Penerapan Algoritma Aho-Corasick Sebagai Optimasi Algoritma Knuth-Morris-Pratt dalam Uji DNA Penyakit Turunan Umum

Nathanael Santoso - 13520129
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13520129@std.stei.itb.ac.id
and nart4hire@gmail.com

***Abstract*—Penyamaan suatu subset DNA sangat penting untuk mendiagnosis penyakit turunan umum agar bisa diobati lebih awal untuk mencegah kasus terburuk penyakit. Tetapi, panjang gen setiap manusia bisa mencapai 6 Milyar pasangan basa dengan pola penyakit turunan yang mencapai ratusan ribu pasangan basa. Untuk itu, jika diperiksa satu per satu menggunakan algoritma Knuth-Morris-Pratt (KMP) akan sangat lama untuk memeriksa setiap pola. Algoritma Aho-Corasick adalah perpanjangan dari algoritma KMP yang mengintegrasikan struktur data pohon sehingga pencocokan pola bisa dilakukan dalam waktu linear untuk pola sebanyak apapun.**

Keywords—Knuth-Morris-Pratt, Aho-Corasick, Pattern Matching, DNA

I. PENDAHULUAN

Uji DNA adalah teknik biologis yang dilakukan untuk menunjukkan keberadaan ataupun ekspresi suatu gen dan perubahannya dalam makhluk hidup. Dalam konteks medis, uji DNA biasa dilakukan untuk memeriksa apakah ada penyakit yang berhubungan dengan gen atau keberadaan pasien sebagai karier gen yang menyebabkan penyakit. Uji DNA sangat berguna untuk meningkatkan kualitas hidup manusia karena dapat menemukan penyakit yang tersembunyi sebelum menjadi parah sehingga dapat dilakukan aksi pencegahan.

Uji DNA dipakai dalam berbagai kasus seperti skrining bayi yang baru lahir, uji karier, diagnosis prenatal, diagnotik/prognostic, ataupun sebagai uji prediktif. Pada uji DNA, sel dari pasien akan diambil dan diekstrak informasi genetiknya untuk diuji di laboratorium[1]. Ada berbagai macam pengujian DNA, namun pada laporan ini hanya akan difokuskan pada uji molecular DNA.

DNA terdiri dari 4 basa, Adenin (A), Sitosin (C), Guanin (G), dan Timin (T). Keempat basa tersebut membentuk rantai protein basa yang terdapat dalam DNA. Secara teori, jika rantai DNA seseorang dapat diekstrak dan dimasukkan sebagai data dalam komputer, maka DNA tersebut dapat dianalisis untuk menemukan gen yang menyebabkan penyakit. Jika diketahui terdapat pola gen penyakit ‘w’ atau pola yang mirip dengan ‘w’ dalam sebuah sekuens gen ‘s’, maka dapat disebutkan bahwa pasien kemungkinan mengidap penyakit turunan akibat gen tersebut.

Meskipun begitu, sangat sulit untuk memetakan seluruh gen manusia sebagai data yang dapat disimpan komputer. Ensembl Project dari European Bioinformatics Institute mengatakan bahwa genome *homo sapiens* adalah tepat sepanjang 3,096,649,726 pasangan basa[2]. Menemukan pola genom spesifik dalam data sebanyak itu tidak mudah dan memungkinkan memakan waktu yang sangat lama. Algoritma string matching Knuth-Morris-Pratt (KMP) dapat mencari pola yang serupa dalam waktu yang relative linear, namun hanya bisa dilakukan satu per satu. Sedangkan, terdapat banyak sekali penyakit turunan sehingga jika dilakukan pencocokan satu per satu akan mengakitbatkan waktu tunggu yang cukup signifikan. Oleh karena itu, algoritma Aho-Corasick diajukan sebagai alternatif solusi karena Aho-Corasick cocok digunakan untuk pattern matching DNA dan memakan waktu yang relatif lebih cepat daripada algoritma KMP.

