k←0; nrP←i*ν; nrT←ostatni element w ListPos[i];
9:              PSp[k]={nrP, nrT};
10:             i'←i+δ;
11              While (i'<r) // poszukiwanie połączeń
12:                   nrP←i'*ν;
13:                   If (X∈ListPos[i']) // gdzie X←nrT+ν*δ | X←nrT+ν*δ-1
14:                                      //               | X←nrT+ν*δ+1
15:                      i'←i'+δ; k++;
16:                      nrT←X;
17:                      usunięcie X z ListPos[i'];
18:                      PSp[k]={nrP, nrT};
19:                   else
20:                      i'++;
21:                      nrT←nrT+ν;
22:                   korekta ed;
23:                   If (ed>MaxD), break;
24:              If (k>0)
25:                   p++;
26:                   dodanie nowego kandydata - PSp;
27:                   usunięcie wszystkich pozycji z ListPos[j], j={i+1,...,r-1},
28:                   które należą do nowego kandydata;
29:                   usunięcie ostatniego elementu z  ListPos[i];
30: return PS;
```

Rys. 6. Algorytm `Connecting` wyszukujący połączenia pomiędzy słowami
Fig. 6. Algorithm `Connecting` for finding links between words

Wyszukiwanie połączeń rozpoczyna się zawsze od najmniejszej wartości (końcowej) dla listy dla danego słowa (linia 8), zaczynając sprawdzanie po kolei dla każdego słowa (linia 2). Ponieważ sekwencja P została podzielona na nakładające się fragmenty, wystarczy sprawdzać nowe połączenie co $\delta = \lfloor s/\upsilon \rfloor$ słowo (linie 10 i 15). Dopiero gdyby takie nie wystąpiło, wtedy analizowane jest kolejne, czyli δ+1. Gdy ma się parę $PS[k]=\{nrP, nrT\}$ dla $S_{i'}$, sprawdza się, czy istnieje pozycja $X = nrT + \upsilon * \delta$ dla $S_{i'+\delta}$ słowa. Jeśli nie, to czy występuje pozycja o 1 mniejsza, a następnie o 1 większa. Po znalezieniu poszukiwanej pozycji jest dodawana nowa para – $PS[k+1]=\{nrP_{new}, nrT_{new}\}$, gdzie: $nrP_{new} = i'_{new} * \upsilon$, $i'_{new} = i' + \delta$, $nrT_{new} = X$ (linie 12-18). W przypadku braku wystąpienia poszukiwanej pozycji sprawdzana jest lista dla następnego słowa. W przypadku znalezienia połączenia z przesunięciami, braku połączenia lub startowania od słowa S_j (j > 0) uwzględniana jest korekta minimalnej liczby zmian (*ed*). Jest to liczba różnic pomiędzy fragmentami, która na pewno występuje w tworzonym połączeniu. Jeśli na jakimś etapie budowania połączeń *ed*>*MaxD*, to algorytm rozpoczyna nowe poszukiwanie.

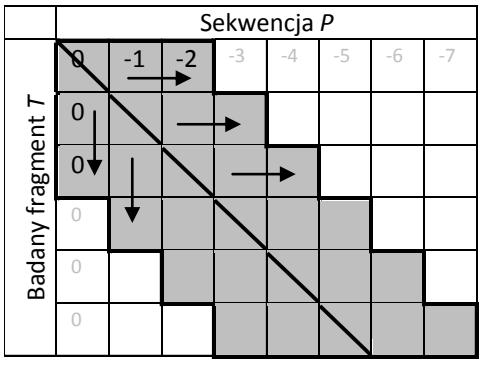
3.2. Weryfikacja kandydatów

W poprzednim podrozdziale wyznaczono pozycje występowania połączonych słów, które mogą spełniać założenie o dopuszczalnej, maksymalnej liczbie błędów w poszukiwanym fragmencie. Przy poszukiwaniu połączeń jednocześnie obliczano pewną minimalną liczbę różnic (*ed*) pomiędzy fragmentami badanym a znalezionym. Jednak w wielu przypadkach liczba ta może wynieść więcej. Przykład takiej sytuacji pokazany jest na rys. 7. Dla $s = 5$ oraz $v = 2$ wektor połączeń wynosi $PS = [\{0,70\}, \{4,74\}, \{12, 82\}, \{17, 88\}]$. Oznacza to, że jest przynajmniej jedno niedopasowanie pomiędzy pozycją 9. a 11. oraz występuje jedna delecja (pomiędzy 16., a 17.), a ponadto nie jest znany ostatni nukleotyd we fragmencie sekwencji *T*. Wynika z tego, że znaleziony fragment może być różny od szukanego *P* o odległość równą od 2 do 5. W celu weryfikacji kandydata należy odczytać brakujące miejsca w sprawdzanym fragmencie *T* oraz porównać je do poszukiwanego wzorca. Nieznane fragmenty odcinka są odtwarzane na podstawie informacji o T^{BWT}. W przedstawionym przykładzie wystarczyłoby porównanie 4 nukleotydów w celu wyznaczenia liczby błędów. Jednak dla niektórych przypadków jest to trudniejsze ze względu na potrzebę uwzględnienia przesunięć. Dlatego wzorzec *P* oraz badany fragment *T* są porównywane za pomocą zmodyfikowanego algorytmu Needlemana-Wunscha [14], w którym nie jest wyznaczana cała macierz podobieństwa, tylko jej przekątna oraz fragmenty tabeli, które są od niej oddalone o wartość maksymalnej dopuszczalnej różnicy. Na rys. 8 przedstawiono ogólny szkic macierzy; szare pole oznacza komórki, które są wyliczane dla *MaxD*=2.

