Uniwersytet Jagielloński w Krakowie

Wydział Matematyki i Informatyki
Informatyka Analityczna

Praca Magisterska

Współbieżny algorytm dla problemu dopasowania krótkich sekwencji DNA

Andrzej Dorobisz

Opiekun: prof. dr hab. inż. Kazimierz Wiatr

Wrzesień 2015

Spis treści

Streszczenie

Problem dopasowania krótkich sekwencji DNA pochodzi z badań genetycznych. Do odczytywania DNA wykorzystuje się sekwencery, w wyniku działania których otrzymuje się serie krótkich fragmentów danego DNA (ang. *short-reads*). Takie odczyty porównuje się do zadanego genomu referencyjnego – celem jest znalezienie dla każdego odczytu miejsca w genomie referencyjnym, które jest identyczne lub podobne do tego odczytu.

Z algorytmicznego punktu jest to problem niedokładnego dopasowania wielu wzorców do zadanego tekstu, przy czym pracujemy na alfabecie 4-literowym (litery A, C, G, T). Możliwość niedokładnego dopasowania sprawia, że znalezienie wystąpień wzorców wymaga dużo czasu pracy procesora. Opracowano kilka technik, które są wykorzystywane do rozwiązania tego problemu – ich zrównoleglenie dokonuje się w większości w oparciu o wspólną pamięć.

Niniejsza praca stanowi przegląd algorytmów niedokładnego dopasowania wzorca do tekstu, analizuje możliwość ich zrównoleglenia i przedstawia implementację tego problemu w środowisku rozproszonym – wielu procesorów komunikujących się ze sobą, bez współdzielenia pamięci.

1 Przedstawienie problemu

W tym rozdziale przedstawiony jest problem, któremu ta praca jest poświęcona, wraz z jego wariantami i typowymi parametrami.

1.1 Dopasowanie wzorca do tekstu

Problem dopasowania wzorca do tekstu, czy inaczej nazywając, wyszukiwania wzorca w tekście (ang. *pattern matching*), jest jednym z najbardziej podstawowych problemów w dziedzinie algorytmów tekstowych. Problem definiujemy następująco: dla tekstu (ciągu znaków) $t = t_1 t_2 \dots t_n$ i wzorca (ciągu znaków) $p = p_1 p_2 \dots p_m$, znaleźć wszystkie wystąpienia wzorca p w tekście t, czyli wszystkie pozycje i takie, że $t_{i \dots i+m-1} = t_i t_{i+1} \dots t_{i+m-1} = p$. Równość dla ciągów oznacza równość na każdej pozycji, czyli w tym wypadku $t_{i+j-1} = p_j$, dla $1 \leqslant j \leqslant m$.

W zależności od potrzeby można rozważać inne warianty problemu dopasowania wzorca: tylko odpowiedź na pytanie czy wzorzec występuje w tekście, wskazanie pierwszego wystąpienia, podanie liczby wystąpień wzorca w tekście.

1.2 Niedokładne dopasowanie wzorca do tekstu

W ramach problemu niedokładnego dopasowania (ang. *approximate string matching* lub rzadziej używane *inexact matching*) można wyróżnić dwa problemy: dopasowanie z pomyłkami (ang. *string matching with mismatches*) oraz bardziej ogólne dopasowanie z różnicami (ang. *string matching with differences*) [1]. W literaturze ten drugi problem występuje także pod nazwą dopasowanie z błędami (ang. *string matching with errors*) [2, rozdział 7.1.8].

Kluczowe pojęcie dla niedokładnego dopasowania to **odległość edycyjna** (ang. *edit distance*). Dla słowa $a = a_1 a_2 \dots a_l$ definiujemy operacje:

- zamiana symbolu na pozycji i na symbol x, słowo a zmienia się w słowo $a_1 \dots a_{i-1} x a_{i+1} \dots a_l$,
- wstawienie symbolu x na pozycji i, powstaje $a_1 \dots a_{i-1} x a_i \dots a_l$,
- usunięcie symbolu na pozycji i, powstaje $a_1 \dots a_{i-1} a_{i+1} \dots a_l$.

Odległość edycyjna to miara podobieństwa dwóch wyrazów do siebie. Dla dwóch słów a i b, definiujemy $edit(a, b)$ jako najmniejszą ilość wyżej wymienionych operacji potrzebnych do zamiany wyrazu a na b. Odległość edycyjna spełnia warunki metryki. W literaturze występuje także jako odległość Levenshteina (ang. *Levenshtein distance*) [2].

Mając zdefiniowaną odległość edycyjną, możemy zdefiniować **problem dopasowania z k błędami**: dla tekstu t, wzorca p i danej liczby naturalnej k, znaleźć wszystkie takie pozycje i, że dla pewnego j zachodzi $edit(t_{i...j}, p) \leqslant k$. Innymi słowy – znaleźć wszystkie wystąpienia wzorca p w tekście t, dopuszczając możliwość co najwyżej k operacji edycji. Gdy $k = 0$ mamy problem dokładnego dopasowania (ang. *exact matching*) czyli problem wyszukiwania wzorca w tekście.

Prostszy **problem dopasowania z k pomyłkami** jest zdefiniowany analogicznie, z tym że dopuszcza się tylko możliwość zamiany jednej litery na drugą, nie można usuwać i wstawiać innych liter. Taka odległość wyrazów jest znana jako odległość Hamminga [2].

W literaturze jako problem niedokładnego dopasowania domyślnie przyjmuje się problem dopasowania z k błędami. Podobnie jak w przypadku dopasowania wzorca w tekście, możemy rozważać warianty: odpowiedź na pytanie czy da się dopasować wzorzec do tekstu przy zadanej ilości błędów, wskazanie wszystkich możliwych dopasowań, dopasowanie o najmniejszej ilości błędów.

1.3 Dopasowania krótkich sekwencji DNA

Problem dopasowania sekwencji DNA pochodzi z bioinformatyki (ang. *sequence alignment*). W związku z rozwojem technologii sekwencjonujących DNA, potrafiących odczytywać duże ilości krótkich fragmentów DNA, pojawiło się następujące zagadnienie: dopasować wiele małych fragmentów DNA (ang. *short-reads*) do dużego genomu referencyjnego. Dopasowanie może zawierać błędy.

Typowo genom referencyjny jest długości rzędu 1Gbp, gdzie jednostka bp oznacza parę zasad (ang. *base pair*), którą w zależności od typu i ułożenia możemy zapisać jako A, C, G lub T (patrzymy na jedną helisę). Zbiór odczytów zawiera z kolei 50-200 milionów krótkich fragmentów DNA, każdy o długości z przedziału 32-100 bp [3].

Z algorytmicznego punktu jest to problem niedokładnego dopasowania wielu krótkich wzorców do bardzo dużego tekstu w języku nad alfabetem czteroliterowym. W niniejszej pracy właśnie w ten sposób będziemy patrzeć na problem dopasowania krótkich sekwencji DNA.

2 Przegląd algorytmów dla problemu dopasowania z błędami

Niniejszy rozdział przedstawia algorytmy dla problemu niedokładnego dopasowania. Punktem wyjścia są algorytmy wyszukiwania wzorca w tekście (problem dokładnego dopasowania), które pozwalają wyrobić sobie ogląd na techniki stosowane w tej dziedzinie.

2.1 Algorytmy wyszukiwania wzorca w tekście

W tej części przedstawione są algorytmy dokładnego dopasowania. Jako, że problem ten nie jest główną częścią pracy, algorytmy są przedstawiane w zwięzły sposób, bez dokładnego uzasadniania poprawności i czasu działania. Tekst oznaczamy jako $t = t_1 t_2 \dots t_n$, wzorzec jako $p = p_1 p_2 \dots p_m$.

2.1.1 Metoda brute-force (wyszukiwanie "naiwne")

Najprostsze wyszukiwanie, opisywane w literaturze w ramach wprowadzenia do bardziej zaawansowanych algorytmów [2, r. 7.1.1], [4, r. 32.1]. Dla każdej pozycji $1 \leqslant i \leqslant n + 1 - m$ w tekście t próbujemy wprost dopasować wzorzec p. Dopasowanie odbywa się poprzez porównywanie kolejnych liter $t_{i+j} == p_j$, gdzie j przebiega od 1 do m. Pesymistyczny czas działania $O(nm)$.

2.1.2 Algorytm Knutha-Morrisa-Pratta (KMP)

Podstawowy algorytm prezentowany w literaturze, który działa w czasie liniowym [2, r. 7.1.2], [4, r. 32.4], [5, r. 3.1]. Wyszukiwanie wzorca odbywa się z użyciem tablicy prefikso-sufiksowej W dla wzorca p (w [4] nazywanej funkcją prefiksową). Wartość $W[i]$ to długość najdłuższego prefiksu właściwego słowa $p_{1\dots i}$ (prefiksu różnego od całego słowa), który jednocześnie jest sufiksem $p_{1\dots i}$.

Wyszukując wzorzec w tekście, czytamy tekst litera po literze i aktualizujemy wartość określającą ile pierwszych znaków wzorca można dopasować kończąc na obecnej pozycji w tekście. Jeśli na pozycji i mogliśmy dopasować j znaków (czyli $t_{i-j+1\dots i} == p_{1\dots j}$) to dokonujemy porównania $t_{i+1} == p_{j+1}$. Jeśli znaki są takie same – zwiększamy j o 1 i przechodzimy do kolejnego znaku w tekście ($i = i+1$). Jeśli nie – zmieniamy j na kolejne możliwe dopasowanie, które odczytujemy właśnie z tablicy prefikso-sufiksowej, tj. $j = W[j]$ i ponawiamy porów-

nanie ze znakiem wzorca. Gdy $j = 0$ wtedy dokonujemy ostatniego porównania i przechodzimy do kolejnego znaku, przy czym jeśli $t_{i+1} == p_1$ to ustawiamy $j = 1$, w przeciwnym wypadku zostawiamy $j = 0$. Gdy w którymś momencie $j = m$, wtedy znaleźliśmy wystąpienie wzorca w tekście.

Czas działania to $O(n+m)$. Potrzebujemy $O(m)$ czasu na preprocessing, czyli budowę tablicy prefikso-sufiksowej dla wzorca. Mając ją, wyszukujemy wzorzec w dowolnym tekście w czasie $O(n)$, to jest w czasie liniowym od długości tekstu.

2.1.3 Algorytm Boyera-Moora

Algorytm opisywany w literaturze [2, r. 7.1.3], [5, r. 4.1], o następującym ogólnym działaniu: idziemy od lewej do prawej przez tekst i próbujemy przypasować wzorzec do tekstu, tak by zaczynał się na danej pozycji. Jednakże samo dopasowanie wzorca wykonujemy od prawej do lewej, a przesunięcia pozycji startowej wykonujemy na podstawie wcześniej policzonej tabeli D.

Opisując działanie bardziej szczegółowo, chcąc dopasować wzorzec na pozycji i zaczynamy porównywać znaki $t_{i+m-1} == p_m$, następnie $t_{i+m-2} == p_{m-1}$, itd. Porównujemy je aż uda się dopasować cały wzorzec (wtedy zwracamy miejsce dopasowania) albo wystąpi różnica znaków. Załóżmy, że doszło do takiej na pozycji j we wzorcu, czyli $t_{i+j\ldots i+m-1} == p_{j+1\ldots m}$ oraz $t_{i+j-1} \neq p_j$. Wtedy korzystając z wcześniej policzonej tablicy D, dokonujemy przesunięcia miejsca początku kolejnego porównywania, czyli $i = i + D[j]$.

Tablica D jest liczona w preprocessingu na podstawie wzorca i jest tak wypełniona, żeby przesunięcie $i = i + D[j]$ było bezpieczne, tzn. żeby wiedząc, że $t_{i+j\ldots i+m-1} == p_{j+1\ldots m}$ oraz $t_{i+j-1} \neq p_j$ wynikało stąd, że wzorca nie da się dopasować na żadnej pozycji $[i, i + D[j] - 1]$. Istnieją różne sposoby i ulepszenia jak policzyć tę tablicę.

Czas wypełniania w preprocessingu tabeli D jest liniowy od długości wzorca i rozmiaru alfabetu [2] lub w innej wersji liniowy od długości wzorca [5]. Czas wyszukiwania pierwszego wystąpienia wzorca w tekście (lub stwierdzenia jego braku) wynosi $O(n)$. Pesymistyczny czas działania algorytmu wynosi $O(nm)$. Jest tak dlatego, że po wykonaniu przesunięcia możemy ponownie porównywać te same litery. Istnieje jednak ulepszenie algorytmu, która daje czas $O(n + rm)$, gdzie r to liczba wystąpień wzorca w tekście [5, r. 4.3]. W praktyce algorytm Boyera-Moora może jednak być szybszy niż KMP, gdyż nie wszystkie znaki tekstu muszą być przetworzone, co zmniejsza liczbę wykonanych porównań.