II. DASAR TEORI

A. *Pencocokan String*

Pencocokan String atau String/Pattern Matching adalah kategori algoritma yang berguna untuk mencocokkan pola yang terdiri dari berbagai karakter dengan subset dari teks yang lebih besar dari pola tersebut. Biasanya string terdiri dari beberapa karakter umum dan karakter khusus, namun untuk bahasa lain jumlah dan varian karakter bertambah dalam jumlah dan kompleksitas.

Algoritma string matching juga terdapat berbagai jenis mulai dari yang naif sampai yang kompleks. Secara umum, algoritma string matching dikategorikan dari cara pemrosesan data awal. Algoritma string matching dapat memproses pola yang dicocokkan terlebih dahulu atau tidak dan juga pemrosesan badan teks yang digunakan. Kategori algoritma string matching dapat dilihat pada diagram berikut.

		Text preprocessing	
		NO	YES
Pattern preprocessing	NO	Elementary algorithms	Factor automata, index methods
	YES	Pattern matching automata	Pattern matching automata, factor automata, signature methods

Fig. 1. Categories of String Matching Algorithm[3]

Dalam lingkup bioinformatika, suatu string DNA terbentuk dari hanya 4 huruf yaitu ‘A’, ‘C’, ‘G’, dan ‘T’ yang masing-masing merepresentasikan protein basa yang terdapat dalam DNA. Karena variasi huruf kecil, maka pencocokan string akan cepat jika dilakukan oleh algoritma KMP atau turunannya.

B. Algoritma Knuth-Morris-Pratt

Algoritma Knuth-Morris-Pratt adalah algoritma yang mengembangkan pendekatan naif pada string matching dengan cara mengkomputasi terlebih dahulu sebuah fungi kegagalan, fail. Fungsi fail ini didefinisikan sebagai posisi dalam string yang merupakan prefix terpanjang dari string yang juga merupakan proper suffix dari string saat posisi kegagalan terjadi.

Misalkan terdapat sekuens dna ‘ATAATG’. Algoritma KMP akan mencocokkan setiap karakter dari badan teks dengan sekuens tersebut. Jika terdapat gagal pencocokan pada indeks ke-4 dari string tersebut, maka algoritma akan kembali mencocokkan pada indeks ke-2. Berikut adalah visualisasi dari fungsi fail KMP.

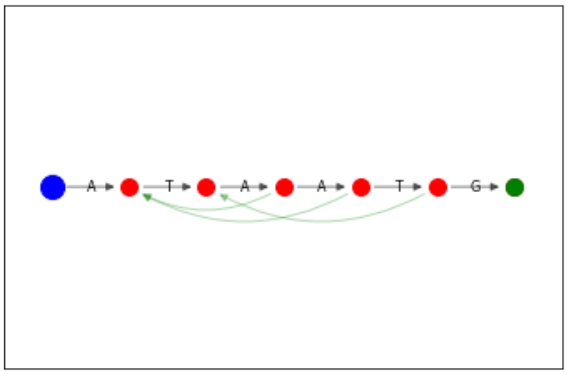

Fig. 2. Failure Function of String ‘ATAATG’ (Private Works)

Pada diagram, panah hijau menandakan fungsi fail dari KMP. Jika dilihat dengan seksama, maka diagram akan terlihat seperti Deterministic Finite Automata (DFA). Hal ini karena failure function meniru perpindahan state DFA dan mengimplementasikannya ke dalam algoritma string matching.

Algoritma KMP secara intuitif diketahui bahwa akan lebih cepat jika sekuens yang dicocokkan memiliki variasi karakter yang kecil. Hal ini karena variasi karakter yang kecil memaksimalkan kemungkinan sebuah proper suffix dari sebuah string merupakan prefixnya sehingga cocok digunakan untuk pencocokkan DNA karena jumlah karakter unik yang terdapat dalam DNA hanya 4.

C. Algoritma Aho-Corasick

Algoritma Aho-Corasick merupakan perpanjangan dari algoritma Knuth-Morris-Pratt untuk jumlah input yang lebih besar. Pada algoritma Aho-Corasick, input akan terlebih dahulu dijadikan node pada pohon DFA seperti pada KMP, namun pohon tersebut mengandung banyak state penerimaan atau ‘out’ node dan failure function didefinisikan sebagai prefix terbesar dari string apapun yang juga merupakan proper suffix dari tempat kegagalan pencocokan tersebut terjadi.