P= AATCGCCTTCGTTCAGTATTTCT, *s*=5, v=2, *PS*=[{0,70}, {4,74}, {12, 82}, {17, 88}]

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		17	18	19	20	21	22
P=	A	A	T	C	G	C	C	T	T	C	G	T	T	C	A	G	T		A	T	T	T	C	T
	|	|	|	|	|	|	|	|	|	?	?	?	|	|	|	|	|	x	|	|	|	|	|	?
T=	A	A	T	C	G	C	C	T	T	?	?	?	T	C	A	G	T	?	A	T	T	T	C	?
	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93

Rys. 7. Przykładowy kandydat
Fig. 7. The example of candidate



Rys. 8. Ogólny zarys macierzy dopasowania
Fig. 8. General outline of the similarity matrix

4. Badania i wyniki eksperymentalne

Badany fragment P o długości m w przedstawionym algorytmie jest dzielony na słowa długości s nakładające się co v miejsce. Połączenie słów, które jest brane pod uwagę przy weryfikacji (rozdział 3.2), musi mieć minimum dwie pary oraz różnicę indeksów słów większą od $\delta = \lfloor s/\upsilon \rfloor$. Zatem parametry algorytmu należy dobrać w taki sposób, aby dla danej długości badanego fragmentu był możliwy taki podział – $\delta * \upsilon + s \le m$. Jednak założenie to nie jest wystarczające przy uwzględnianiu dopuszczalnej maksymalnej liczby zmian. W celu znalezienia fragmentu, w którym występuje *MaxD* różnic, należy parametry dobrać tak, aby $\lceil (k-2)/\lceil s/\upsilon \rceil \rceil \ge MaxD$, gdzie k jest liczbą podziału P na słowa S.

Tabela 1

Czasy poszukiwania 6 różnych sekwencji wraz z liczbą znalezionych fragmentów przy uwzględnieniu różnych wielkości parametrów

|P|	s	v	MaxD	l. S	czas [s]
35	6	2	3	77	0,858
	7	2	3	77	0,608
	8	2	3	76	0,390
	9	2	3	75	0,141
	10	2	3	54	0,063
30	6	2	3	261	0,936
	6	4	3	261	0,750
	7	1	3	260	0,655
	7	2	3	259	0,468
	8	1	3	245	0,203
	8	2	3	244	0,140
25	5	2	2	108	1,030
	6	2	2	108	0,297
	7	2	2	97	0,109
	8	1	2	45	0,031
	8	2	2	45	0,031
	9	1	2	16	0,031
20	5	2	2	13	1,654
	5	3	2	13	1,107
	6	2	2	12	0,327
	6	3	2	8	0,234
	7	2	2	7	0,078
18	4	2	1	1796	3,198
	4	4	1	1794	1,763
	5	2	1	1796	0,875
	6	2	1	1796	0,312
	7	2	1	1114	0,109
	8	2	1	942	0,062
15	4	1	1	629	3,713
	4	4	1	628	0,967
	5	1	1	629	0,733
	5	5	1	627	0,234
	6	1	1	627	0,187
	6	4	1	618	0,063

Przeprowadzone testy wykonano na sekwencji *>gi|74273667|gb|CM000266.1| Homo sapiens chromosome 15, whole genome shotgun sequence* o długości 78 891 134 par zasad, pobranej ze strony NCBI. Porównywano, jak wybór parametrów wpływa na liczbę znalezionych wyników oraz na zmianę czasu wyszukiwania. Do obliczeń użyto maszyny z procesorem Intel i5 430M, 2,27 GHz, 4 GB RAM. W celu przyspieszenia obliczeń na potrzeby po-

równań wyników dla różnych parametrów użyto małej wartości kroku markowania pozycji ($\eta = 2$, co zajęło 157 MB). Drzewo falkowe dla T^{BWT} wraz z zapisanymi zliczeniami jedynek w wektorach zajęło 35 MB. Tabela 1 przedstawia przykładowe wyniki liczb znalezionych fragmentów wraz z czasem wyszukiwania wzorców w zależności od długości sekwencji P oraz parametrów algorytmu: długości słów (s), przesunięcia (v) i maksymalnej dopuszczalnej liczby zmian ($MaxD$). Przy zmniejszaniu długości słowa znajdowana jest większa liczba słów w wyszukiwanej sekwencji (T). Przez to czas obliczania miejsc ich występowania wydłuża się, jednak takie rozwiązanie jest bardziej odporne na większą liczbę zmian położonych niedaleko siebie.

