PRACA MAGISTERSKA

# Współbieżny algorytm dla problemu dopasowania krótkich sekwencji DNA

Andrzej Dorobisz

Opiekun: prof. dr hab. inż. Kazimierz Wiatr

# Wprowadzenie

Dopasowanie krótkich sekwencji DNA:

- genom referencyjny, długość rzędu 1Gbp,
  (bp – para zasad, reprezentacja litera *A, C, G* lub *T*)
- zbiór odczytów, 50-200 milionów krótkich fragmentów DNA (*short-reads*)
- krótkie fragmenty, długość 32-100 bp

**Zadanie: dopasować krótkie fragmenty do genomu referencyjnego**
**Zasadnicza trudność** - dopasowania mogą być niedokładne.

Algorytmiczny punkt widzenia:

**problem niedokładnego dopasowania wielu krótkich wzorców**
**do bardzo dużego tekstu w języku nad alfabetem czteroliterowym**

# Struktura pracy

1. Przedstawienie problemu
2. Przegląd algorytmów dla problemu dopasowania z błędami
3. Propozycja własnej metody
4. Implementacja
5. Analiza wyników
6. Podsumowanie

# Przedstawienie problemu

**Odległość edycyjna -** miara podobieństwa dwóch wyrazów do siebie, najmniejsza ilość operacji edycji potrzebnych do zamiany jednego wyrazu na drugi.

**Operacje edycji:**

- zamiana litery
- wstawienie litery
- usunięcie litery

Przykład

słowo: ***abbabac***

| operacje edycji / pozycja | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| wyjściowe słowo | a | b | b | a | b | a | c | - |
| zamiana litery na pozycji 3 na literę c | a | b | **c** | a | b | a | c | - |
| wstawienie litery c na pozycji 3 | a | b | **c** | b | a | b | a | c |
| usunięcie litery na pozycji 3 | a | b | a | b | a | c | - | - |

Przykład:

słowa:

A = ***abbabac***

B = ***ababbad***

$$edit(A, B) = 3$$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **A** | = | a | b | **b** | a | b | **-** | a | **c** |
| | | ‖ | ‖ | | ‖ | ‖ | | ‖ | |
| **B** | = | a | b | **-** | a | b | **b** | a | **d** |

## Problem dopasowania z k błędami

- tekst *text*, wzorzec *pat*, liczba naturalna *k*
- znaleźć wszystkie takie pozycje *i*, że wzorzec *pat*, można dopasować do fragmentu tekstu od pozycji *i* za pomocą nie więcej niż *k* błędów (operacji edycji)

Formalnie:

znaleźć wszystkie i, takie że dla pewnego j zachodzi

$$edit(text_{i...j}, pat) \leq k$$

Dla k = 0 mamy problem dokładnego dopasowania czyli wyszukiwanie wzorca w tekście.

# Przykład

text = ACGACTTGACCGCTTA, pat = GAT, k = 1

| 1 | 2 | **3** | 4 | 5 | 6 | 7 | **8** | 9 | | 10 | 11 | **12** | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | C | G | A | C | T | T | G | A | - | C | C | G | C | T | T | A |
| | | ‖ | ‖ | | ‖ | | ‖ | ‖ | | | | ‖ | | ‖ | | |
| | | G | A | - | T | | ‖ | ‖ | | | | ‖ | | ‖ | | |
| | | | | | | | G | A | T | | | ‖ | | ‖ | | |
| | | | | | | | | | | | | G | A | T | | |

Wzorzec GAT można dopasować z jednym błędem na pozycjach: 3, 8, 12

# Przegląd algorytmów dla problemu dopasowania z błędami

Punkt wyjścia – algorytmy wyszukiwania wzorca w tekście (dokładne dopasowanie). Pozwalają wyrobić sobie ogląd na techniki stosowane przy wyszukiwaniu tekstu.