Misalkan diberikan sekumpulan string ‘ATAATG’, ‘ATCGTG’, ‘CGTAG’, dan ‘ATA’ maka komputasi dari out dan failure dari sekumpulan string tersebut akan membuat diagram pencarian seperti berikut.

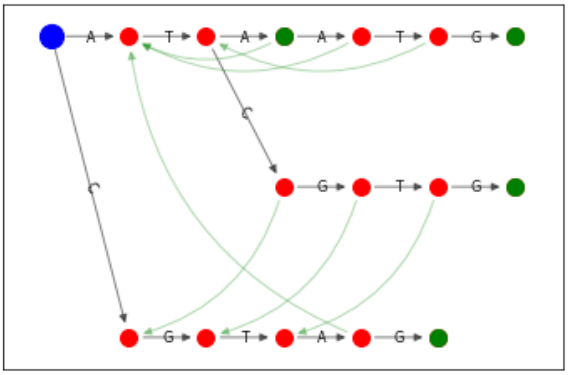

Fig. 3. Failure Function of Set of Strings With Out Node (Private Works)

Pada Diagram, Node yang berwarna hijau menandakan node keluaran. Jika node tersebut dikunjungi, maka akan ditambahkan indeks awal kata beserta kata output dari node tersebut pada himpunan penyelesaian. Fungsi fail dari diagram tersebut ditandai oleh panah hijau. Berbeda dengan KMP, fungsi fail dari Aho-Corasick dapat pergi lintas cabang. Misalkan pada pola ‘ATCGTG’, jika terjadi kegagalan pada indeks ke-4 dari pola, maka pohon akan melompat ke indeks ke-2 dari pola ‘CGTAG’. Implementasi seperti ini menjamin bahwa waktu pencarian tidak akan dependen dengan jumlah pola yang dicocokkan dan akan tetap linear meskipun jumlah pola yang dicocokkan banyak.

III. IMPLEMENTASI

Implementasi dari algoritma Aho-Corasick menggunakan bahasa Python versi 3.10. Algoritma Aho-Corasick diimplentasikan menjadi sebuah Pustaka dengan kelas AhoCorasick yang menjadi interface dari algoritma ini.

A. Search Tree, Fail, and Output Generation

Pada bagian ini terdapat detil implementasi terkait pohon pencarian, fungsi fail, dan juga node out.

Pohon pencarian diimplementasikan sebagai Goto[i][j] dengan i adalah nomor identifikasi node dan j merupakan hash dari karakter protein basa dengan ‘A’ adalah 0, ‘C’ adalah 1, ‘G’ adalah 2, dan ‘T’ adalah 3. Misalkan algoritma sedang berada pada node dasar dan masukan adalah ‘A’, maka dipanggil Goto[0][0].

Fungsi Fail diimplementasikan sebagai array Fail[i] yang berisi nomor identifikasi node jika fail pada node dengan nomor identifikasi i. Misalkan terdapat gagal pencocokan pada node dengan nomor identifikasi 1, maka node yang dicocokkan berikutnya adalah Fail[1].