Również zmniejszenie v pozwala znaleźć większą liczbę wzorców P w sekwencji T, jednak zmiana ta jest już mniej odporna na blisko położone różnice pomiędzy dwoma sekwencjami. Zmiana parametru przesunięcia v wpływa w sposób liniowy na czas, ponieważ proporcjonalnie maleje liczba badanych słów. Dużo większe różnice w czasie są dla zmian s. Zmniejszenie długości słowa o 1 powoduje, że liczba znalezionych miejsc dla każdego słowa może wzrosnąć o około rząd wielkości (rys. 9). Parametr określający maksymalną dopuszczalną liczbę różnic nie ma dużego wpływu na czas wykonywania operacji, jednak należy wziąć go pod uwagę przy doborze dwóch pozostałych parametrów.

Rys. 9. Zależność średniej liczby znalezionych słów w sekwencji od ich długości
Fig. 9. Average number of found words in the sequence

5. Podsumowanie i kierunki dalszych prac

W niniejszym artykule przedstawiono algorytm wyszukiwania zadanych fragmentów DNA w sekwencji bazowej. Wykorzystanie podziału wzorca na krótkie, nakładające się słowa pozwoliło realizować wyszukiwanie z uwzględnieniem zadanej maksymalnej odległości Levenshteina. Przy doborze długości słów oraz przesunięcia należy wziąć pod uwagę długość poszukiwanego fragmentu P oraz dopuszczalną odległość Levenshteina. Im dłuższy jest po-

szukiwany wzorzec, tym słowa też mogą być dłuższe, a zwiększenie dopuszczalnej liczby zmian w badanej sekwencji powoduje, że słowa muszą być krótsze.

Przedstawiony algorytm wykrywa pojedyncze delecje i insercje w badanym fragmencie, jednak w przyszłych pracach należałoby również uwzględnić dopuszczalność dłuższych przerw lub wstawionych fragmentów. Ponadto przyszłe prace należałoby ukierunkować na przyspieszenie dokładnego wyszukiwania słów przez użycie lepszych algorytmów do wyznaczania OCC.

Dziękuję dr hab. inż. Sebastianowi Deorowiczowi za cenne wskazówki i porady, które były pomocne w realizacji przedstawionych badań.

Praca była współfinansowana ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego (nr umowy o dofinansowanie projektu: UDA-POKL.04.01.01-00-106/09).

BIBLIOGRAFIA

1. Altschul S. F., Gish W., Miller W., Myers E. W., Lipman D. J.: Basic local alignment search tool. J Mol Biol, Vol. 215(3) , 1990, s. 403÷410.
2. Baxevanis A., Ouellette B.: Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins, edition 2. Wiley, 2001.
3. Benson D. A., Karsch-Mizrachi I., Lipman D. J., Ostell J., Sayers E. W.: GenBank. Nucleic Acids Res, Vol. 39, 2011, s. D37÷D39.
4. Burrows M., Wheeler D.: A block sorting lossless data compression algorithm. Technical Report, Vol. 124, Digital Equipment Corporation, 1994.