2.1.4 Algorytm Rabina-Karpa

Kolejnym algorytmem przedstawianym w literaturze jest algorytm Rabina-Karpa [2, r. 7.1.5], [4, r. 32.2]. Wykorzystuje on hashowanie do wyszukiwania wzorca w tekście. Obliczamy hash dla wzorca, a następnie liczymy hash dla każdego podsłowa tekstu długości m. Jeśli hash któregoś podsłowa jest równy hashowi wzorca, to wtedy dokonujemy porównania znak po znaku. Liczenie hashu dla podsłów realizujemy w następujący sposób: liczymy hash dla pierwszego podsłowa (słowa utworzonego przez pierwsze m liter tekstu), a następnie przesuwając się w prawo uaktualniamy hash.

Funkcja hashująca, która umożliwia łatwe obliczanie z poprzedniej wartości jest postaci:

$$h(t_{i\ldots i+m-1}) = t_i R^{m-1} + t_{i+1} R^{m-2} + \ldots + t_{i+m-1} R^0. \tag{2.1}$$

Wtedy obliczenie hashu podciągu długości m zaczynającego się na pozycji $i+1$ na podstawie wartości hashu dla podciągu zaczynającego się na pozycji i, to:

$$h(t_{i+1\ldots i+m}) = (h(t_{i\ldots i+m-1}) - t_i R^{m-1})R + t_{i+m}. \tag{2.2}$$

Działania wykonujemy w arytmetyce modularnej.

Czas preprocessingu to $O(m)$, tyle czasu potrzebujemy na obliczenie hashu wzorca. Wyszukiwanie wzorca w tekście zajmuje $O(n + rm)$, gdzie r to liczba podsłów o tym samym hashu co wzorzec ($O(n)$ zajmuje policzenie wszystkich hashów, $O(rm)$ zajmuje porównanie znak po znaku ze wzorcem słów o takim samym hashu).

2.1.5 Algorytm Aho-Corasick

A. V. Aho i M. J. Corasick w swojej pracy [6] przedstawili bardzo użyteczną metodę do wyszukiwania wielu wzorców w tekście. Dla zadanego zbioru wzorców P tworzymy drzewo *trie*. Jest to drzewo, którego krawędzie są etykietowane literami alfabetu. Wierzchołkowi odpowiada wyraz powstały z konkatenacji znaków na ścieżce z korzenia do tego wierzchołka. Takie drzewo sprawia, że jeśli kilka wyrazów ma wspólny prefiks, to część ścieżki do wierzchołków odpowiadających tym wyrazom jest wspólna.

Tak zbudowane drzewo uzupełniamy do automatu. Z każdego wierzchołka (stanu) będzie wychodzić krawędź dla każdej litery alfabetu. Dla wierzchołka

odpowiadającego słowu W, przejście literą a ma prowadzić do stanu odpowiadającego słowu Wa. Jeśli w drzewie z wierzchołka W wychodziła krawędź a, zostaje ona bez zmian. Jeśli niebyło takiej krawędzi, to znaczy że Wa nie występuje w drzewie (nie jest prefiksem żadnego wzorca). Wtedy krawędź a ma prowadzić do najdłuższego sufiksu Wa występującego w drzewie.

Przykładowo: dla drzewa *trie* zbudowanego dla wzorców *aab*, *abbaa*, *babab*, jeśli jesteśmy w stanie odpowiadającym *abba*, to krawędź a prowadzi do *abbaa*, natomiast krawędź b prowadzi do najdłuższego sufiksu słowa *abbab* występującego w drzewie, czyli do *bab*.

Automat buduje się wypełniając w kolejności BFS (przeszukiwanie wszerz z ang. *breadth-first search*) funkcję przejścia $next(s, a)$. Używa się w tym celu liczonej na bieżąco pomocniczej funkcji $fail(s)$, która dla stanu s prowadzi do najdłuższego występującego w tym drzewie właściwego sufiksu słowa odpowiadającego temu wierzchołkowi. Zależność $next$ od $fail$ jest następująca: jeśli w stanie s nie było w drzewie krawędzi a, to $next(s, a) = next(fail(s), a)$. Przy obliczaniu kolejność BFS jest kluczowa, gdyż gwarantuje nam, że przetwarzając stan s, funkcja przejścia dla $fail(s)$ będzie już policzona (BFS przetwarza stany w kolejności odległości od korzenia, a $fail(s)$ znajduje się bliżej korzenia niż s). Przyjmuje się, że wzorce nie zawierają się w sobie. Jeśli tak nie jest, trzeba w drzewie odpowiednio oznaczyć stany, odpowiadające znalezieniu wzorca.

Mając taki automat, przy wyszukiwaniu wystąpień wzorców ze zbioru P w tekście t, wczytujemy tekst litera po literze i chodzimy po naszym automacie, używając funkcji przejścia. W momencie dojścia do stanu odpowiadającego któremuś ze wzorców – mamy wystąpienie wzorca w tekście.

Czas działania algorytmu jest zależny od rozmiaru alfabetu. Preprocessing zajmuje $O(M|\Sigma|)$, gdzie M to sumaryczna długość wszystkich wzorców ze zbioru P, a Σ to alfabet. Wynika to z potrzeby policzenia funkcji przejścia dla każdego stanu. Mając zbudowany automat wyszukiwanie zajmuje już tylko $O(n)$ czasu.

2.1.6 Metoda shift-or

G. H. Gonnet i R. Baeza-Yates [2, r. 7.1.7] prezentują ciekawą technikę wyszukiwania wzorca w tekście. Pracujemy w niej na m-bitowych maskach, gdzie m to długość wzorca. Dla każdej litery alfabetu tworzymy na początku maskę, w której i-ty bit jest równy 0 jeśli we wzorcu i-ty znak jest równy tej literze. Maskę dla litery a oznaczmy przez $mask[a]$.

Na potrzeby wyszukiwania wzorca w tekście trzymamy maskę $match\_mask$, w której i-ty bit jest równy 0, jeśli ostatnie i znaków przeczytanego tekstu jest równe pierwszym i znakom wzorca. Wyszukiwanie rozpoczynamy od maski z samymi jedynkami (brak dopasowania) i od pozycji w tekście $j = 0$. W każdym kroku aktualizujemy maskę i przechodzimy do kolejnego znaku ($j = j + 1$). Wzorzec zostaje dopasowany jeśli m-ty bit maski wyniesie 0.

Aktualizując maskę dla pozycji j wykonujemy operację, od której nazwę bierze ta metoda:

$$match\_mask = (match\_mask \ll 1) \mid mask[text[j]]. \tag{2.3}$$

Uaktualnia się ona odpowiednio: $match\_mask[i + 1] == 0$ wtedy i tylko wtedy, gdy poprzednio $match\_mask[i] == 0$ oraz $mask[text[j]][i + 1] == 0$. Oznacza to, że aby $(i + 1)$ znaków pasowało do pozycji j, musiało do wcześniejszej pozycji (poprzednia wartość maski) pasować i znaków oraz aktualnie czytany znak tekstu ($text[j]$) musi być równy $(i+1)$-szemu znakowi we wzorcu. Za to ostatnie sprawdzenie odpowiada właśnie warunek $mask[text[j]][i + 1] == 0$.

Czas działania liczony w operacjach na maskach wynosi $O(|\Sigma| + m + n)$. W fazie preprocessingu liczone są maski dla każdego znaku alfabetu co wymaga $O(|\Sigma| + m)$ operacji na maskach. W fazie wyszukiwania wzorca w tekście w każdym kroku aktualizowana jest maska, co daje $O(n)$ operacji na niej.

2.1.7 Wyszukiwanie z użyciem tablicy sufiksowej

Tablica sufiksowa SA to tablica, w której $SA[i]$ zawiera pozycję i-tego leksykograficznie sufiksu danego słowa (czyli pozycję na której rozpoczyna się ten sufiks). Jak podaje literatura [5, r. 5.6] da się ją stworzyć w czasie liniowologarytmicznym od długości słowa.

Mając zbudowaną tablicę sufiksową dla tekstu, łatwo w nim wyszukać zadany wzorzec. Wszystkie jego wystąpienia w tekście przedłużone do końca tekstu, to sufiksy o tym samym prefiksie (którym jest właśnie wzorzec), a więc występują one w spójnym obszarze tablicy sufiksowej. Z racji na porządek leksykograficzny, możemy binarnie wyszukać pierwsze i ostatnie wystąpienie wzorca. Pozwala nam to znaleźć wszystkie wystąpienia wzorca w czasie $O(m \log(n)+r)$, gdzie r to liczba wystąpień wzorca w tekście. Składnik $m \log(n)$ odpowiada za wyszukiwanie binarne: $O(\log(n))$ porównań, każde wykonujemy w czasie $O(m)$. Składnik r odpowiada za odczytanie wartości tablicy SA w znalezionym przedziale.

2.1.8 Wyszukiwanie z użyciem transformaty Burrowsa-Wheelera

H. Li i R. Durbin [3] opisują ciekawą metodę wyszukiwania wzorca. Dla tekstu t obliczamy tablicę sufiksową SA. Następnie obliczamy transformatę Burrowsa-Wheelera, zdefiniowaną następująco

$$BWT[i] = t_{SA[i]-1}. \tag{2.4}$$

Za t_0 (pojawia się dla $SA[i] == 1$) przyjmujemy specjalny symbol \$, mniejszy leksykograficznie od wszystkich symboli alfabetu. Na BWT można popatrzeć jako na słowo powstałe z liter bezpośrednio poprzedzających sufiksy danego wyrazu, przy czym litery te sczytujemy zgodnie z kolejnością leksykograficzną sufiksów.

Jak wspomnieliśmy w poprzednim paragrafie, jeśli wzorzec W występuje w tekście, to odpowiada mu spójny fragment tablicy sufiksowej – są to sufiksy dla których jest prefiksem. Oznaczmy krańce tego przedziału przez $\underline{R}(W)$ oraz $\overline{R}(W)$ (są to odpowiednio najmniejszy i największy leksykograficznie sufiks, którego prefiksem jest W). Dla ϵ (pustego słowa) $\underline{R}(\epsilon) = 1$ oraz $\overline{R}(\epsilon) = n$.

Oznaczmy przez $C(a)$ ilość znaków tekstu t mniejszych leksykograficznie od litery a oraz przez $O(a, i)$ liczbę wystąpień znaku a w $BWT_{1\ldots i}$. Okazuje się wtedy, że jeśli W występuje w tekście t, to:

$$\begin{aligned} \underline{R}(aW) &= C(a) + O(a, \underline{R}(W) - 1) + 1, \\ \overline{R}(aW) &= C(a) + O(a, \overline{R}(W)), \end{aligned} \tag{2.5}$$

i $\underline{R}(aW) \leqslant \overline{R}(aW)$ wtedy i tylko wtedy, gdy aW również jest podsłowem t.

Wzory te da się dość prosto uzasadnić. Każde wystąpienie aW w tekście pociąga za sobą wystąpienie W. Powyższe wzory realizują odwrotne spojrzenie – pytamy, które z wystąpień W w tekście poprzedza litera a. Każda taka sytuacja daje nam wystąpienie aW. We wzorze na $\underline{\mathrm{R}}(\mathrm{aW})$, składnik $C(a)$ odpowiada za zliczenie wszystkich sufiksów leksykograficznie mniejszych od tych zaczynających się na literę a. Składnik $O(a, \underline{R}(W) - 1)$ mówi z kolei, ile spośród sufiksów zaczynających się na a jest leksykograficznie mniejszych od tych zaczynających się na aW. Dostajemy więc, że sufiksy zaczynające się na aW jeśli występują, to od pozycji $\underline{R}(aW) = C(a) + O(a, \underline{R}(W) - 1) + 1$. Analogicznie tłumaczy się $\overline{R}(aW)$, przy czym tam liczona jest pozycja gdzie kończą się sufiksy leksykograficznie nie mniejsze niż te, zaczynające się od aW.

Taka zależność pozwala, po uprzednim preprocessingu tekstu, wyszukiwać w nim dowolny wzorzec w w bardzo szybki sposób w czasie $O(|w|)$, tylko za pomocą iteracyjnego wyliczania wartości $\underline{R}$ i $\overline{R}$, począwszy od słowa pustego.

2.2 Algorytmy niedokładnego dopasowania wzorca w tekście

W tej części przedstawione są algorytmy dla głównego problemu – niedokładnego dopasowania wzorca w tekście, przy zadanej dopuszczalnej ilości błędów. Tekst oznaczamy jako $t = t_1 t_2 \dots t_n$, wzorzec jako $p = p_1 p_2 \dots p_m$, liczba dopuszczanych błędów to k.

2.2.1 Programowanie dynamiczne

Podstawowy algorytm dla problemu niedokładnego dopasowania wzorca jest bardzo podobny do sposobu liczenia odległości edycyjnej dwóch słów [5, r. 11.1]. Jest to programowanie dynamiczne, opisywane w większości pozycji związanych z problemem niedokładnego dopasowania [1], [2, r. 7.1.8], [5, r. 11.3], [7].