Node out diimplentasikan sebagai dictionary out[i] yang mempunyai kunci nomor indentifikasi node dan memetakan kepada himpunan solusi untuk node tersebut. Misalkan pada node 5, terdapat kecocokan terdahap string 'ACACA', maka out[5] = ('ACACA').

```
from __future__ import annotations
from random import randint
import matplotlib.pyplot as plt
from networkx import DiGraph, draw_networkx_nodes, draw_networkx_edges, draw_networkx_edge_labels

class AhoCorasick:
    # Character set for DNA
    charset = {
        'A': 0,
        'C': 1,
        'G': 2,
        'T': 3
    }

    # Number of allowed characters
    charset_len = len(charset)

    def __init__(self, words: list[str]) -> None:
        # Neutralize Case in words and remove characters other than ACGT
        self.words = [word.upper() for word in words if not any(char not in self.charset.keys() for char in word.upper())]

        # Get Upper Bound Number of Nodes
        self.max_nodes = sum([len(word) for word in words]) + 1

        # Output Dictionary
        self.out = {
            i: set() for i in range(self.max_nodes) # Set of Output Words
        }

        # Failure Array
        self.fail = [-1] * self.max_nodes

        # Goto Matrix, rows = max_nodes, cols = charset_len
        self.goto = [[-1] * self.charset_len for _ in range(self.max_nodes)]

        # Build Output, Failure, and Goto
        self.nodes = self.__initialize()

    def __initialize(self) -> int:
        # Initial number of nodes (Includes Root Node)
        nodes = 1

        # Assign States
        for word in self.words:
            current_node = 0

            for char in word:
                hash = self.charset[char]

                if self.goto[current_node][hash] == -1:
                    self.goto[current_node][hash] = nodes
                    nodes += 1

                current_node = self.goto[current_node][hash]

            self.out[current_node].add(word)

        # Preparing To Compute Fail Function
        node_q = []

        for hash in range(self.charset_len):
            node = self.goto[0][hash]
            if node != -1:
                self.fail[node] = 0
                node_q.append(node)
            else:
                self.goto[0][hash] = 0

        # Compute Fail Function
        while node_q:
            parent_node = node_q.pop(0)

            for hash in range(self.charset_len):
                current_node = self.goto[parent_node][hash]
                if current_node != -1:
                    fallback_node = self.fail[parent_node]

                    while self.goto[fallback_node][hash] == -1:
                        fallback_node = self.fail[fallback_node]
                    fallback_node = self.goto[fallback_node][hash]

                    self.fail[current_node] = fallback_node

                    self.out[current_node].union(self.out[fallback_node])
                    node_q.append(current_node)

        return nodes
```

B. Search Function

Pada bagian ini terdapat implementasi dari pencarian menggunakan Kelas AhoCorasick.

Implementasi dari penjalaran pohon pencarian terdapat pada fungsi private get_next_node yang menerima id node referensi dan input karakter, kemudian mengeluarkan id node berikutnya. Jika terdapat node yang bisa dituju dari node referensi menggunakan karakter input, maka fungsi akan mengeluarkan node tersebut, namun jika tidak ada, maka akan pergi ke node yang dituju oleh fungsi fail sampai didapatkan node yang valid.

```
class AhoCorasick:
    ...
    # Node Traversal Using Goto and Fail Functions
    def __get_next_node(self, ref_node: int, input: str) -> int:
        current_node = ref_node
        hash = self.charset[input]

        next_node = self.goto[current_node][hash]
        while next_node == -1:
```

```
            current_node = self.fail[current_node]
            next_node = self.goto[current_node][hash]

        return next_node

    # Search for initialized strings in text body,
returns dictionary of words and indices
    def search_in_sequence(self, body: str) -> dict:
        sequence = body.upper()

        result: dict[list[tuple]] = dict()

        current_node = 0
        for i in range(len(sequence)):
            current_node                                =
self.__get_next_node(current_node, sequence[i])

            if self.out[current_node]:
                for word in self.out[current_node]:
                    if result.get(word):
                        result[word].append(i          -
len(word) + 1)
                    else:
                        result[word] = [i - len(word) +
1]

        return result
```

C. Auxiliary Functions

Untuk membantu visualisasi data dan pembangkitan data acak maka dibuat beberapa fungsi pembantu.

Fungsi Visualize() menggunakan Pustaka Networkx dan MatPlotLib untuk membuat diagram pohon pencarian.