5. Clark D.: Compact Pat Trees. PhD thesis. University of Waterloo, Canada 1996.
6. Claude F., Navarro G.: Practical rank/select queries over arbitrary sequences. Proc. 15th International Symposium on String Processing and Information Retrieval, LNCS, Vol. 5280, Springer-Verlag, 2008, s. 176÷187.
7. Ferragina P., Manzini G.: Indexing compressed text. Journal of the ACM, Vol. 52, 2005, s. 552÷581.
8. Ferragina P., Manzini G., Makinen V., Navarro G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms, Vol. 3(2), 2007.
9. Grabowski S., Navarro G., Przywarski R., Salinger A., Makinen V.: A simple alphabet-independent FM-index. Int. J. Found. Comput. Sci., Vol. 17(6), 2006, s. 1365÷1384.
10. Grossi R., Gupta A., Vitter J.: High-order entropy-compressed text indexes. SODA, 2003, s. 481÷580.

11. Jacobson G.: Succinct Static Data Structures. PhD thesis. CMU-CS-89-112, Carnegie Mellon University, 1989.
12. Kent W. J.: BLAT – the BLAST-like alignment tool. Genome Research, Vol. 12 (4), 2002, s. 656÷664.
13. Makinen V., Navarro G.: New search algorithms and time/space tradeoffs for succinct suffix arrays. Tech. Rep. C-2004-20, University of Helsinki, 2004.
14. Needleman S., Wunsch C.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol., Vol. 48(3), 1970, s. 443÷453.
15. Okanohara D., Sadakane K.: Practical entropy-compressed rank/select dictionary. Proc. Work-shop on Algorithm Engineering and Experiments, ALENEX, 2007.
16. Russo L. M., Navarro G., Oliveira A. L., Morales P.: Approximate String Matching with Compressed Indexes. Algorithms, Vol. 2, 2009, s. 1105÷1136.
17. Smith T. F., Waterman M. S.: Identification of Common Molecular Subsequences. J. Mol. Biol., Vol. 147(1), 1981, s. 195÷197.
18. Sonnhammer E. L., Durbin R.: A dot-matrix program with dynamic threshold control suited for genomic DNA and protein sequence analysis. Gene, Vol. 167(1-2), 1995.
19. Soumya R.: Computational text analysis for functional genomics and bioinformatics. Oxford University Press, New York 2006.
20. Välimäki N., Ladra S., Mäkinen V.: Approximate all-pairs suffix/prefix overlaps. CPM'10 Proceedings of the 21st annual conference on Combinatorial pattern matching, Springer-Verlag, 2010, s. 76÷87.

Wpłynęło do Redakcji 29 stycznia 2012 r.

Abstract

Nowadays, DNA sequencing methods are very efficient, and the number of known sequences is increasing rapidly. Therefore, the algorithms are needed which could help in fast searching of specific DNA patterns in the DNA databases. In many cases, the sequences are expected to be matched despite some small differences between them (termed insertions, deletions or mismatches), hence the searching algorithm should allow for approximate matching. In this paper we introduce an algorithm for searching DNA patterns in long sequences. It consists of three basic steps: dividing a pattern into words, searching for connections between exact positions of these words in the examined sequence, and final verification.

In the first step, the pattern is divided into words which overlap themselves. Subsequently, their positions in the sequence are determined using FM-index. The base sequence is compressed using the Burrows-Wheeler Transform and the Wavelet Tree, which makes it possible to find the words' positions in the sequence. Knowing the location of each word, it is possible to search such connections between them, which fulfill the assumption about a permissible maximum value of the Levenshtein distance. This creates a set of candidates for the alignment, which are verified in the last step of the algorithm. The obtained connections are incomplete, and their missing parts need to be restored before the candidate and the original pattern are compared using a modified Needleman-Wunsch algorithm.

In this study, it was investigated how the length of words and shifts between them affect the results. The experiments confirmed that the longer the words are, the less matched positions can be found. In particular, the largest differences in the number of matched positions are found, when the differences between the pattern and the sequence are located near each other. The main disadvantage of the proposed algorithm is that only single-nucleotide deletions and insertions are allowed, and this aspect will be investigated in the future work. Furthermore, it will be explored how to increase the speed of the searching process.

Adres

Jolanta KAWULOK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-101 Gliwice, Polska, jolanta.kawulok@polsl.pl.