		0	1	2	3	4	5	6	7	8	9
			a	**b**	**b**	**d**	**a**	**d**	**c**	**b**	**c**
0		0	0	0	0	0	0	0	0	0	0
1	**a**	1	0	1	1	1	0	1	1	1	1
2	**d**	2	1	1	2	1	1	0	1	2	2
3	**b**	3	2	1	1	2	2	1	1	1	2
4	**b**	4	3	2	1	2	3	2	2	1	2
5	**c**	5	4	3	2	2	3	3	2	2	1

Rysunek 2.1: Tablica D dla dopasowania wzorca *adbbc* w tekście *abbdadcbc*

Wypełniamy tabelę $D[0 \dots m][0 \dots n]$ tak, aby $d_{i,j} = D[i][j]$ było równe minimalnej odległości edycyjnej pomiędzy pierwszymi i znakami wzorca a jakimś podsłowem tekstu t, kończącym się na pozycji j, czyli

$$d_{i,j} = \min_{1 \leqslant q \leqslant j+1} (edit(p_{1\dots i}, t_{q\dots j})) \tag{2.6}$$

(dopuszczamy $q = j+1$, wtedy $t_{j+1\dots j}$ oznacza słowo puste). Inicjujemy pierwszy rząd w tabeli $d_{0,j} = 0$ (dopasowanie pustego słowa nie wymaga żadnych edycji) oraz pierwszą kolumnę $d_{i,0} = i$ (dopasowanie słowa długości i do pustego słowa wymaga i usunięć). Dla pozostałych wartości ($1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n$) zachodzi

$$d_{i,j} = \min(d_{i-1,j} + 1, d_{i,j-1} + 1, d_{i-1,j-1} + (p_i \neq t_j)). \tag{2.7}$$

Jest tak dlatego, że najlepsze dopasowanie $p_{1\ldots i}$ do jakiegoś podsłowa $t_{q\ldots j}$ powstało albo z dopasowania $p_{1\ldots i-1}$ do $t_{q\ldots j}$ i usunięcia p_i, albo z dopasowania $p_{1\ldots i}$ do $t_{q\ldots j-1}$ i wstawienia t_j na koniec wzorca, albo z z dopasowania $p_{1\ldots i-1}$ do $t_{q\ldots j-1}$ oraz porównania znaków p_i z t_j – jeśli były różne, potrzeba było jeszcze jednej edycji, zamieniającej p_i na t_j.

Mając taką zależność oraz wypełnione skrajne wartości (pierwszą kolumnę i pierwszy rząd) możemy wypełnić całą tablicę idąc w kolejności rząd po rzędzie lub kolumna po kolumnie. Jeśli któraś z wartości w ostatnim rzędzie tej tablicy wyniesie nie więcej niż k, wtedy możliwe jest dopasowanie wzorca. Dokładniej: jeśli dla jakiegoś j zachodzi $d_{m,j} \leqslant k$, to znaczy, że robiąc nie więcej niż k edycji, da się dopasować nasz wzorzec do sufiksu $t_{1,j}$.

Rysunek 2.1 przedstawia wypełnioną tablicę D dla tekstu *abbdadcbc* i wzorca *adbbc*. Dopuszczając dwa błędy możemy dopasować wzorzec do tekstu kończąc na pozycjach 3, 4, 7, 8 i 9.

Czas działania algorytmu to $O(mn)$. Tablica D jest zależna zarówno od tekstu jak i wzorca, przy zmianie wzorca musimy więc liczyć ją od nowa.

2.2.2 Wyszukiwanie za pomocą automatu

E. Ukkonen [7] przedstawia algorytm, w którym dla wzorca budowany jest automat, który pozwala, w czasie liniowym od długości tekstu, odpowiedzieć czy da się dopasować wzorzec przy zadanej liczbie błędów. Algorytm bazuje na standardowym rozwiązaniu, opisanym w poprzednim podpunkcie. Stanem w automacie jest kolumna z tablicy D. W fazie preprocessingu, kiedy tworzymy automat, rozważamy wszystkie możliwe przejścia, czyli jak zmienia się kolumna pod wpływem każdej litery z alfabetu. Jeśli w wyniku przejścia powstaje stan jakiego jeszcze nie było, dodajemy go do kolejki stanów do przetworzenia. Na początku zaczynamy od stanu $Q = (0, 1, 2, \ldots, m)$. Dla dowolnego stanu S przez S_i oznaczamy wartość i-tej komórki w kolumnie. Stan S jest akceptujący jeśli $S_m \leqslant k$.

Obliczanie stanu B do jakiego trafimy z danego stanu A, idąc daną literą x, jest analogiczne do obliczenia $(i + 1)$-szej kolumny na podstawie i-tej kolumny w standardowym algorytmie. Zaczynamy od przypisania $B_0 = 0$, po czym kolejno dla $i = 1, i = 2, \ldots, i = m$ obliczamy $B_i = min(B_{i-1} + 1, A_i + 1, A_{i-1} + (x \neq p_i))$. Rysunek 2.2 ukazuje przykładowy stan $A = (0, 1, 2, 1, 1, 2)$ i jego funkcję przejścia, tj. do jakich stanów przejdziemy idąc literą a, b, c lub d. Stan A odpowiada czwartej kolumnie z tablicy D, przedstawionej jako przykład w poprzednim pa-

0	
1	**a**
2	**d**
3	**b**
4	**b**
5	**c**

0
1
2
1
1
2

(a) przykładowy stan A

0
0
1
2
2
2

0
1
2
2
1
2

0
1
2
2
2
1

0
1
1
2
2
2

(b) stany powstałe z A po przejściu odpowiednio literą a, b, c lub d

Rysunek 2.2: Przykład funkcji przejścia w automacie dla wzorca *adbbc*

ragrafie (rysunek 2.1), czyli jest to stan do jakiego dotrzemy ze stanu startowego idąc kolejno literami a, b, b.

Zauważmy, że jeśli w którymś miejscu kolumny wpisana jest wartość powyżej k, to w ciągu przekształceń, ta wartość nie wygeneruje już mniejszej wartości, w szczególności nie może spowodować, żeby na końcu innej kolumny była wartość nie większa niż k. Oznacza to, że ta wartość nie może sprawić, że jakiś pochodny stan będzie stanem akceptującym. Możemy więc każdą wartość większą niż k zamieniać na $k+1$. Takie podejście powoduje, że liczba możliwych stanów ogranicza się do $(k+2)^m$ (stan ma $m+1$ wartości z zakresu $0, 1, \ldots, k+1$, ale pierwsza wartość zawsze jest równa 0).

Kolejną obserwacją, która pozwala na oszacowanie ilości możliwych stanów oraz ich zwartą reprezentację, jest zależność sformowana przez Ukkonena jako Lemat 1. Dla dowolnego stanu B:

$$B_i - B_{i-1} = -1, 0 \text{ lub } 1, \text{ gdzie } 1 \leqslant i \leqslant m. \tag{2.8}$$

Lemat ten mówi, że różnica sąsiednich wartości w kolumnie nie przekracza 1. Udowodnimy go indukcyjnie. Dla stanu startowego ($B_i = i$) zależność jest spełniona. Teraz pokażemy, że jeśli stan B powstał ze stanu A, dla którego lemat

zachodzi, to zależność zachodzi również dla B. W oczywisty sposób $B_i \leqslant B_{i-1}+1$, gdyż $B_{i-1}+1$ wchodzi do minimum, z którego powstaje B_i. Stąd $B_i - B_{i-1} \leqslant 1$. Pozostaje pokazać, że różnica ta jest nie mniejsza niż -1. Wartość B_i pochodzi od jednej z trzech wartości: A_{i-1}, A_i, B_{i-1}. Jeśli pochodzi od B_{i-1}, to zależność jest spełniona, gdyż $B_i - B_{i-1} = 1$. W drugim przypadku pochodzi od A_{i-1} lub A_i. Oznaczając $A_{i-1} = x$, z założenia indukcyjnego wiemy, że $A_i = x-1, x$ lub $x+1$. Niezależnie od przypadku, $B_i = x$ lub $B_i = x+1$. Z kolei ze wzoru z minimum $x+1 = A_{i-1}+1 \geqslant B_{i-1}$. Stąd możemy zapisać $B_i \geqslant x = (x+1)-1 \geqslant B_{i-1}-1$, co nam daje oczekiwaną zależność $B_i - B_{i-1} \geqslant -1$.

Taka zależność i fakt, że dla każdego stanu $S_0 = 0$, pozwala przechowywać stany jako liście w drzewie ternarnym o wysokości m. W takim drzewie każdy wierzchołek ma co najwyżej trzy krawędzie wychodzące – w naszym wypadku byłyby to krawędzie odpowiadające zmianie wartości o $-1, 0, 1$ idąc z góry do dołu kolumny. Przy takiej reprezentacji dodawanie nowych stanów oraz sprawdzanie czy jakiś stan już występował jest szybkie. Dostajemy także lepsze ograniczenie na liczbę stanów: 3^m (stan kodujemy za pomocą m wartości $-1, 0, 1$).

Po skonstruowaniu automatu niedokładne dopasowywanie wzorca w tekście jest tak proste jak czytanie tekstu znak po znaku i przechodzenie po stanach zgodnie z funkcją przejścia. Jeśli dojdziemy do stanu akceptującego – wtedy zwracamy odpowiedź, że da się w tym miejscu dopasować wzorzec. Czas działania samego dopasowania wzorca w tekście wynosi $O(n)$. Przy wyszukiwaniu tego samego wzorca w wielu tekstach takie podejście oszczędza dużo czasu, gdyż automat tworzymy tylko raz.

2.2.3 Liczenie wzdłuż przekątnych

W. Rytter i M. Crochemore [5, r. 11.3] opisują algorytm, będący ulepszeniem standardowego programowania dynamicznego (patrz 2.2.1). Podobnie jak w podstawowym algorytmie, pracujemy na tablicy $D[0 \ldots m][0 \ldots n]$, ale tym razem obliczamy tylko niektóre jej wartości. Wartości te obliczamy wzdłuż przekątnych, nie przechowując całej tablicy w pamięci.

Z poprzedniego paragrafu wiemy, że jeśli $d_{i,j} > k$, to z pola (i, j) nie otrzymamy dopasowania o nie więcej niż k błędach. Do tego faktu dodajmy jeszcze jedną dość intuicyjną zależność:

$$d_{i,j} \leqslant d_{i+1,j+1} \leqslant d_{i,j} + 1. \tag{2.9}$$

Mówi ona tyle, że w tabeli D, idąc wzdłuż przekątnych, z lewej do prawej, z góry

do dołu, mamy do czynienia z ciągiem niemalejącym. Inaczej mówiąc, dopasowując od danego miejsca w tekście kolejne litery wzorca, nie możemy dopasować dłuższego prefiksu wzorca za pomocą mniejszej ilości operacji edycji.

Uzasadniając tę zależność formalnie, prawa strona nierówności wynika ze wzoru 2.7 dla $d_{i+1,j+1}$, w którym w minimum występuje $d_{i,j}+1$. Lewą stronę nierówności udowadniamy indukcyjnie. Bazą indukcyjną jest pierwszy ($i = 0$) i drugi rząd ($i = 1$), gdyż $d_{0,j} = 0 \leqslant d_{1,j+1}$. Dalej udowadniamy w kolejności rząd po rzędzie. Dla $(i+1, j+1)$, $0 \leqslant i \leqslant m-1$, $0 \leqslant j \leqslant n-1$, korzystamy z tego, że teza zachodzi dla wszystkich pól z wcześniejszych rzędów. Oznaczmy $x = d_{i,j}$. Z obserwacji z poprzedniego paragrafu wiemy, że różnica sąsiednich wartości w kolumnie wynosi co najwyżej 1. Stąd wnioskujemy, że $x-1 \leqslant d_{i+1,j}$ oraz że $x-1 \leqslant d_{i-1,j}$. Z założenia indukcyjnego mamy, że $x-1 \leqslant d_{i-1,j} \leqslant d_{i,j+1}$. Razem zestawiając, otrzymujemy: $x \leqslant d_{i+1,j}+1$, $x \leqslant d_{i,j+1}+1$, $x \leqslant d_{i,j}+(p_{i+1} \neq t_{j+1})$, co prowadzi nas do wniosku, że minimum z tych wartości, czyli $d_{i+1,j+1}$, jest nie mniejsze niż x, co kończy dowód.

Z wykazanego faktu wiemy, że pola o tej samej wartości tworzą w obrębie przekątnej spójny fragment. Dla każdej przekątnej definiujemy tak zwany d-node, dla $d = 0, 1 \ldots k$, jako najdalsze pole o wartości d w obrębie tej przekątnej. Sumarycznie jest ich $O(k(n+m))$, gdyż mamy $n+m+1$ przekątnych. Wzorzec możemy dopasować do tekstu na pozycji j, gdy zachodzi $d_{m,j} \leqslant k$, czyli jeśli pole (m, j) jest d-nodem w obrębie swojej przekątnej.