Fungsi getRandomGenomes dan getRandomBody membangkitkan pola dan badan teks acak menggunakan pusataka Random.

```
class AhoCorasick:
    ...
    # Private Get Position Function
    def __getpos(self, current_node, depth, pos):
        count = 0
        for hash in range(self.charset_len):
            next_node = self.goto[current_node][hash]
            if next_node != -1 and next_node != 0:
                if count > 0:
                    depth -= 10
                pos[next_node]  =  (pos[current_node][0]
+ 10, depth)
                depth,  pos  =  self.__getpos(next_node,
depth, pos)
                count += 1
        return depth, pos

    # Visualization using Networkx Digraph
    def visualize(self) -> None:
        dg = DiGraph()

        pos = {0: (0, 0)}
        _, pos = self.__getpos(0, 0, pos)

        edge_labels = dict()
        dg.add_node(0)
        for i, row in enumerate(self.goto):
            for j, col in enumerate(row):
                if col != 0 and col != -1:
                    dg.add_node(col)
                    dg.add_edge(i, col)
                    edge_labels[(i, col)] = "ACGT"[j]
        draw_networkx_nodes(dg,   pos,   node_size=100,
node_color="red")
        draw_networkx_edge_labels(dg,              pos,
edge_labels=edge_labels, font_size=8)
        draw_networkx_edges(dg, pos, alpha=0.5)

        node_list = list()
        for key, value in self.out.items():
            if value:
                node_list.append(key)
        draw_networkx_nodes(dg,                    pos,
nodelist=node_list, node_size=100, node_color="green")
        draw_networkx_nodes(dg,   pos,   nodelist=[0],
node_size=200, node_color="blue")

        edge_list = list()
        for i in range(self.max_nodes):
            if self.fail[i] != -1 and self.fail[i] !=
0:
                dg.add_edge(i, self.fail[i])
                edge_list.append((i, self.fail[i]))
        draw_networkx_edges(dg,                    pos,
edgelist=edge_list,                edge_color="green",
connectionstyle="arc3, rad=-0.3", alpha=0.3)

        plt.show()


def getRandomGenomes(length: int = 8, frequency: int =
10) -> list[str]:
    words = list()
    for _ in range(frequency):
        words.append("".join("ACGT"[randint(0,  3)]  for
_ in range(length)))
    return words

def getRandomBody(length: int = 10000) -> str:
    return  "".join("ACGT"[randint(0,  3)]  for  _  in
range(length))
```

D. Usage Examples

Misalkan suatu laboratorium genetika ingin memberikan diagnosis umum untuk sekumpulan penyakit turunan umum yang terdapat pada bayi yang baru lahir sebagai bentuk prosedur skrining. Maka laboratorium dapat membuat tabel seperti tabel pada diagram di bawah. (Tabel di bawah hanya contoh, dan Sequence yang dipakai adalah sequence acak)

Disease Name	Sequence
Cystic Fibrosis	CTCCCACTTA
Alpha-Thalassemia	TCTTAAGCTC
Beta-Thalassemia	ATCCGTGCAG
Sickle Cell Anemia	GCCCGGTAGG
Marfan Syndrome	AACAGCGCAG
Fragile X Syndrome	CGCAGTAATC
Huntington's Disease	TAAGTGTGAT
Hematochromatis	TAACCTCGTT

Fig. 4. Disease Name and DNA Sequence (Random) Table (Private Works)

Dari data yang diperoleh pada table tersebut, pustaka AhoCorasick akan membuat Pohon Pencarian, Fungsi Fail, dan Fungsi Output dari data tersebut. Visualisasi dari pohon pencarian tersebut terdapat pada diagram di bawah.

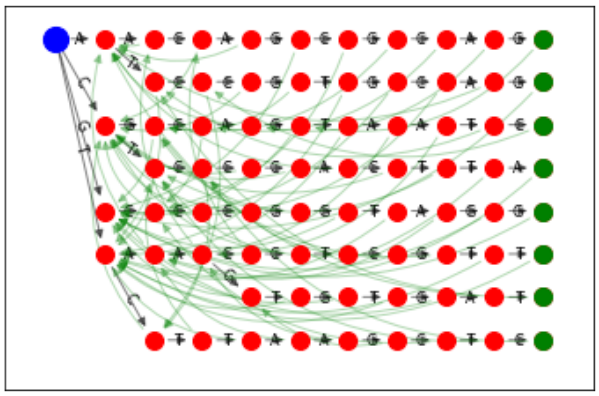

Fig. 5. Mock Search Tree Diagram (Private Works)

Misalkan dimasukkan patient genome (Random) sepanjang satu juta karakter pada fungsi search_in_sequence, maka mungkin dapat dihasilkan data sebagai berikut.