## Algorytmu wyszukiwania wzorca w tekście

| Algorytm | Charakterystyka |
|---|---|
| Metoda brute-force | próba bezpośredniego dopasowania wzorca na każdej pozycji |
| Algorytm Knutha-Morrisa-Pratta (KMP) | wyszukiwanie z użyciem tablicy prefikso-sufiksowej dla wzorca; czas liniowy |
| Algorytm Boyera-Moora | próba dopasowania wzorca na pozycjach od lewej do prawej z użyciem tablicy przejść; samo dopasowanie na danej pozycji zaczynamy od prawej do lewej; |
| Algorytm Rabina-Karpa | wyszukiwanie za pomocą porównywania hashów |
| Algorytm Aho-Corasick | algorytm do wyszukiwania wielu wzorców, konstrukcja automatu na drzewie trie wzorców, czas wyszukiwania liniowy |
| Metoda shift-or | dopasowanie z użyciem maski o tylu bitach co długość wzorca, aktualizacja maski przy każdym przejściu |
| Wyszukiwanie z transformatą Burrowsa-Wheelera (BWT) | konstrukcja tablicy sufiksowej a następnie BWT dla tekstu, wyszukiwanie na podstawie pewnych wzorów, obliczanie ich dla wzorca od prawej do lewej, czas samego dopasowania liniowy od długości wzorca |

# Algorytmy niedokładnego dopasowania wzorca w tekście

Stosowane są inne techniki, ciężko wykorzystać pomysły z dokładnego wyszukiwania wzorca.

Klasyczna metoda **- programowanie dynamiczne**

D[i][j] := minimalna odległość edycyjna pomiędzy pierwszymi ***i*** znakami wzorca a jakimś podsłowem tekstu, kończącym się na pozycji ***j***, czyli $D[i][j] = \min_{1 \le q \le j+1} \text{edit}(pat_{1...i}, text_{q...j})$

Zależność:

$$D[i][j] = \min(D[i-1][j] + 1, D[i][j-1] + 1, D[i-1][j-1] + (pati_i \neq text_j))$$

# Przykład

wzorzec **adbbc**, tekst **abbdadcbc,** dopuszczalna ilość błędów **k = 2**

| **D** | | 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 |
| | | | | | ^ | ^ | | | ^ | ^ | ^ |

Inicjalizacja:

$$D[0][j] = 0$$

$$D[i][0] = i$$

Dopasowanie wzorca: $D[m][j] \leq k$

Wzorzec można dopasować kończąc na pozycjach 3, 4, 7, 8, 9.

# Algorytmy niedokładnego dopasowania wzorca w tekście

| Algorytm | Charakterystyka |
|---|---|
| Wyszukiwanie za pomocą automatu | Bazując na klasycznej metodzie budujemy automat dla wzorca. Stanem jest kolumna z tabeli programowania dynamicznego (wektor *m* liczb), rozważamy przejścia wszystkimi możliwymi literami. Stan jest akceptujący gdy ostatnia wartość ≤ *k*.<br>Po zbudowaniu automatu, wyszukiwanie to chodzenie po automacie, zgodnie z literami tekstu. Każde dojście do stanu akceptującego – znalezienie jednego niedokładnego dopasowania |
| Liczenie wzdłuż przekątnych | Usprawnienie klasycznej metody. Nie obliczamy całej tabeli programowania dynamicznego. W tym celu dla każdej przekątnej pamiętamy tylko specjalne komórki *d*-node,<br>(dla d = 0, 1, ..., k) – najdalsze pole o wartości *d* w obrębie danej przekątnej. Wartości te obliczamy wzdłuż przekątnych, nie przechowujemy całej tablicy. Algorytm korzysta ze specjalnego wspólnego drzewa prefiksowego dla wzorca i tekstu. |
| Backtracking z użyciem BWT | Użycie backtrackingu – rekurencyjne wywołania sprawdzające wszystkie możliwe operacje edycji, dla metody wyszukiwania wzorca z użyciem transformaty Burrowsa-Wheelera |

# Propozycja własnej metody

## Sub-Sequences TRIE

Budujemy explicite drzewo TRIE dla wszystkich sufiksów, ale ograniczmy ich długość do maksymalnej długości krótkiego odczytu.