		0	1	2	3	4	5	6	7	8	9
			a	**b**	**b**	**d**	**a**	**d**	**c**	**b**	**c**
0		0	0	0	0	0	0	0	0	0	0
1	**a**	1	0	(1)	1	1	0	1	(1)	(1)	(1)
2	**d**	2	1	1	2	(1)	1	0	1	2	2
3	**b**	3	2	1	(1)	2	2	(1)	1	(1)	2
4	**b**	4	3	2	(1)	2	3	2	2	1	2
5	**c**	5	4	3	2	2	3	3	2	2	(1)

Rysunek 2.3: Tablica D z zaznaczonymi polami d-node dla $d = 1$

Rysunek 2.3 przedstawia wypełnioną tabelę D dla dopasowania wzorca *adbbc* w tekście *abbdadcbc* z wyróżnionymi polami 1-node. Jak widać, nie w każdej przekątnej musi występować każdy d-node. W przedstawionym przykładzie, dopuszczając jeden błąd wzorzec można dopasować tylko na samym końcu tekstu.

Zaproponowany przez W. Ryttera i M. Crochemora algorytm przedstawia efektywny sposób policzenia tych pól. W kolejnych iteracjach liczymy d-nody dla $d = 1, \ldots, k$ na podstawie wcześniejszego zbioru $(d-1)$-nodów. Korzystamy z dodatkowej funkcji $match(i, j)$, która zwraca długość najdłuższego dokładnego dopasowania wzorca od pozycji $i + 1$ do tekstu od pozycji $j + 1$ (jest to maksymalny wspólny prefiks dla słów $p_{i+1\ldots m}$ oraz $t_{j+1\ldots n}$). Zauważmy, że zachodzi $d_{i,j} = d_{i+match(i,j),j+match(i,j)}$. Dla łatwiejszego zapisu niech zapis $(i, j)+t$ oznacza pole $(i + t, j + t)$. Zaczynamy od zbioru 0-nodów, który liczymy następująco:

$$DN_0 = \{(0, j) + match(0, j) \mid 0 \leqslant j \leqslant n\}. \tag{2.10}$$

Następnie w każdej iteracji, w następujących krokach obliczamy zbiór d-nodów:

$$\begin{aligned} SN_d &= \{(d, 0)\} \cup \{(i + 1, j + 1), (i, j + 1), (i + 1, j) \mid (i, j) \in DN_{d-1}\}, \\ LN_d &= \{\text{dla każdej przekątnej najniższy } (i, j) \text{ ze zbioru } SN_d\}, \\ DN_d &= \{(i, j) + match(i, j) \mid (i, j) \in LN_d\}. \end{aligned} \tag{2.11}$$

Poprawność wynika stąd, że dla danej przekątnej wartość d może pojawić się na niej na jeden z trzech sposobów: z wcześniejszej przekątnej, z tej samej przekątnej lub z następnej przekątnej; w każdym wypadku z pola o wartości $d - 1$. Zbiór SN_d to zbiór najdalszych miejsc gdzie takie wartości się pojawiają. Biorąc z tych wartości najdalszą (o największych współrzędnych (i, j)) w obrębie danej przekątnej, wartość d może się dalej pojawić już tylko na skutek przejścia dokładnym dopasowaniem wzdłuż przekątnej. Stąd po dodaniu $match(i, j)$ do elementów zbioru LN_d otrzymujemy zbiór najdalszych wystąpień wartości d na każdej przekątnej.

Jak łatwo zauważyć, ten algorytm wykonuje $O(k)$ iteracji, w każdej przeglądając zbiory wielkości $O(n + m)$. Kluczowy dla czasu działania algorytmu jest czas działania operacji $match$. Okazuje się, że przy alfabecie o stałym rozmiarze, można wykonać preprocessing w czasie liniowym, który będzie pozwalał na obliczanie $match(i, j)$ w czasie stałym [5]. W tym celu buduje się wspólne drzewo sufiksowe dla wzorca i tekstu. Zapytania $match(i, j)$ odpowiadają problemowi LCA w tym drzewie, czyli zapytaniom o najniższego wspólnego przodka (ang *lowest common ancestor*). W [5] podano sposób w którym, po specjalnym liniowym preprocessingu wspomnianego drzewa, na zapytania te odpowiada się w czasie stałym (patrz [5]). Sumarycznie daje nam to czas $O(k(n + m))$ co przy wzorcu mniejszym od tekstu daje czas $O(kn)$.

2.2.4 Backtracking z użyciem BWT

H. Li i R. Durbin w swojej pracy [3] używają opisanego sposobu wyszukiwania opartego o transformatę Burrowsa-Wheelera (paragraf 2.1.8) do problemu niedokładnego dopasowania. Jest to tak naprawdę backtracking, testujący w każdym kroku wszystkie możliwości operacji edycji (pójście po literze, zamiana litery, pominięcie litery, wstawienie litery).

Wyszukiwanie realizuje funkcja $InexactRec(W, i, z, r_{beg}, r_{end})$, której parametrami są:

- W, wzorzec,
- i, pozycja we wzorcu, którą teraz próbujemy dopasować,
- z, dopuszczalna liczba błędów,
- r_{beg}, wartość $\underline{R}(W_{i+1\ldots l})$, gdzie $l = |W|$,
- r_{end}, wartość $\overline{R}(W_{i+1\ldots l})$.

Przed rozpoczęciem dopasowania wykonujemy preprocessing, w którym obliczamy tablicę sufiksową SA oraz spamiętujemy wartości tablic C i O (potrzebnych do obliczania wartości $\underline{R}$ i $\overline{R}$, zgodnie ze wzorami 2.5). Wyszukiwaniu wzorca p odpowiada wywołanie $InexactRec(p, m, k, 1, n)$.

Wewnątrz funkcji wywołujemy się rekurencyjnie, sprawdzając wszystkie możliwe operacje edycji. W kolejnych wywołaniach zmniejszamy parametr i, gdyż dopasowujemy wzorzec od tyłu. Mamy następujące wywołania:

- $InexactRec(W, i-1, z-1, r_{beg}, r_{end})$, odpowiadające usunięciu litery ze wzorca,
- $InexactRec(W, i, z-1, r_{beg}`, r_{end}`)$, odpowiadające wstawieniu litery x na pozycję $i+1$,
- $InexactRec(W, i-1, z-(x \neq W_i), r_{beg}`, r_{end}`)$, odpowiadające przejściu literą W_i lub zamianie litery na inną.

W ostatnim wywołaniu x to litera, którą właśnie przechodzimy (gdy $x == W_i$, wtedy nie wykorzystujemy jednego błędu i parametr z nie jest zmniejszany). Wartości $r_{beg}`, r_{end}`$ to odpowiednio $\underline{R}(xW_{i+1\ldots l})$ oraz $\overline{R}(xW_{i+1\ldots l})$ policzone zgodnie ze wzorami 2.5 ($r_{beg}` = C(x) + O(x, r_{beg} - 1) + 1$, $r_{end}` = C(x) + O(x, r_{end})$). Oczywiście gdy z spadnie do 0, wtedy kontynuujemy już tylko dokładne wyszukiwanie. Jeśli $r_{beg} > r_{end}$, to nie kontynuujemy rekurencji. Jeśli $i = 0$, to zwracamy sufiksy $SA[r_{beg}], SA[r_{beg}+1], \ldots, SA[r_{end}]$ jako te, do których da się dopasować wzorzec.

2.3 Analiza zastosowania algorytmów do głównego problemu

W tej części przedstawiamy ogólne pomysły zastosowania opisanych algorytmów do głównego problemu jakim jest niedokładne dopasowanie krótkich sekwencji DNA. Dla algorytmów wyszukiwania wzorca w tekście rozważamy możliwości zastosowania tych technik do problemu niedokładnego dopasowania. Dla algorytmów niedokładnego dopasowania rozważamy sposoby ich zrównoleglenia w kontekście głównego problemu.

2.3.1 Zastosowanie technik wyszukiwania wzorca w tekście do problemu dopasowania z błędami

Pierwsza metoda – algorytm brute-force, jest użyteczna, ale dla prostszego problemu, czyli dopasowania z k pomyłkami. Wtedy tak jak wcześniej, dla każdej pozycji dopasowujemy wzorzec litera po literze, ale nie przerywamy dopasowywania w momencie wystąpienia różnicy tylko zliczamy liczbę pomyłek. Próbę dopasowania przerywamy dopiero w momencie przekroczenia liczby k. Jednakże dla ogólnego problemu niedokładnego dopasowania ciężko mówić o metodzie brute-force. W takiej metodzie próbujemy dopasować wzorzec najpierw zaczynając od pozycji 0 w tekście, potem od pozycji 1, i tak aż do końca tekstu. Dopasowanie wzorca do danej pozycji tekstu poprzez sprawdzenie wszystkich możliwości błędnego dopasowania prowadzi do backtrackingu podobnego do tego z użyciem BWT (paragraf 2.2.4), czyli wywołania rekurencyjne dla każdej możliwej operacji edycji. Jednakże w tamtej metodzie backtracking uruchamiamy tylko raz dla wzorca. Tutaj uruchamiamy go dla każdej pozycji tekstu co prowadzi do bardzo dużej złożoności.

Techniki stosowane w algorytmach KMP oraz Boyera-Moora wydają się nie do przeniesienia do problemu niedokładnego dopasowania. W tych algorytmach bazujemy na tym, że mając dopasowaną do tekstu w sposób dokładny część wzorca, umiemy coś powiedzieć o możliwych kolejnych dopasowaniach. W momencie wystąpienia różnicy, wykorzystujemy tę informację do skrócenia dopasowanego już fragmentu (KMP), czy też do przesunięcia do kolejnej możliwej pozycji (Boyer-Moore). Jednakże w niedokładnym dopasowaniu w przypadku różnicy możemy po prostu ją zignorować wykorzystując błąd. Można myśleć o policzeniu odległości edycyjnej fragmentów wzorca, ale taka informacja też jest trudna do wykorzystania. Załóżmy, że znamy odległość edycyjną fragmentu wzorca do

fragmentu tekstu i próbujemy przesunąć wzorzec inaczej dopasowując go do tego samego tekstu. Wtedy nawet znając odległość edycyjną pomiędzy pokrywającymi się fragmentami wzorca, nie wiemy czy występujące różnice działają na korzyść czy na niekorzyść dopasowania do danego fragmentu tekstu.

Hashowanie na pierwszy rzut oka wydaje się kompletnie nieprzydatne do problemu niedokładnego dopasowania. Hash nie oddaje w żaden sposób struktury słowa i dwa różne hashe oznaczają dwa różne słowa beż żadnej informacji o ich podobieństwie. Zauważmy jednak, że przy niedużej liczbie dopuszczanych błędów wiemy, że jakiś fragment wzorca musi wystąpić w sposób dokładny w tekście. Przykładowo dla wzorca długości 32 i dopuszczalnych 4 błędów, żeby dopasować wzorzec do tekstu, muszą w nim wystąpić dwa rozłączne podciągi długości 6 z wzorca lub jeden długości 8. Obserwacja ta jest wynikiem prostego rozumowania: wystąpienia 4 błędów dzielą wzorzec na 5 części, do tego każdy błąd może usuwać jedną literę. Stąd mamy 5 rozłącznych części z dokładnym dopasowaniem, o łącznej długości co najmniej 28. Jeśli tylko jeden blok ma długość większą niż 5, wtedy łączna długość pozostałych czterech nie przekracza 20, co oznacza, że piąta część musi mieć długośc co najmniej 8. Hashe podciągów określonej długości mogą więc przydać się do szybkiego sprawdzenia gdzie takie podciągi występują. Mogłaby to być pierwsza faza algorytmu, który szybko zawęża potencjalne miejsca wystąpienia wzorca, a potem już tylko na nich inną metodą przeprowadza niedokładne dopasowanie.

Automat Aho-Corasick wydaje się ciekawą techniką dla głównego problemu, gdyż mamy w nim do czynienia z wieloma wzorcami. Chodzenie po tym automacie musiałoby opierać się na backtrackingu podobnym do tego jak w metodzie używającej BWT. Problemem jednakże jest określenie liczby błędów po przejściu krawędzią niedrzewową. Nawet gdyby udało się temu zaradzić, to wobec bardzo długiego tekstu taki backtracking zająłby dużo więcej czasu niż w metodzie z użyciem BWT, gdzie backtracking odbywa się na wzorcu, który jest stosunkowo krótki. Można natomiast do głównego problemu wykorzystać samo drzewo trie zbudowane dla wzorców. Jeśli wzorce mają wspólny prefiks, to nie trzeba dla każdego z tych prefiksów obliczać w jakie miejsca może on pasować, wystarczy obliczyć to tylko jeden raz. Stąd wniosek, że w algorytmach niedokładnego dopasowania, zamiast przetwarzać wzorzec po wzorcu, możemy przetwarzać wierzchołki z drzewa trie wszystkich wzorców, oszczędzając w ten sposób czas na wspólnych fragmentach wzorców.

W metodzie shift-or, zaletą metody jest to, że przechowujemy maskę bitową, którą aktualizujemy operacjami bitowymi, co może dać duże przyspieszenie w obliczeniach. Jednakże przy niedokładnym dopasowaniu potrzebujemy więcej informacji – z iloma błędami możemy dopasować dany fragment wzorca. Zamieniając bity na liczby dostaniemy tablicę z liczbami od 1 do k, aktualizowaną przy przejściu do kolejnej litery tekstu. To zaś prowadzi nas do klasycznego sposobu jakim jest programowanie dynamiczne.