Sequence	Disease Name	Index Found
GCCCGGTAGG	Sickle Cell Anemia	2478, 50638, 622554
CTCCCACTTA	Cystic Fibrosis	234556
TAAGTGTGAT	Huntington's Disease	288560, 452253
TAACCTCGTT	Hematochromatis	341045, 715220
ATCCGTGCAG	Beta-Thalassemia	430630, 750690, 786647
CGCAGTAATC	Fragile X Syndrome	729173, 807540
AACAGCGCAG	Marfan Syndrome	820763

Fig. 6. Mock Search Data (Private Works)

Dari Fig. 6. Dapat diketahui bahwa pasien berpotensi mengidap penyakit turunan Sickle Cell Anemia, Cystic Fibrosis, Huntington's Disease, Hematochromatis, Beta-Thalassemia, Fragile X Syndrome, dan Marfan Syndrome. Penyakit juga disertakan indeks dalam badan teks di mana genome penyakit ditemukan.

Aplikasi dari Pustaka ini pada dunia nyata tentu saja akan membuahkan hasil yang lebih realistis, namun dapat dibuktikan dari use case dengan mock data bahwa Pustaka bekerja dengan semestinya dan berfungsi. Salah satu kelemahan dari Pustaka yang dibuat adalah kurangnya fungsi untuk mengetahui most similar string yang sering juga dipakai dalam dunia nyata.

Oleh karena itu, dapat disimpulkan bahwa Aho-Corasick merupakan salah satu metode yang valid dalam penyelesaian masalah general diagnosis penyakit turunan.

IV. Pengujian

Pengujian Pustaka Python hasil implementasi diujicobakan pada kakas Jupyter Notebook menggunakan Pustaka pengukuran waktu dan visualisasi data. Pengujian dilakukan dengan cara progressive overloading yaitu setiap algoritma yang diuji akan diberikan beban yang semakin berat setiap iterasi dan diukur perbedaan waktu per iterasi tersebut. Sampel uji yang dipakai akan dibangkitkan secara acak menggunakan Pustaka Random. Hasil waktu dari setiap algoritma kemudian divisualisasikan dan dibandingkan.

A. Stress Testing

1) Word Frequency

Pada Pengujian Word Frequency, Setiap iterasi akan menggandakan jumlah kata. Berikut adalah diagram area yang dihasilkan uji coba.

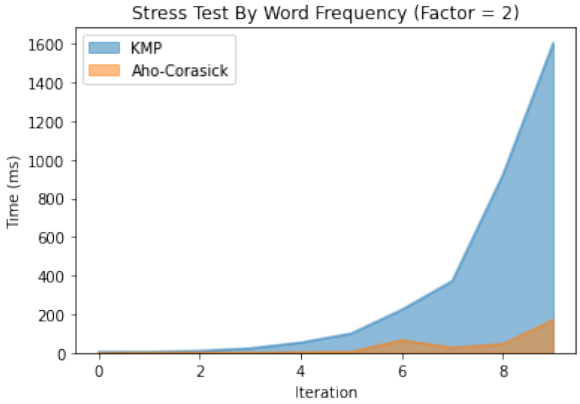

Fig. 7. Time Taken By KMP and Aho-Corasick based on Word Frequency (Private Works)

Pada Diagram dapat terlihat hasil dari uji yang menggandakan jumlah kata. Waktu yang diperlukan Algoritma KMP meningkat berbanding lurus dengan jumlah kata baru, namun Aho-Corasick terlihat relatif linear. Dapat disimpulkan bahwa metode Aho-Corasick Lebih baik untuk jumlah kata yang lebih banyak dibangingkan KMP.

2) Body Text Length

Pada Pengujian Body Text Length, setiap iterasi akan menggandakan panjang dari teks masukan. Berikut adalah diagram area yang dihasilkan uji coba.