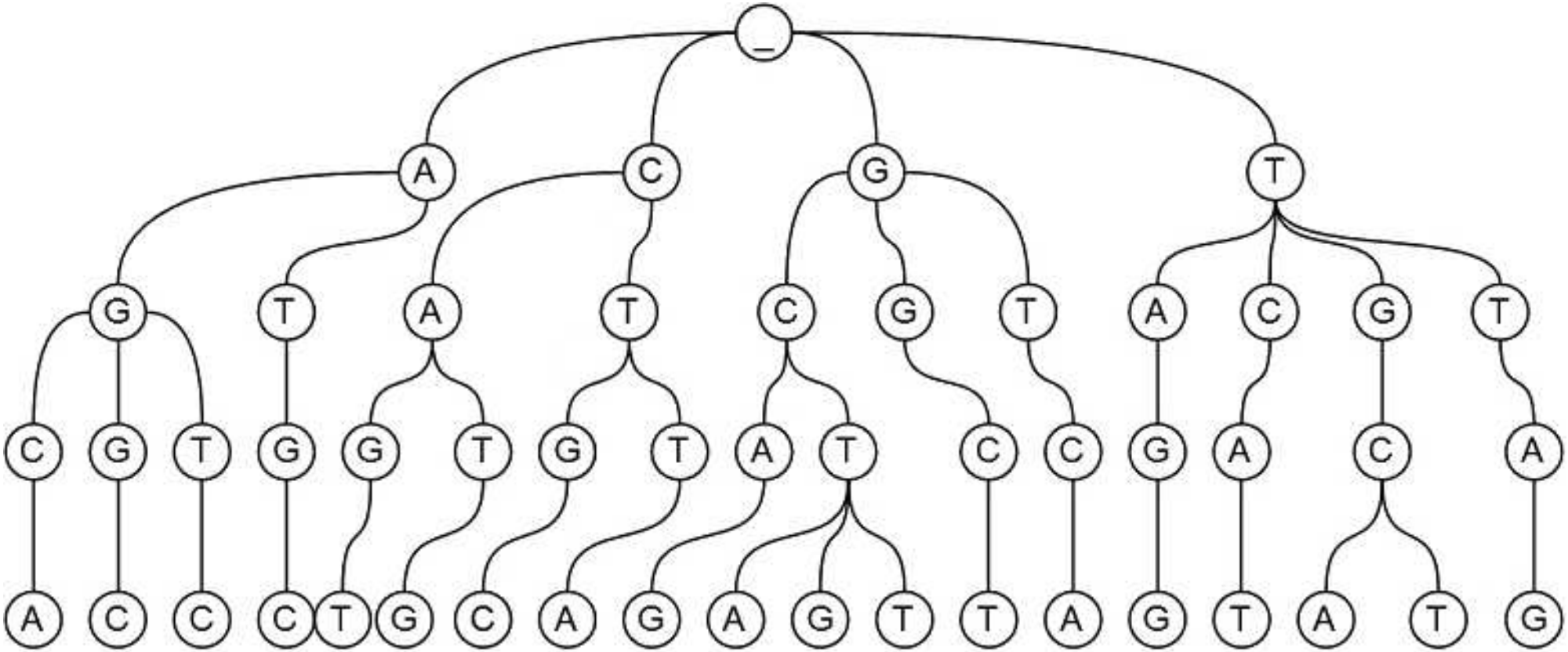
Genom:

AGCATGCTGCAGTCATGCTTAGGCTA

Przyjmując długość 4, odczytujemy wszystkie podsłowa długości 4:

AGCA, GCAT, CATG, ATGC, TGCT,
GCTG, CTGC, TGCA, GCAG, CAGT,
AGTC, GTCA, TCAT, CATG, ATGC,
TGCT, GCTT, CTTA, TTAG, TAGG,
AGGC, GGCT, GCTA

Budujemy drzewo TRIE dla tych podciągów – nazywamy je **ss_trie**




Wyszukiwanie – backtracking

chodzenie po drzewie, rekurencyjne wywołania, zmniejszanie możliwej liczby błędów przy niedokładnym dopasowaniu