Odnośnie tablicy sufiksowej oraz transformaty Burrowsa-Wheelera, z paragrafu 2.2.4 wiemy już, że da się je skutecznie wykorzystać do problemu niedokładnego dopasowania.

2.3.2 Analiza możliwości zrównoleglenia metod niedokładnego dopasowania

W klasycznej metodzie można rozważać zrównoleglenie poprzez liczenie tablicy D przez wiele procesorów. W architekturze ze współdzieloną pamięcią i synchronizacją wątków (takiej jak na GPGPU) można to zrobić przypisując wątkom komórki na przekątnej prowadzącej z prawego-górnego rogu do lewego-dolnego. W każdej iteracji obliczamy równolegle jedną przekątną. W architekturze rozproszonej można podzielić tabelę wzdłuż kolumn, przypisując każdemu procesorowi do wypełnienia spójny fragment tabeli. Liczenie odbywa się wtedy w rzędach. Gdy procesor dotrze do końca swojego wiersza, wtedy przesyła tę wartość do procesora obliczającego sąsiedni blok. Dzięki tej wartości sąsiedni procesor może policzyć kontynuację tego wiersza, podczas gdy pierwszy procesor przechodzi już do następnego wiersza. Aby poznać możliwe pozycje dopasowania, na końcu każdy procesor odczytuje wartość w ostatnim wierszu. W kontekście dopasowywania krótkich odczytów taka metoda jest ciekawa, gdyż pozwala na podzielenie długiego genomu referencyjnego pomiędzy wiele procesorów, co oszczędza pamięć.

Odnosząc metodę z wyszukiwaniem za pomocą automatu do głównego problemu, można dopasowanie zrównoleglić poprzez podzielenie wzorców pomiędzy wiele maszyn. Każda maszyna tworzy wtedy automat dla swoich wzorców. Każda maszyna przetwarza cały tekst, uruchamiając go na swoich automatach. Trzeba przyznać, że podejście z automatem, choć ciekawe, w tym wypadku jest jednak niepraktyczne. W naszym problemie mamy bowiem bardzo długi tekst, który uruchamialibyśmy na każdym automacie, co nawet przy rozproszeniu daje bardzo długi czas.

W metodzie z liczeniem wzdłuż przekątnych występuje podobny problem co wcześniej – tekst, czyli cały genom musiałby zostać przesłany do każdej maszyny i dla każdego wzorca trzeba przetworzyć go całego w trakcie oblicznia dopasowania. To zaś prowadzi do bardzo dużej złożoności (długość genomu jest bardzo duża i ilośc wzorców również).

W metodzie z BWT ciężko zrównoleglić ją w środowisku rozproszonym, gdyż dla jednego dopasowania pytania o wartości tablicy O tyczą się różnych pozycji. Podzielenie tej tablicy pomiędzy procesory wiązałoby się z ciągłą komunikacją pomiędzy nimi. Najprędzej więc trzeba by podzielić pomiędzy procesory zbiór krótkich odczytów, natomiast strukturę dla genomu trzeba by zduplikować.

Na koniec tych rozważań należy dodać dwa ogólne schematy zrównoleglania metod dla problemu dopasowywania krótkich sekwencji DNA. Jeśli mamy metodę, którą możemy zastosować do genomu i zbioru krótkich odczytów możemy trywialnie zrównoleglić ją poprzez:

- podział zbioru odczytów pomiędzy procesory i duplikację genomu,
- podział genomu pomiędzy procesory i duplikację zbioru odczytów.

W pierwszym wypadku postępujemy tak, że każda maszyna przeprowadza dopasowanie dla swojego zbioru krótkich odczytów, po czym łączymy wyniki z wielu maszyn. W drugim wypadku każdy krótki odczyt jest dopasowywany na każdej maszynie do umieszczonego tam fragmentu tekstu. Następnie wyniki z wszystkich maszyn muszą zostać połączone. W tej metodzie trzeba jeszcze zadbać o wyszukiwanie dopasowania na łączeniu dwóch fragmentów – najłatwiej zrobić to przez podzielenie tekstu na zazębiające się fragmenty. Obydwie metody oszczędzają niewiele pamięci, ale prowadzą do zrównoleglenia obliczeń. Odbywa się to jednak kosztem czasu komunikacji – rozesłania danych, bądź odczytania ich z dysku przez każdy procesor.

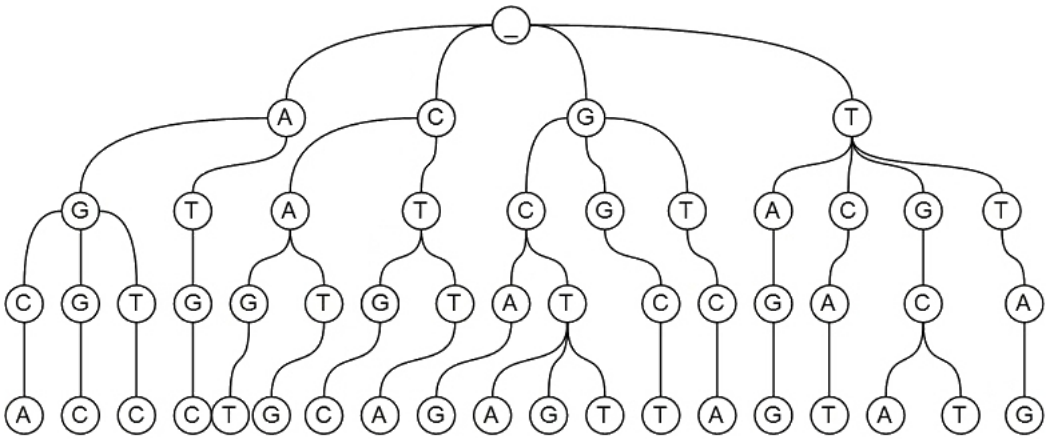
3 Propozycja własnej metody

W tym rozdziale przedstawiona jest własna metoda rozwiązania problemu niedokładnego dopasowania.

3.1 Opis metody

Russek i inni [8] zaproponowali własne podejście, w którym możliwe było użycie układów FPGA (ang. *Field-Programmable Gate Array*) do przyspieszenia obliczeń. Ich metoda stała się inspiracją do niniejszej pracy.

Korzystając z tego, że długość krótkich odczytów jest ograniczona z góry (oznaczmy to ograniczenie przez *max_length*) i jest stosunkowo mała, możemy z tekstu odczytać wszystkie spójne podciągi długości *max_length* i zbudować z nich drzewo *trie*. Nazwijmy takie drzewo *ss_trie* (od *subsequences trie*).



Rysunek 3.1: Drzewo *ss_trie* dla $max\_length = 4$

Dla zobrazowania, rysunek 3.1 przedstawia drzewo *ss_trie* jakie otrzymamy dla tekstu *AGCATGCTGCAGTCATGCTTAGGCTA*. W tym przypadku wszystkie podciągi długości 4 to zbiór: {*AGCA, GCAT, CATG, ATGC, TGCT, GCTG, CTGC, TGCA, GCAG, CAGT, AGTC, GTCA, TCAT, CATG, ATGC, TGCT, GCTT, CTTA, TTAG, TAGG, AGGC, GGCT, GCTA*}.

Mając takie drzewo, możemy w tekście wyszukiwać niedokładne dopasowanie wzorca za pomocą backtrackingu na drzewie *ss_trie*, w bardzo podobny sposób do opisanej w paragrafie 2.2.4 metody wykorzystującej backtracking i BWT. Sposób wyszukania przedstawia algorytm 3.1.

W algorytmie 3.1 słowo p to wzorzec, *pos* to aktualnie dopasowywana pozycja z wzorca ($p_{1...pos-1}$ jest już przetworzone), *cur* to wierzchołek drzewa *ss_trie*,

Algorytm 3.1 Niedokładne dopasowanie z użyciem *ss_trie*

```
1: procedure MismatchRec(p, pos, cur, k)
2:     if cur is NULL then return ∅
3:     end if
4:     if pos == |p| + 1 then return SS(cur)
5:     end if
6:     R ← MismatchRec(p, pos + 1, next(cur, p_pos), k)
7:     if k > 0 then
8:         for all x ∈ {A, C, G, T}, x ≠ p_pos do
9:             R ← R ∪ MismatchRec(p, pos + 1, next(cur, x), k − 1)
10:        end for
11:        for all x ∈ {A, C, G, T} do
12:            R ← R ∪ MismatchRec(p, pos, next(cur, x), k − 1)
13:        end for
14:        R ← R ∪ MismatchRec(p, pos + 1, cur, k − 1)
15:    end if
16:    return R
17: end procedure
```

w którym aktualnie się znajdujemy, k to dopuszczalna ilość błędów jakie jeszcze możemy wykonać. Funkcja $next(cur, x)$ to funkcja przejścia w drzewie *ss_trie*, czyli wierzchołek w jakim znajdziemy się przechodząc literą x ze stanu cur. Jeśli przejście prowadzi do stanu, którego nie ma w drzewie, to zwracane jest $NULL$. Wynikiem zwracanym przez $MismatchRec$ jest zbiór wszystkich pozycji na których możliwe jest niedokładne dopasowanie wzorca. Funkcja $SS(cur)$ zwraca pozycje podciągów genomu, które przechodzą przez aktualny wierzchołek cur.

Aby metoda wyszukiwała wszystkie możliwe dopasowania trzeba uwzględnić przypadki brzegowe. W wyniku operacji wstawiania litery, krótki odczyt może dopasować się do podciągu genomu dłuższego niż max_length. Z kolei przy bardzo krótkich odczytach oraz usunięciach litery, dopasowania mogą wystąpić na ostatnich fragmentach genomu, krótszych niż max_length. Stąd drzewo *ss_trie* należy skonstruować dla podciągów długości $max\_length + error\_num$, gdzie $error\_num$ to liczba dopuszczanych błędów w danym wyszukiwaniu (lub górne ograniczenie tej liczby). Aby uwzględnić dopasowania na samym końcu genomu odczytujemy podciągi z wszystkich pozycji (od 1 do n, gdzie n to długość genomu), przy czym ostatnie podciągi są krótsze niż $max\_length + error\_num$

(kończą się wraz z końcem genomu). Oznacza to, że w drzewie *ss_trie* nie tylko liście odpowiadają podciągom – podciągi krótsze niż głębokość drzewa kończą się na wcześniejszych poziomach. Struktura drzewa musi uwzględniać taki przypadek.

3.2 Zastosowanie do problemu dopasowania krótkich sekwencji DNA

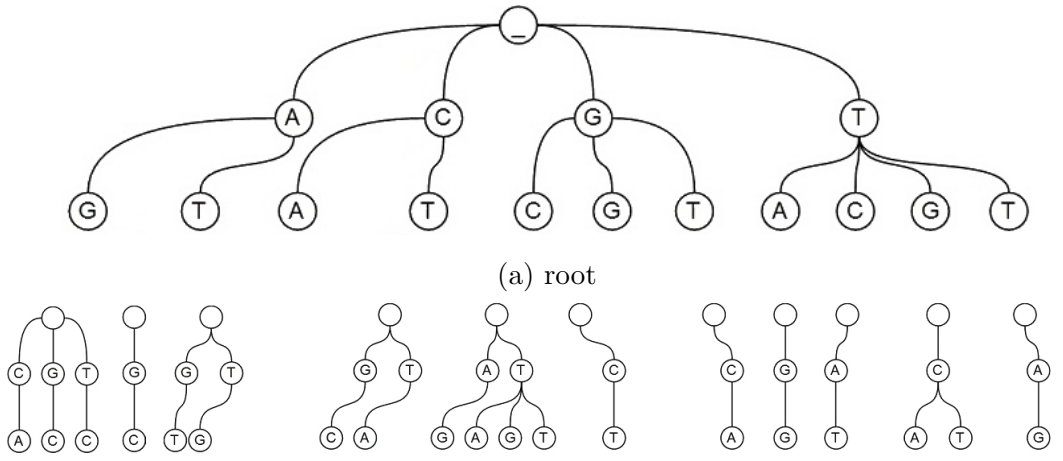
Opisana w poprzednim paragrafie metoda stosuje się wprost do problemu dopasowania krótkich sekwencji DNA. Dla genomu referencyjnego tworzy się drzewo *ss_trie*. Następnie po kolei dla każdego wzorca wywołujemy *MismatchRec*.

3.3 Sposób zrównoleglenia

W modelu wielu procesorów z współdzieloną pamięcią problem można zrównoleglić tak, by każdy procesor przetwarzał przypisaną do siebie część krótkich odczytów. W modelu z brakiem współdzielonej pamięci genom może zostać powielony tak, by każdy procesor zbudował swoją kopię drzewa *ss_trie*, po czym można podzielić zbiór krótkich odczytów do dopasowania pomiędzy procesory. Nie jest to jednak zrównoleglenie jakie by nas satysfakcjonowało, gdyż każdy procesor musi otrzymać cały genom, co nie oszczędza ani czasu konstrukcji drzewa ani ilości pamięci potrzebnej jednemu procesorowi.