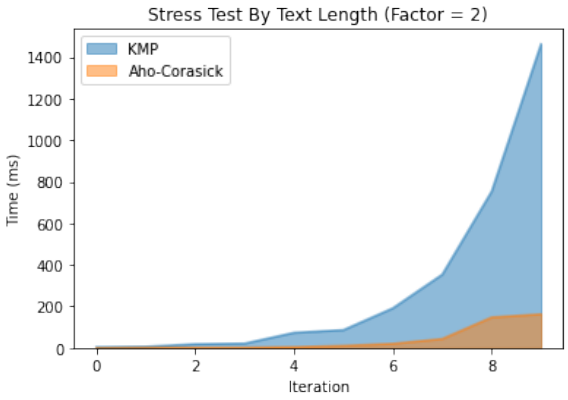

Fig. 8. Time Taken by KMP and Aho-Corasick based on Body Text Length (Private Works)

Pada Diagram dapat terlihat hasil dari uji yang menggandakan panjang badan teks. Waktu yang diperlukan Algoritma KMP meningkat berbanding lurus dengan panjang teks, namun Aho-Corasick terlihat tidak terlalu terpengaruh. Dapat disimpulkan bahwa metode Aho-Corasick Lebih baik untuk badan teks yang lebih panjang dibangingkan KMP.

3) Sequence Length and Body Text Length

Pada Pengujian Sequence Length dan Body Text Length, setiap iterasi akan menggandakan panjang dari teks masukan dan juga panjang pola yang dicari. Berikut adalah diagram area yang dihasilkan uji coba.

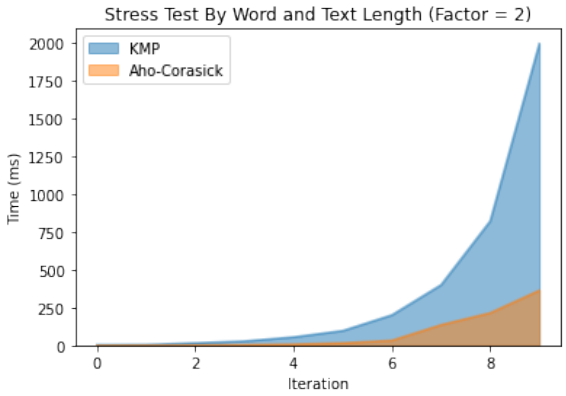

Fig. 9. Time Taken by KMP and Aho-Corasick based on Sequence Length and Body Text Length (Private Works)

Pada kedua area terlihat bahwa waktu yang diperlukan kedua uji coba berbanding lurus dengan panjang pola dan panjang teks, namun pengaruh pada algoritma Aho-Corasick terlihat jauh lebih kecil dibandingkan pengaruh pada KMP. Dapat disimpulkan bahwa metode Aho-Corasick lebih baik untuk badan teks dan pola yang lebih panjang dibandingkan KMP.

VIDEO LINK AT YOUTUBE

Link: https://youtu.be/qZiIEfuos7s

REFERENCES

Figures that are denoted from private works are all generated by the writer and is available on Github[4]

[1] Genetic Alliance; District of Columbia Department of Health. Washington (DC). (n.d.). Diagnosis of a genetic disease - understanding genetics - NCBI bookshelf. Understanding Genetics: A District of Columbia Guide for Patients and Health Professionals. Retrieved May 23, 2022, from https://www.ncbi.nlm.nih.gov/books/NBK132142/

[2] U.S. National Library of Medicine. (n.d.). T2T-chm13v2.0 - genome - assembly - NCBI. T2T CHM13v2.0 Telomere-to-Telomere assembly of the CHM13 cell line, with chrY from NA24385. Retrieved May 23, 2022, from https://www.ncbi.nlm.nih.gov/assembly/GCF_009914755.1#/st

[3] Melichar, B., Holub, J., & Polcar, T. (2004). TEXT SEARCHING ALGORITHMS VOLUME I: FORWARD STRING MATCHING (Vol. 1). Czech Technical University in Prague Faculty of Electrical Engineering Department of Computer Science and Engineering. Retrieved May 23, 2022, from http://stringology.org/athens/TextSearchingAlgorithms/tsa-lectures-1.pdf

[4] Santoso, N. (n.d.). Nart4hire/MAKALAH_STIMA: Repository Untuk Makalah Stima. GitHub. Retrieved May 23, 2022, from https://github.com/nart4hire/Makalah_Stima

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 23 Mei 2022

Nathanael Santoso 13520129