# Algorytm realizujący dopasowanie

**Algorytm 3.1** Niedokładne dopasowanie z użyciem *ss_trie*

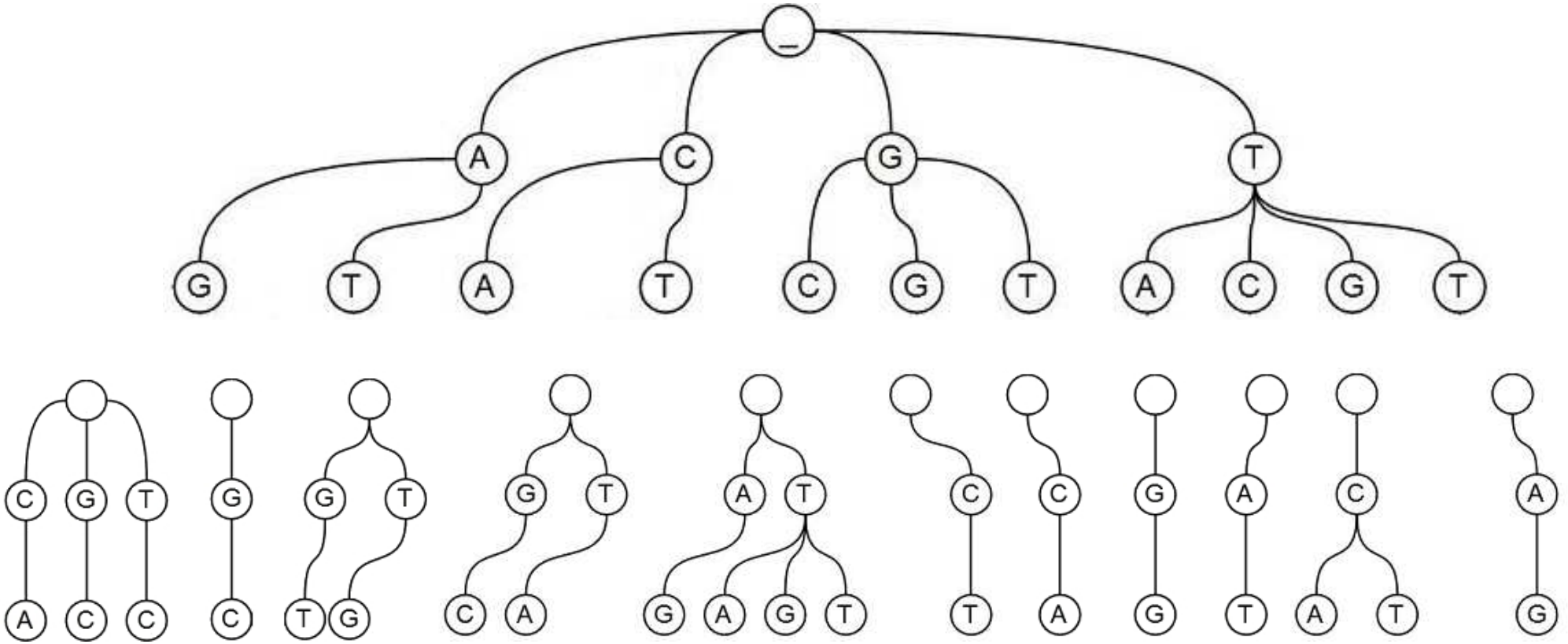
```
procedure MismatchRec(p, pos, cur, k)
    if cur is NULL then return ∅
    end if
    if pos == |p| + 1 then return SS(cur)
    end if
    R ← MismatchRec(p, pos + 1, next(cur, p_pos), k)
    if k > 0 then
        for all x ∈ {A,C, G, T} , x ≠ p_pos do
            R ← R ∪ MismatchRec(p, pos + 1, next(cur, x), k − 1)
        end for
        for all x ∈ {A,C, G, T} do
            R ← R ∪ MismatchRec(p, pos, next(cur, x), k − 1)
        end for
        R ← R ∪ MismatchRec(p, pos + 1, cur, k − 1)
    end if
    return R
end procedure
```