Aby dokonać efektywnego zrównoleglenia dzielimy drzewo pomiędzy procesory. W tym celu wprowadzamy parametr *root_depth*. Pierwsze *root_depth* poziomów zostawiamy na pierwszym procesorze (*root*) natomiast pozostałe fragmenty dzielimy pomiędzy resztę procesorów (*worker*). W efekcie takiego podziału powstaje rozproszone drzewo *ss_trie*, w którym liście drzewa na głównym procesorze są korzeniami poddrzew rozdzielonych między pozostałe procesory. Przykładowy podział drzewa przy czterech procesorach worker został przedstawiony na rysunku 3.2.

Dopasowanie z użyciem takiego rozproszonego drzewa przebiega następująco. Dla pojedynczego odczytu root wykonuje normalnie algorytm 3.1, ale w momencie dotarcia do liścia, zapamiętuje numer poddrzewa do jakiego ma przejść oraz z jakimi parametrami tam dotarł (liczba już dopasowanych znaków, liczba pozostałych błędów). Każde takie dotarcie do liścia oznacza zlecenie dokończenia przetwarzania workerowi, który przechowuje odpowiednie poddrzewo. Kiedy więc



(a) root

(b) worker 1 (c) worker 2 (d) worker 3 (e) worker 4

Rysunek 3.2: Podział drzewa *ss_trie* przy $root\_depth = 2$

root spamięta wszystkie zlecenia dotyczące danego odczytu, rozsyła je po kolei do workerów, po czym przechodzi do następnego odczytu. Dla każdego odczytu przesyłany jest identyfikator, zawartość oraz lista zleceń dla danej maszyny w postaci trójki parametrów: numer drzewa od którego należy kontynuować wyszukiwanie, liczba już dopasowanych znaków, liczba pozostałych błędów. Odebrawszy listę zleceń, worker wywołuje dla każdego zlecenia procedurę $MismatchRec$ z przesłanymi parametrami oraz z $pos = root\_depth$. W ten sposób worker kontynuuje obliczenia, które zostałyby wykonane w wersji sekwencyjnej w momencie dotarcia do wierzchołka odpowiadającego liściowi na procesorze root. Po przetworzeniu wszystkich odczytów, każdy worker odsyła wyniki do głównego procesora, ten zaś zbiera i porządkuje je.

Kluczowy dla dobrego zrównoleglenia obliczeń jest parametr *root_depth* i sposób przesyłania zleceń. Ilość procesorów na które można zrównoleglić problem jest zależna od *root_depth* i jest ograniczona przez 4^{root_depth}. Już przy $root\_depth = 4$ i długim genomie, daje nam to możliwość uruchomienia zadania na ok. 250 procesorach. Z drugiej strony, jeśli *root_depth* będzie za duże, wtedy czas potrzebny rootowi na przetworzenie odczytu będzie długi i procesory worker będą czekać na otrzymanie zlecenia. Z kolei przesyłanie zleceń po jednym odczycie może prowadzić do nierówniemiernego obciążenia workerów. Przypuszczalnie najlepsze efekty osiągnie przetwarzania krótkich odczytów niedużymi paczkami. Wtedy po przetworzeniu przez roota paczki krótkich odczytów, wszystkie zlecenia z paczki zostaną przesłane do workerów, co powinno średnio prowadzić do równomiernego obciążenia. W momencie przetwarzania paczki zleceń przez workery, root prze-

twarza następną paczkę. Rozmiary paczek i wartość *root_depth* trzeba dobrać na podstawie testów, obserwując w jakiej konfiguracji uzyskuje się najlepsze wyniki.

Taka metoda prowadzi do dobrego zrównoleglenia. Pamięć jest podzielona, gdyż każdy procesor odpowiada za inny fragment drzewa. Sumaryczny czas obliczeń wszystkich procesorów jest taki sam jak w wersji sekwencyjnej, ale obliczenia są wykonywane równolegle przez kilka procesorów, zostaną więc wykonane w krótszym czasie. Ceną za to zrównoleglenie jest czas komunikacji: rozsyłania krótkich odczytów i parametrów wywołań. Dane są powielane, gdyż jeden krótki odczyt może trafić do kilku procesorów. Wydaje się jednak, że samo dopasowanie trwa na tyle długo, że czas komunikacji będzie niewielki w porównaniu do zysku jaki daje zrównoleglenie.

W przedstawionej metodzie, aby poprawić komunikację pomiędzy procesorami i zlikwidować ryzyko niepotrzebnych przestojów, nasuwa się użycie dwóch wątków na procesor. Jeden wątek byłby odpowiedzialny za komunikację, to jest przesyłanie i odbieranie danych, drugi za obliczenia. Przy takim rozdziale, w momencie dotarcia do liścia, root automatycznie zlecałby przesłanie zadania do odpowiedniego workera. Wtedy wątek odpowiedzialny za komunikację przesłałby zlecenie, w trakcie gdy wątek obliczeniowy kontynuowałby rekurencyjne wywołania. W przypadku workerów byłoby odwrotnie – wątek obliczeniowy czekałby na wykonanie zlecenia, które dostarczałby wątek odpowiedzialny za komunikację. Takie rozwiązanie wymagałoby dodatkowej kolejki na zlecenia pomiędzy dwoma wątkami. Jest to klasyczny schemat producenta (wątek obliczeniowy roota, wątek komunikacji workera) i konsumenta (wątek komunikacji roota, watek obliczeniowy workera).

Inne usprawnienie można byłoby wprowadzić jeśli wartość *root_depth* byłaby nieduża. Wtedy poszczególne prefiksy przetwarzane przez roota występowałyby wiele razy. Można byłoby więc dokonać spamiętania jakie zlecenia generuje który prefiks. W przypadku niedużej wartości *root_depth* można byłoby nawet dokonać preprocessingu wszystkich możliwych prefiksów i spamiętać je wszystkie. Wtedy dla danego krótkiego odczytu, działanie roota ograniczałoby się do odczytania z tabeli jakie zlecenia generuje prefiks tego krótkiego odczytu i do rozesłania zleceń odpowiednim workerom.

3.4 Uproszczenie metody

Jeśli w wyjściowym problemie dopuści się wprowadzenie pewnego ograniczenia, wtedy problem daje się dużo wygodniej i efektywniej zrównoleglić. Załóżmy, że każdy krótki odczyt musi być dopasowany dokładnie na pierwszych kilku znakach. Parametr określający ilość tych znaków oznaczmy przez $exact\_prefix$. Wprowadzenie takiego parametru do algorytmu $MismatchRec$ jest bardzo proste, wystarczy w linii 7 sprawdzać warunek $if\ (k > 0)\ and\ (pos > exact\_prefix)$.

Na podstawie parametru $exact\_prefix$ dokonujemy zrównoleglenia całego przetwarzania. Budujemy drzewo ss_trie w sposób rozproszony, tak jak w głównej metodzie, zostawiając na pierwszym procesorze pierwsze $exact\_prefix$ poziomów drzewa. Dopasowanie krótkich odczytów jest prostsze niż poprzednio. Każdy krótki odczyt przechodzi przez fragment drzewa na głównym procesorze (szczyt ss_trie) i trafia do któregoś poddrzewa. Z racji na dokładne dopasowanie, przejście to jest bardzo szybkie. Następnie krótki odczyt zostaje wysłany do workera, który zawiera to poddrzewo i tam wywoływana jest procedura $MismatchRec$, z $pos = exact\_prefix$. Przesyłamy tylko identyfikator krótkiego odczytu i jego zawartość. Nie trzeba przesyłać żadnych dodatkowych parametrów.

Taka metoda daje nam efektywne zrównoleglenie: procesory dzielą pomiędzy siebie dane, każdy procesor przechowuje w dostępnej dla siebie pamięci inny fragment ss_trie oraz każdy procesor otrzymuje do dopasowania pewien podzbiór wszystkich krótkich odczytów (jeden krótki odczyt trafia tylko do jednego workera). Dopasowanie dokładne jest na tyle szybkie, że możemy w pierwszej kolejności dokonać podziału całego zbioru krótkich odczytów na workery, następnie przesłać wszystkie zadania, a potem czekać już tylko na wyniki.

4 Implementacja

Metoda opisana w poprzednim rozdziale została zaimplementowana w uproszczonej wersji. Jako język programowania został wybrany C++11. Do zrównoleglania wykorzystano MPI (*Message Passing Interface*). Oprócz głównej metody (rozproszone drzewo *ss_trie*) zaimplementowano sekwencyjną wersję *ss_trie*, zrównoleglanie poprzez duplikację pełnego drzewa oraz generator genomów i krótkich odczytów. Kod źródłowy jest dostępny w repozytorium mieszczącym się pod adresem `https://bitbucket.org/andrzejdoro/pda/`. Testowana wersja została oznakowana tagiem `msc`.

4.1 Opis architektury

Przyjęto architekturę z wieloma maszynami bez współdzielenia pamięci. Pierwsza maszyna jest wyszczególniona (root) i zarządza całym przetwarzaniem. Pozostałe (workers) nie różnią się od siebie i wykonują zlecone przez główną maszynę obliczenia. Przyjęto, że tylko główna maszyna ma dostęp do pliku z danymi wejściowymi, wczytuje je z dysku i rozsyła potrzebne dane do pozostałych maszyn. W efekcie działania programu pierwsza maszyna zapisuje na dysku plik z wynikami.

W praktyce program uruchamiany jest w wielu procesach MPI. W zależności od konfiguracji MPI, ilości fizycznych maszyn oraz ilości fizycznych procesorów i rdzeni w które są wyposażone, niektóre procesy mogą być wykonywane przez tę samą fizyczną maszynę. Aby osiągnąć najlepszą wydajność, każdemu procesowi powinien odpowiadać jeden rdzeń obliczeniowy. Z racji na to, że kontrola nad miejscem uruchomienia każdego procesu jest pozostawiona konfiguracji MPI i w kodzie w żaden sposób nie jest wykorzystywany fakt czy dane procesy są uruchomione na tej samej maszynie czy nie, w dalszej części pracy będziemy używać słowa maszyna do określenia jednego procesu MPI.

4.2 Struktura plików wejściowych i wyjściowych

Wejściem do programu są dwa pliki: plik z genomem oraz plik z krótkimi odczytami. Przyjęto możliwie prosty format plików wejściowych.

Plik z genomem zawiera na początku jedną liczbę n – długość genomu referencyjnego, a następnie w drugiej linii znajduje się zapis genomu (ciąg długości n,

liter ze zbioru $\{A, C, G, T\}$). Na początku pliku mogą być umieszczone komentarze (linie rozpoczynające się od "//").

Plik z krótkimi odczytami zawiera na początku jedną liczbę m – ilość krótkich odczytów. W kolejnych m liniach znajdują się zapisy kolejnych krótkich odczytów (jedna linia – jeden odczyt). Komentarze mogą być umieszczone zarówno na początku pliku jak i na końcu każdej linii.

Wyjściem programu jest jeden plik z zapisanymi dopasowaniami dla każdego krótkiego odczytu. Zawiera on m linii, każda odpowiada jednemu krótkiemu odczytowi, w takiej kolejności jak w pliku wejściowym. Zapis dopasowania, to wyszczególnienie wszystkich pozycji na jakich można dopasować dany krótki odczyt.

4.3 Parametry programu

Uruchamiając program podaje się następujące parametry:

- nazwa pliku z genomem referencyjnym,
- nazwa pliku z sekwencjami krótkich odczytów,
- długość podciągów dla których ma zostać zbudowane drzewo trie,
- liczba możliwych błędów w dopasowaniu,
- długość prefiksu, która ma zostać dopasowana dokładnie,
- nazwa pliku do zapisania wyniku,
- nazwa pliku do zapisania informacji o wykonaniu.

Obowiązkowe są tylko parametry z nazwą pliku genomu referencyjnego i nazwą pliku z sekwencjami krótkich odczytów, pozostałe posiadają swoje domyślne wartości.

4.4 Root

Działanie głównej maszyny jest następujące. Najpierw wczytywane są dane (genom referencyjny oraz krótkie odczyty). Fragmentowi DNA odpowiada struktura *genome*. Następnie zostaje zbudowany szczyt drzewa *ss_trie* o głębokości $exact\_prefix$. Drzewo to przechowywane jest w strukturze *dispatch_ss_trie*. Drzewo to będzie służyło do dzielenia zadań (dopasowań krótkich odczytów) pomiędzy pozostałe maszyny. Na podstawie zbudowanego drzewa i ilości dostępnych maszyn zostaje stworzony *sub_tree_dispatcher*, który jest odpowiedzialny za przypisanie, który fragment drzewa trafia do którego workera. Zaraz po tym

następuje przesłanie odpowiednich podciągów do każdego z workerów. Gdy ta faza dobiegnie końca, całość jest gotowa do przetwarzania krótkich odczytów.