**p -** wzorzec
**pos -** aktualnie dopasowywana pozycja
**cur –** aktualny wierzchołek drzewa ss_trie
**k -** dopuszczalna ilość błędów jakie jeszcze możemy wykonać

**NULL –** brak wierzchołka
**SS(cur) –** pozycje podciągów genomu przechodzące przez wierzchołek *cur*
**next(cur, x) –** funkcja przejścia w drzewie ss trie

# Sposób zrównoleglenia

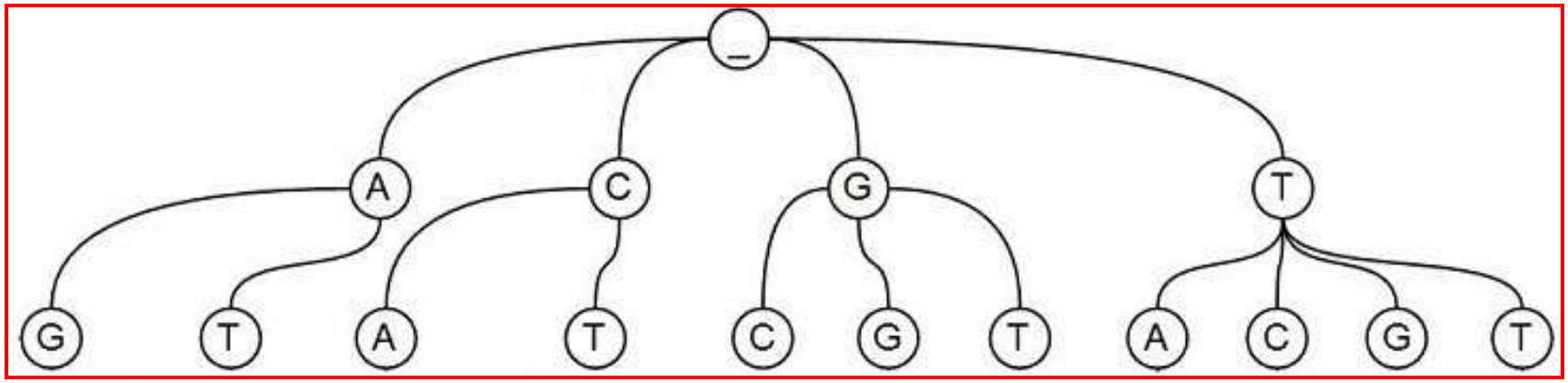
parametr *root_depth* – pierwsze *root_depth* poziomów zostawiamy na pierwszej maszynie, pozostałe poddrzewa rozsyłamy do innych maszyn.



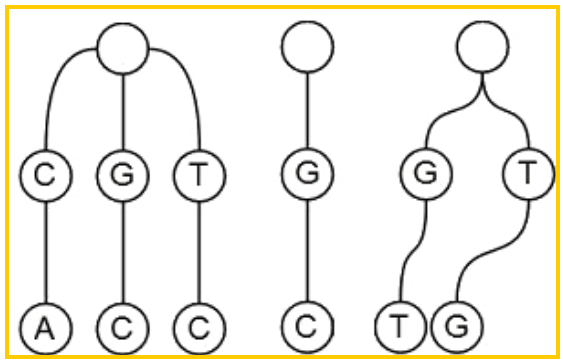

Drzewo po podziale na pierwsze *root_depth* = 2 poziomów. Powstaje 11 poddrzew.

Podział drzewa pomiędzy 5 maszyn:

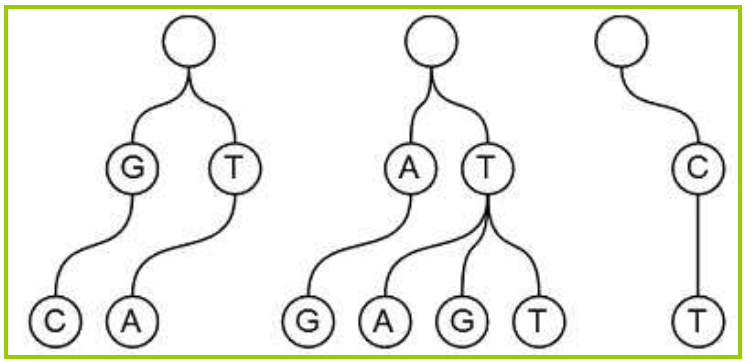
**root**



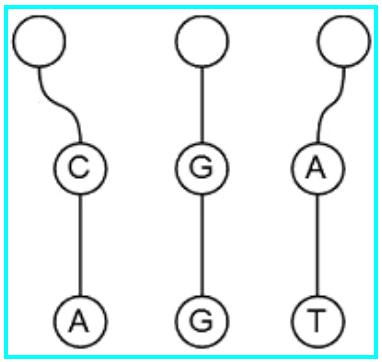

**worker 1**



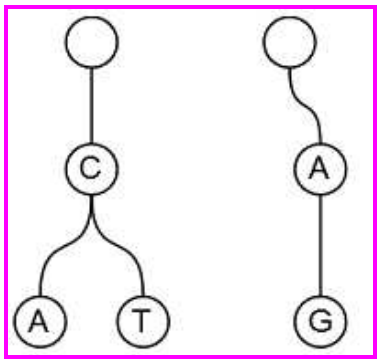

**worker 2**




**worker 3**




**worker 4**



## Wyszukiwanie

- początek na maszynie root
- dotarcie do liścia, zapamiętanie parametrów funkcji
- rozesłanie zleceń dokończenia wyszukiwania do maszyn worker
- worker, dla każdego zlecenia kontynuacja z przesłanymi parametrami
- odesłanie wyników z wszystkich zleceń

## Uproszczona metoda

- zamiast *root_depth* parametr *exact_prefix*
- na pierwszych *exact_prefix* znakach musi być dokładne dopasowanie
- na maszynie root pierwsze *exact_prefix* poziomów

Wyszukiwanie:

- root, dokładne dopasowanie
- kontynuacja dopasowania, tylko jeden worker

# Implementacja

- język C++11
- MPI *(Message Passing Interface)*
- prostsza metoda

## Architektura

- wiele maszyn
- brak współdzielenia pamięci
- pierwsza maszyna (root) zarządza przetwarzaniem
- pozostałe (worker) nie różnią się od siebie i wykonują zlecone obliczenia
- tylko root czyta pliki wejściowe z dysku, dane do workerów są rozsyłane
- po skończonym dopasowaniu worker odsyła wyniki do roota
- root zapisuje wyniki w pliku wynikowym na dysku

# Pliki wejściowe

### Genom referencyjny

```
// length = 100000
// seed = 7
100000
CTTTCTGCACTCTGCAGGGAATAACACACGCGGAG
TTACACACAGTACCGCCGCCTCGATACCGGTGCTT
CCTTCCGGGTATAGACGGTGCTTGGCGATCTAGGT
ATCGGAGCACGCTAACCAATGGCGCTCCCTCCGGT
ATGGTTACGACTCCTATCGCGCGAGCCAATTAGCG
GAGTGGACTGCACGGTCTCCCTGCTCCTAGGGTAC
CTCTGTAGCGAATTCACAGGTATTAGCGCCTGAAC
TACCCAGCTCGAGTGCGGACGGGATATGAATTACC
CCTGAAAATTTAGCATATGAGGAAGAAGACTCGGT
GGTTCTGCCTGATCGATAAATAACACACTGACATA
GGGTTGCACGGCGTCGGCGTCTTGGCGTCGATAGT
AAAAGTG...
```

### Krótkie odczyty

```
// short-read number: 2000
// seed: 8
// range: [32, 100]
// reference genome:
test_out/20_cpus/data/ref2_100k
// reference length: 100000
2000
TGTCTATTGTCTCAATGCGCACTTCCTGATAAG    // pos: 11071
CACCCGGGGACACTATGGTAAAATCCAAGGATGCATTTCG…
GATAGCTGTCTGATTAATGATTGCAAATGAGCACCGGACC…
GGGTGCGAAGGGATACAGGGAACTTAATTATCGCCC // pos: 96793
TCGATTTGAGCTCTCTCCGTTCGCGTTCATCTATATTTCTCG…
TGCTGACTTGACTTTGACACTTTCCTCGGCTCTAAGAGCAA…
CATTATGACAAAGGTATGAACGAGACTTACGCCCAGAGGC…
AATGATAGATGAGACAGCAGCAATCTAGCTACTTTTGTCTG…
TAGCTGAACCCGATTTCATGAGGCCTGAAGTTGCACAACCC…
...
```