W drugiej fazie root przepuszcza wszystkie krótkie odczyty przez drzewo *dispatch_ss_trie*, dokonując podziału zbioru odczytów – który odczyt trafi do którego workera. Mając gotowy podział zadań, krótkie odczyty zostają przesłane do pozostałych maszyn. Rozpoczyna się dopasowanie krótkich odczytów równolegle na wszystkich maszynach.

W trzeciej fazie root oczekuje na wyniki od wszystkich maszyn. Gdy otrzyma wszystkie wyniki, wtedy zapisuje je do pliku, zgodnie z kolejnością w jakiej krótkie odczyty występowały w pliku wejściowym. Na samym końcu odbywa się czyszczenie zasobów (zwalnianie pamięci) i zapis informacji o przebiegu programu do pliku z logiem.

4.5 Worker

Działanie pojedynczej maszyny przetwarzającej jest następujące. Na początku worker czeka na dane potrzebne do skonstruowania swoich fragmentów rozproszonego drzewa *ss_trie*. Gdy je otrzyma wtedy dla każdego fragmentu tworzy obiekt typu *part_genome_ss_trie*.

W drugiej fazie worker czeka na otrzymanie zadań do przetwarzania. Gdy odbierze wszystkie przypisane mu krótkie odczyty, wtedy po kolei dopasowuje każdy z nich, wywołując się na odpowiednim *part_genome_ss_trie*.

W trzeciej fazie, po przetworzeniu wszystkich zadań, worker przesyła do głównej maszyny efekty dopasowania w postaci listy pozycji, na które można dopasować dany odczyt. Na samym końcu odbywa się czyszczenie zasobów i zapis informacji o przebiegu programu do pliku z logiem (każdy worker ma swój plik).

4.6 Opis użytych struktur

Do przechowywania danych w trakcie działania programu użyto kilku samodzielnie zaimplementowanych struktur.

4.6.1 genome

Struktura do przechowywania fragmentów DNA. Przechowuje ciąg znaków odpowiadający fragmentowi DNA oraz jego długość. Znaki nie są przechowywane jako A, C, G, T, ale jako liczby 0, 1, 2, 3.

4.6.2 dispatch_ss_trie

Jest to szczyt rozproszonego drzewa *ss_trie*, służy zarówno do podziału podciągów genomu na pozostałe maszyny jak i do przydzielania który odczyt jest dopasowywany na której maszynie.

Wierzchołki tego drzewa przechowują tablicę $next[4]$ trzymającą wskaźniki do swoich dzieci, w zależności od pójścia literą A, C, G lub T. Oprócz tego wierzchołki przechowują wartość *tree_num*, która jest wypełniona tylko dla liści tego drzewa. Wartość ta, to numer poddrzewa do jakiego trafiają podciągi i krótkie odczyty, mające prefiks odpowiadający danemu liściowi (inaczej mówiąc, to wartość, która określa do jakiego drzewa należy przejść, jeśli przechodząc przez drzewo dotarło się do tego liścia).

Numery *tree_num*, są nadawane w momencie konstrukcji drzewa z wszystkich podciągów. Wtedy też zostaje zapamiętany podział, który podciąg trafia do którego poddrzewa. Ta informacja jest przechowywana w polu *dispatched_ss*.

4.6.3 sub_tree_dispatcher

Obiekt odpowiedzialny za podział poddrzew rozproszonego *ss_trie* pomiędzy maszyny worker. W trakcie konstrukcji przyjmuje liczbę maszyn worker i podział podciągów na poddrzewa. Na tej podstawie dokonuje przypisania, które poddrzewa trafiają do której maszyny.

Jako, że każdy worker numeruje drzewa, które otrzymał niezależnie, począwszy od 0, to właśnie ten obiekt jest odpowiedzialny za tłumaczenie numeru poddrzewa na numer maszyny worker i numer drzewa w obrębie tej maszyny.

4.6.4 part_genome_ss_trie

Jest to struktura odpowiadająca poddrzewu w rozproszonym *ss_trie*. Każdy worker przechowuje listę takich drzew – każde odpowiada jednemu poddrzewu przypisanemu tej maszynie przez obiekt *sub_tree_dispatcher*. Do stworzenia takiego drzewa worker wykorzystuje otrzymane fragmenty podciągów głównego genomu, wraz z informacją który to podciąg (indeks pozycji startowej w całym genomie referencyjnym).

Wierzchołki tego drzewa przechowują tablicę $next[4]$ trzymającą wskaźniki do swoich dzieci, w zależności od pójścia literą A, C, G lub T. Oprócz tego wierzchołki przechowują dwie dodatkowe wartości $left$ i $right$ oznaczające przedział

podciągów, dla których słowo reprezentowane przez dany wierzchołek jest prefiksem. Możemy mówić o przedziale, gdyż w momencie tworzenia drzewa sortujemy podciągi w kolejności leksykograficznej, co daje nam, że każdy prefiks odpowiada spójnemu fragmentowi podciągów. W tym celu drzewo przechowuje również tablicę z posortowanymi indeksami podciągów, która umożliwia odczytanie, do których dokładnie podciągów odnosi się dany wierzchołek.

4.6.5 part_ss_trie_aligner

Każde drzewo *part_genome_ss_trie* posiada powiązany ze sobą jeden obiekt typu *part_ss_trie_aligner*, który jest odpowiedzialny za wyszukiwanie wszystkich pozycji gdzie może występować dany krótki odczyt.

To właśnie ten obiekt jest wyposażony w funkcję *mismatch_align*, która wywołuje algorytm zaprezentowany w sekcji 3.1.

5 Analiza wyników

Opisana w poprzednim rozdziale implementacja została przetestowana. Niniejszy rozdział przedstawia wykonane testy i dokonane pomiary czasu działania wraz z analizą wyników.

5.1 Środowisko testowe

W ramach pracy przetestowano implementację rozproszonego drzewa *ss_trie* oraz implementację drzewa *ss_trie* dla jednego procesora, zestawiając ze sobą otrzymane wyniki. Testy wykonano na klastrze obliczeniowym ZEUS, udostępnianym przez Akademickie Centrum Komputerowe "CYFRONET". Oferuje on 25 tysięcy rdzeni obliczeniowych, zgrupowanych w węzły przeważnie po 12 rdzeni [9].

Przedstawione testy zostały wykonane na węzłach, wyposażonych w procesory Intel o prędkości 2,4 GHz (model E5645). Dane testowe zostały stworzone w sposób losowy, za pomocą napisanego generatora. Dopasowywano krótkie odczyty pobrane z genomu referencyjnego do tegoż genomu.

Kod źródłowy był kompilowany poleceniem `g++ -std=c++11` dla wersji sekwencyjnej oraz `mpic++ -std=c++11` dla wersji rozproszonej. Program w wersji rozproszonej był uruchamiany poleceniem `mpiexec` z podaniem odpowiedniej ilości procesów MPI do uruchomienia. Rezerwowano zawsze minimum $W+1$ rdzeni, gdzie W to liczba maszyn (procesów) typu worker w danym uruchomieniu.

5.2 Pomiary

Na potrzeby pomiarów przygotowano skrypt, który przeprowadza wszystkie testy, generując potrzebne dane i uruchamiając kolejno programy z odpowiednimi parametrami. Testy wykonano kilkukrotnie, pomiary pomiędzy różnymi uruchomieniami były porównywalne. Do celów prezentacji wybrano wyniki testów najbardziej oddających średnią lub średnią z wykonań w kilku testach.

Ogólne parametry dla testów, jeśli nie zaznaczono inaczej, były następujące:

- przedział krótkich odczytów, 32-100
- maksymalna długość podciągu (głębokość *ss_trie*), 100
- długość dokładnego prefiksu, 3.

Wyniki testów są podane w sekundach.

5.2.1 Czas obsługi genomu

W tym teście mierzony był czas preprocessingu genomu referencyjnego, czyli czas budowy drzewa *ss_trie* oraz *part_genome_ss_trie* odpowiednio w wersji sekwencyjnej i w wersji rozproszonej. Celem zmierzenia zależności wykonano testy ze zmieniającym się parametrem długości genomu, przy niezmienianiu innych parametrów. Ilość krótkich odczytów ustawiono na 0 aby czas dopasowywania nie wpływał na czas działania programu.

Tabela 5.1: Czas konstrukcji drzewa (w sekundach)

długość genomu	**10000**	**20000**	**50000**	**100000**	**1000000**
ss-trie	0.6090	1.2181	3.0738	6.0874	60.8781
dispatch-trie	0.0022	0.0040	0.0094	0.0181	0.1935
distributed W=1	0.8096	1.5586	4.0049	8.2916	111.6763
distributed W=2	0.4072	0.8179	1.9827	4.7564	57.7716
distributed W=4	0.2139	0.4240	1.0589	2.1682	26.6293

W tabeli 5.1 wiersz ss-trie to czas budowy *ss_trie* w wersji sekwencyjnej, dispatch-trie to czas budowy fragmentu drzewa przez główną maszynę (root), wiersze distributed odnoszą się do budowy rozproszonego drzewa na pozostałych maszynach (worker), gdzie W oznacza liczbę maszyn (procesów MPI) typu worker. W wersji rozproszonej, spośród czasów wszystkich maszyn wybierano najdłuższy czas.

Tabela 5.2: Czas wczytywania i rozsyłania genomu [s]

długość genomu	**10000**	**20000**	**50000**	**100000**	**1000000**
wczytanie	0.0009	0.0010	0.0016	0.0024	0.0217
wysłanie W=1	0.0263	0.0538	0.1312	0.2667	2.3386
wysłanie W=2	0.0185	0.0326	0.1348	0.1373	1.9709
wysłanie W=4	0.0268	0.0537	0.2694	0.2489	2.1734

W tabeli 5.2 wczytanie to czas wczytywania genomu z dysku do pamięci. Wysłanie to sumaryczny czas przesyłania przez główną maszynę fragmentów genomu, potrzebnych do zbudowania poddrzew rozproszonego *ss_trie* na pozostałych maszynach.

Tabela 5.3 przedstawia czas działania całego programu w wersji sekwencyjnej (ss-trie) i wersji rozproszonej.

Tabela 5.3: Czas obsługi genomu [s]

długość genomu	**10000**	**20000**	**50000**	**100000**	**1000000**
ss-trie	0.6116	1.2209	3.0770	6.0926	60.9089
W=1	0.8457	1.6240	4.1713	8.5879	114.2553
W=2	0.4339	0.8598	2.1348	4.9200	60.0036
W=4	0.2408	0.4759	1.3434	2.3573	28.4670

5.2.2 Czas dopasowywania krótkich odczytów

W tym teście, aby zbadać zależność czasu dopasowywania krótkich odczytów od ilości dopuszczanych błędów, zmieniany był tylko parametr dopuszczalnej ilości błędów. Test był wykonywany przy parametrach wejścia:

- długość genomu, 100 000
- liczba krótkich odczytów, 2000.

Tabela 5.4: Czas niedokładnego dopasowania [s]

M	**0**	**1**	**2**	**3**	**4**
ss-trie	0.0092	0.0440	0.4745	3.3516	34.4662
distributed W=1	0.0132	0.0522	0.4688	3.4172	36.6059
distributed W=2	0.0071	0.0234	0.2457	1.7311	17.9924
distributed W=4	0.0033	0.0122	0.1263	0.9084	9.7550

W tabeli 5.4 wartość M to ilość dopuszczonych błędów w dopasowaniu w danym teście. Testy wykonano podobnie jak wcześniej dla wersji sekwencyjnej oraz dla wersji rozproszonej z 1, 2 i 4 maszynami przetwarzającymi.

Tabela 5.5: Przetwarzanie krótkich odczytów przez maszynę root [s]

podział SR	0.0004
wysłanie SR, W=1	0.0115
wysłanie SR, W=2	0.0096
wysłanie SR, W=4	0.0087

W tabeli 5.5 przedstawiony jest czas podziału zbioru krótkich odczytów (SR) jaki dokonywany jest na maszynie root (dopasowanie dokładne na prefiksie) oraz czas przesyłania do jednej maszyny worker przypisanych jej do przetworzenia krótkich odczytów.

Tabela 5.6: Średni czas przesyłania i zapisywania wyników [s]

wysłanie wyników W=1	0.0068
wysłanie wyników W=2	0.0060
wysłanie wyników W=4	0.0043
zapis wyników	0.0046

W tabeli 5.6 przedstawiony jest średni czas przesyłania wyników przez jedną maszynę worker do głównej maszyny root oraz czas zapisu pliku z wynikami na dysku przez maszynę root.

Tabela 5.7: Czas całości procesu dopasowania [s]

M	**0**	**1**	**2**	**3**	**4**
ss-trie	6.1216	6.1394	6.6123	9.9825	40.6099
W=1	9.7202	9.4948	9.2150	12.4659	45.7143
W=2	4.5516	4.6187	4.8488	6.2812	22.4932
W=4	2.4298	2.4039	2.5494	3.2053	11.9713

Czas działania całego procesu dopasowywania, z wgraniem genomu i budową drzewa *ss_trie*, jest przedstawiony w tabeli 5.7. Średni czas budowy drzewa *ss_trie* w tych testach wyniósł odpowiednio: 6.2074 (dla wersji sekwencyjnej), 8.93884 (wersja rozproszona, W=1), 4.26434 (W=2), 2.12895 (W=4).

5.2.3 Testy na kilku węzłach

Z racji na dostępność 12 rdzeni na jednym węźle klastra, przedstawione testy były uruchamiane tylko na jednym węźle. Aby otrzymać bardziej wiarygodne wyniki dotyczące czasu komunikacji, zostały wykonane testy z większą liczbą rdzeni, umieszczonych na kilku węzłach. Było to odpowiednio 10 rdzeni typu worker (1 węzeł), 20 rdzeni (2 węzły), 30 rdzeni (3 węzły).

Tabela 5.8: Czas obsługi genomu [s]

	50000	**100000**	**1000000**
W=10	0.5623	1.0487	10.4532
W=20	0.5787	0.8946	6.6454
W=30	0.8191	1.0266	5.5014

Tabela 5.8 przedstawia test obsługi samego genomu, taki jak w paragrafie 5.2.1, czyli jest to czas wykonania całości programu przy pustym zbiorze krótkich odczytów. Pominięto małe testy dla genomu długości 10000 i 20000.

Tabela 5.9: Czas rozsyłania genomu [s]

	50000	**100000**	**1000000**
W=10	0.1344	0.2353	2.0585
W=20	0.3514	0.4531	2.5529
W=30	0.5760	0.7376	2.7780

Czas komunikacji pomiędzy procesami w teście obsługi samego genomu przedstawia tabela 5.9. Jest to czas wysyłania podciągów genomu do maszyn worker.

Tabela 5.10: Czas niedokładnego dopasowania [s]

M	**3**	**4**	**5**
W=10	1.4670	5.5265	30.7280
W=20	1.0697	3.2101	17.9343
W=30	1.0965	2.7122	14.1153

Tabela 5.10 przedstawia wyniki drugiego testu – dopasowania zbioru 2000 krótkich odczytów do genomu długości 100000 (patrz paragraf 5.2.2). Z racji na większą moc obliczeniową dopuszczono możliwość 3, 4 i 5 błędów.

Tabela 5.11: Średni i maksymalny czas dopasowania przez maszyny worker [s]

	avg, 3	**max, 3**	**avg, 4**	**max, 4**	**avg, 5**	**max, 5**
W=10	0.3234	0.3968	3.5993	4.4508	23.8115	29.6555
W=20	0.1582	0.2323	1.7727	2.5599	11.8145	17.2361
W=30	0.1053	0.1773	1.1816	1.9916	7.8717	13.3444

Tabela 5.11 prezentuje czas samego dopasowania, wykonywanego przez maszyny worker. Kolumny z oznaczeniem avg to średni czas dopasowania wszystkich workerów, kolumna z max, to czas najdłużej dopasowującego workera. Liczbę dopuszczonych błędów podano w nagłówku kolumny.

5.3 Analiza

Testy obsługi genomu pokazały spodziewaną zależność, że **wraz ze wzrostem długości genomu, liniowo rośnie czas potrzebny na zbudowanie drzewa** *ss_trie*. Porównując czas budowy drzewa (tabela 5.1) w wersji sekwencyjnej (ss-trie) z czasem budowy prawie takiego samego drzewa przez jeden procesor w wersji rozproszonej (W=1), widać dużą różnicę na niekorzyść tego drugiego rozwiązania. Jest to interesujące, gdyż sumaryczna wielkość drzew zbudowanych w tym drugim rozwiązaniu jest minimalnie mniejsza niż drzewa zbudowanego w wersji sekwencyjnej, a sposób konstrukcji drzew jest taki sam. Różnica musi wynikać ze szczegółów implementacyjnych, tj. sposobu przechowywania podciągów, z których są budowane drzewa w wersji rozproszonej. W wersji sekwencyjnej są one bezpośrednio odczytywane z tablicy przechowującej cały genom. W wersji rozproszonej pojedyncza maszyna worker nie otrzymuje całego genomu, tylko jego fragmenty, każdy jest więc trzymany w osobnym fragmencie pamięci, z dodatkową informacją, z jakiego miejsca w genomie pochodzi.

Najważniejszą informacją, którą można wywnioskować z tych pomiarów jest fakt, że **konstrukcja drzewa dobrze się zrównolegla**. Widać, że wraz ze wzrostem liczby procesorów, proporcjonalnie spada wielkość danych do przetworzenia przez daną maszynę, co skutkuje odpowiednio mniejszym czasem konstrukcji drzewa. Jest tak zarówno przy małej jak i dużej liczbie procesorów. W przypadku testów z większą ilością procesorów, widzimy to po odjęciu czasu komunikacji (tabela 5.9) od czasu działania programu (tabela 5.8). Z kolei czas poświęcony przez główną maszynę na zbudowanie swojego fragmentu i podzielenie zadań na maszyny worker jest bardzo niewielki. Oczywiście wynika to z niskiego parametru $exact\_prefix$ – wraz z jego wzrostem rośnie wielkość drzewa do zbudowania przez główną maszynę, ale za to zyskuje się na czasie dopasowania, tego więc nie należy się obawiać.

Patrząc na pomiary czasu wczytywania i rozsyłania danych potrzebnych do zbudowania rozproszonego drzewa (tabela 5.2) widać, że czas komunikacji jest nieduży w porównaniu do czasu działania programu (tabela 5.3). Analizując czas rozsyłania genomu do procesorów umieszczonych w kilku węzłach (tabela 5.9) przy genomie długości 50000 oraz 100000 możemy odnieść wrażenie, że liczba węzłów wyraźnie wpływa na czas rozsyłania danych. Jednakże pomiary dla genomu długości 1000000 pokazują, że **liczba maszyn wpływa na czas komunikacji, ale w łagodny sposób** – nie jest to proporcjonalny wzrost. Odnosząc

czas rozsyłania genomu do czasu działania całego programu (tabela 5.8) widzimy, że wraz ze wzrostem ilości procesorów, czas rozsyłania danych coraz bardziej dominuje czas budowy drzewa. Taki efekt uzyskujemy jednak, dzięki wyraźnemu przyspieszeniu samej konstrukcji drzewa. Możemy więc wyciągnąć wniosek, że **koszt komunikacji pomiędzy maszynami jest opłacalny w porównaniu do przyspieszenia jakie daje**. Jest to bardzo ważne, pokazuje to bowiem, że zaproponowana metoda od zwyczajnego powielenia pełnego *ss_trie* zyskuje podwójnie: oszczędzana jest pamięć oraz oszczędzany jest czas konstrukcji drzewa.

Analizując pomiary czasu niedokładnego dopasowania krótkich odczytów (tabela 5.4 i 5.11) można zaobserwować, że **kluczowy dla czasu działania programu jest parametr dopuszczalnej liczby błędów**. Dopuszczenie jednego błędu więcej powoduje wzrost czasu działania o rząd wielkości, tj. 9-10 razy. Gdy się nad tym zastanowić jest to zgodne z intuicją – w funkcji $MismatchRec$ będąc w jednym stanie (wartość parametrów wywołania) oprócz dokładnego dopasowania możemy, wykorzystując jeden błąd, przejść do 8 innych stanów (patrz algorytm 3.1). Mając więc jeden więcej błąd do dyspozycji, możemy dotrzeć w drzewie *ss_trie* wszędzie tam gdzie w przypadku mniejszym i z każdego takiego stanu przejść jeszcze do 8 innych stanów. Interesującą obserwację daje też porównanie średniego czasu pracy workera z maksymalnym czasem, w teście z większą liczbą maszyn (tabela 5.11). Widać dużą rozbieżność – przypuszczalnie jest to spowodowane małym zbiorem krótkich odczytów, który przy tym teście został podzielony w nierównomierny sposób pomiędzy maszyny.

Widać, podobnie jak przy obsłudze genomu, że wersja rozproszona z jedną maszyną jest mniej wydajna niż wersja sekwencyjna, choć tym razem czasy działania są zbliżone. Podobnie jednak jak przy budowie drzewa, tak i **przy porównywaniu krótkich odczytów problem dobrze się zrównolegla i wraz ze wzrostem liczby przetwarzających procesorów, czas dopasowania proporcjonalnie spada**. Dodatkowe koszty w postaci komunikacji pomiędzy maszynami: podział zbioru krótkich odczytów pomiędzy maszyny, przesłanie danych i potem zebranie wyników (tabele 5.5, 5.6), są znikome w porównaniu do czasu potrzebnego na obliczenie dopasowania.

Patrząc na wyniki całości procesu dopasowania (tabela 5.7 i 5.10) widzimy, że przy stosunkowo niedużej liczbie krótkich odczytów, czas działania przy dopuszczalnej liczbie błędów $\leqslant 2$ jest zdominowany przez czas konstrukcji drzewa. Jednak już w przypadku $M = 3$ czas dopasowania zaczyna mieć spore znaczenie,

a przy $M = 4$ dominuje długość procesu dopasowania. Podobnie jest przy większej ilości maszyn i dopasowaniu z $M = 5$. Należy również pamiętać, że testy wykonano przy małej ilości krótkich odczytów (2000). Przy większej ich ilości, czas dopasowania będzie proporcjonalnie większy, natomiast czas budowy drzewa nie ulegnie zmianie. Wszystko to prowadzi nas do wniosku, że **w pierwszej kolejności trzeba polepszać sam algorytm dopasowania niż poprawiać szczegóły implementacji wybranej metody**.

6 Podsumowanie

W niniejszej pracy rozważaliśmy problem dopasowania krótkich sekwencji DNA. Popatrzyliśmy na to zagadnienie z punktu teoretycznego – jako na problem niedokładnego wyszukiwania wielu wzorców w tekście, przy zadanej liczbie dopuszczanych błędów i o określonych pewnych parametrach. Szukając metody pozwalającej na efektywne zrównoleglenie obliczenia tego problemu, dokonaliśmy przeglądu istniejących algorytmów wyszukiwania wzorca w tekście oraz niedokładnego dopasowania.

Przedstawione algorytmy wyszukiwania wzorca prezentują ciekawe techniki podejścia do tego samego problemu. Przechodząc jednak do szerszego problemu jakim jest problem niedokładnego dopasowania wzorca, widać, że do jego rozwiązania stosuje się inne metody, które ciężko efektywnie zrównoleglić. W kontekście wyjściowego problemu dopasowania wielu krótkich sekwencji DNA, najprostszym zrównolegleniem jest podział dopasowania krótkich odczytów pomiędzy wiele procesorów z dostępem do wspólnej pamięci, w której znajduje się struktura pozwalająca obliczać takie dopasowanie. Nie daje to jednak żadnego rozwiązania dla architektury wielu procesorów bez wspólnej pamięci, czyli takiej jaka jest w środowiskach klastrowych, a jaką oddaje interfejs MPI.

Zaproponowana w niniejszej pracy metoda pokazuje, że problem da się dobrze zrównoleglić, budując rozproszone drzewo trie z wszystkich spójnych podciągów genomu o określonej długości. Potwierdzeniem tego stanowią wykonane testy implementacji prostszej metody, w której zabrania się możliwości wystąpienia błędów w dopasowaniu na krótkim prefiksie. Słabością przedstawionej metody jest ilość potrzebnej pamięci, której przy tekście długości gigabajta znaków potrzeba bardzo dużo.

Podsumowując, problem niedokładnego dopasowania wielu krótkich sekwencji DNA do długiego genomu referencyjnego jest bardzo ciekawym problemem, możliwym do zrównoleglenia. W dalszych pracach nad efektywnym rozwiązaniem tego problemu, należy szukać innych metod umożliwiających zrównoleglenie, które będą wymagały mniej obliczeń celem dopasowania, czy też będą mniej kosztowne pamięciowo.

Literatura

[1] Z. Galil, K. Park, *An Improved Algorithm for Approximate String Matching*, 1989.

[2] G. H. Gonnet, R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, Addison-Wesley Publishers Ltd., 1991, Second Edition.

[3] H. Li, R. Durbin, *Fast and accurate short read alignment with Burrows-Wheeler Transform*, Bioinformatics Vol. 25 no. 14, 2009, s. 1754-1760.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowo-Techniczne, 2004, wydanie szóste zmienione i rozszerzone.

[5] M. Crochemore, W. Rytter, *Text Algorithms*, Oxford University Press, Inc., 1994.

[6] A. V. Aho, M. J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search*, Communications of the ACM Vol. 18 no. 6, 1975, s. 333-341.

[7] E. Ukkonen, *Finding Approximate Patterns in Strings*, J. Algorithms 6, 1985, s. 132-137.

[8] M. Balcerak, T. Strzebak, P. Russek, S. Koryciak, K. Wiatr, *Pattern Searching Scheme Using CPU & FPGA*, w Proceedings of Cracow Grid Workshop 2014, s. 59-60.

[9] *Opis klastra Zeus*, Akademickie Centrum Komputerowe CYFRONET AGH [dostęp 28.09.2015], `https://kdm.cyfronet.pl/portal/Zeus`.