# Plik wynikowy

```
0     -> {11071}
1     -> {40906}
2     -> {86357}
3     -> {96793}
4     -> {55768}
5     -> {29696}
6     -> {87203}
7     -> {49666}
8     -> {53295}
9     -> {35348}
10    -> {92176}
11    -> {19574}
12    -> {30882}
13    -> {52190}
14    -> {49991}
15    -> {13683}
16    -> {75816, 75821}
17    -> {36653, 36655, 36657}
18    -> {67161}
19    -> {87577}
20    -> {41669}
...
```

# Analiza wyników

Przetestowano

- implementację rozproszonego drzewa ss_trie
- implementację drzewa ss_trie dla jednego procesora

Testy wykonano na klastrze obliczeniowym ZEUS.
Standardowy węzeł: procesory Intel, 2.4 GHz, sumarycznie 12 rdzeni.

Ogólne parametry testów

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

# Testy

- czas obsługi genomu
  - mierzony czas konstrukcji drzewa w zależności od długości genomu i liczby procesorów
  - mierzony czas wczytywania i rozsyłania genomu

- czas dopasowywania krótkich odczytów
  - mierzony czas niedokładnego dopasowania w zależności od dopuszczalnej ilości błędów oraz liczby procesorów
  - mierzony czas przesyłania krótkich odczytów i gromadzenia wyników

Testy wykonane zarówno dla kilku jak i kilkunastu procesorów.

# Testy – przykładowe wyniki

| długość genomu | 50 000 | 100 000 | 1 000 000 |
|---|---|---|---|
| ss-trie | - | - | - |
| W=4 | 0.26 | 0.24 | 2.17 |
| W=20 | 0.35 | 0.45 | 2.55 |

Czas rozsyłania danych [s]

| długość genomu | 50 000 | 100 000 | 1 000 000 |
|---|---|---|---|
| ss-trie | 3.07 | 6.09 | 60.88 |
| W=4 | 1.06 | 2.17 | 26.63 |
| W=20 | 0.29 | 0.51 | 5.26 |

Czas konstrukcji drzewa [s]

# Testy – przykładowe wyniki

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

| ilość błędów | 3 | 4 | 5 |
|---|---|---|---|
| ss-trie | 3.35 | 34.46 | - |
| W=4 | 0.90 | 9.75 | - |
| W=20 | 1.09 | 2.71 | 14.11 |

Czas całości procesu dopasowania [s]

# Analiza wyników - wnioski

- wraz ze wzrostem długości genomu, liniowo rośnie czas potrzebny na zbudowanie drzewa ss_trie
- konstrukcja drzewa dobrze się zrównolegla
- koszt komunikacji pomiędzy maszynami jest opłacalny w porównaniu do przyspieszenia jakie daje
- kluczowy dla czasu działania programu jest parametr dopuszczalnej liczby błędów
- przy porównywaniu krótkich odczytów problem dobrze się zrównolegla; wraz ze wzrostem liczby przetwarzających procesorów, czas dopasowania proporcjonalnie spada

# Podsumowanie

- problem niedokładnego dopasowania wielu krótkich sekwencji DNA jest ciekawym problemem, możliwym do zrównoleglenia
- zaproponowana metoda dostarcza efektywnego sposobu zrównoleglenia, potwierdzeniem czego są testy implementacji prostszej metody
- słabością metody jest ilość potrzebnej pamięci do przechowywania struktury potrzebnej do wyszukiwania
- w dalszych pracach należy szukać dalszych sposobów zrównoleglenia, które będą wymagały mniej obliczeń lub będą mniej kosztowne pamięciowo