LAPORAN TUGAS BESAR 3

IF2211 STRATEGI ALGORITMA

Pemanfaatan *Pattern Matching* dalam Membangun Sistem Deteksi Individu Berbasis Biometrik Melalui Citra Sidik Jari

Disusun oleh:

Daniel Mulia Putra Manurung	13522043
Haikal Assyauqi	13522052
Marvin Scifo Y. Hutahaean	13522110

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

DAFTAR ISI

BAB I

Deskripsi Tugas

1.1 Penjelasan Masalah

Di era digital ini, keamanan data dan akses menjadi semakin penting. Perkembangan teknologi membuka peluang untuk berbagai metode identifikasi yang canggih dan praktis. Beberapa metode umum yang sering digunakan seperti kata sandi atau pin, namun memiliki kelemahan seperti mudah terlupakan atau dicuri. Oleh karena itu, biometrik menjadi alternatif metode akses keamanan yang semakin populer. Salah satu teknologi biometrik yang banyak digunakan adalah identifikasi sidik jari. Sidik jari setiap orang memiliki pola yang unik dan tidak dapat ditiru, sehingga cocok untuk digunakan sebagai identitas individu.

Pattern matching merupakan teknik penting dalam sistem identifikasi sidik jari. Teknik ini digunakan untuk mencocokkan pola sidik jari yang ditangkap dengan pola sidik jari yang terdaftar di database. Algoritma *pattern matching* yang umum digunakan adalah Bozorth dan Boyer-Moore. Algoritma ini memungkinkan sistem untuk mengenali sidik jari dengan cepat dan akurat, bahkan jika sidik jari yang ditangkap tidak sempurna.

Dengan menggabungkan teknologi identifikasi sidik jari dan pattern matching, dimungkinkan untuk membangun sistem identifikasi biometrik yang aman, handal, dan mudah digunakan. Sistem ini dapat diaplikasikan di berbagai bidang, seperti kontrol akses, absensi karyawan, dan verifikasi identitas dalam transaksi keuangan.

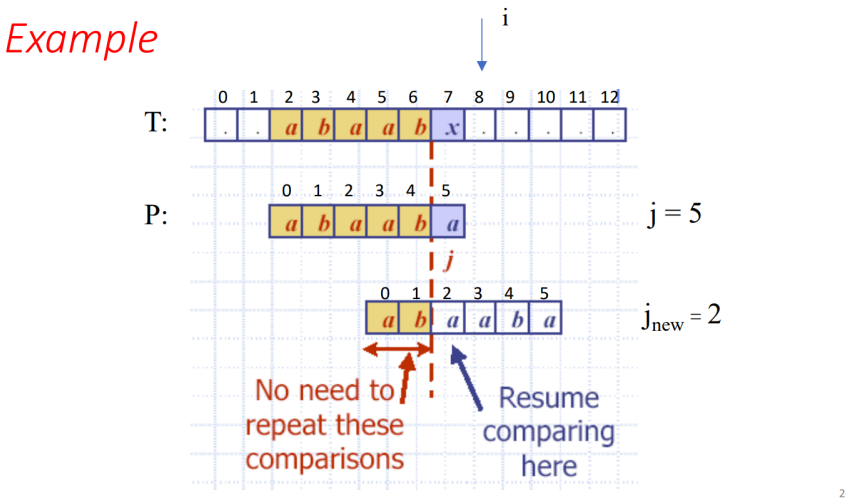
Gambar 1.1.1 - Fingerprint Scanning

BAB II

Landasan Teori

2.1 Algoritma KMP, BM, dan Regex

KMP adalah algoritma string-searching efisien yang digunakan untuk mencari substring dari sebuah pola string di text. Algoritma ini dibuat oleh Donald Knuth, Vaughan Pratt, dan James H. Morris. Fitur yang dimiliki oleh algoritma KMP adalah kemampuannya untuk memproses string pola terlebih dahulu untuk menghindari perbandingan yang berlebihan, sehingga mencapai kompleksitas waktu O(n+m), di mana n adalah panjang teks dan m adalah panjang pola. Algoritma KMP memanfaatkan informasi yang dikumpulkan selama fase pra pemrosesan string pola untuk melewati bagian string teks yang telah dicocokkan dengan pola. Hal ini menghindari evaluasi ulang yang tidak perlu dan secara signifikan mengurangi jumlah perbandingan.

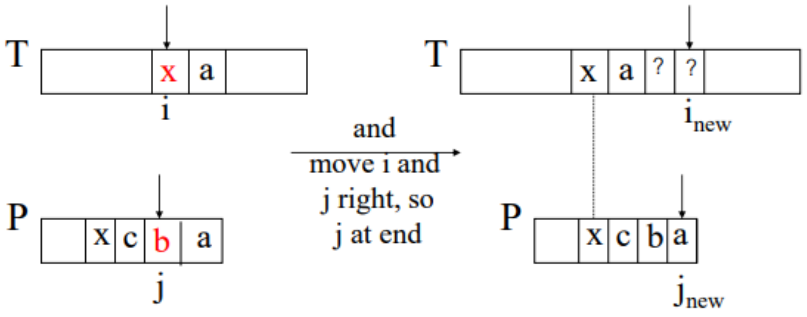


Gambar 2.2.1 - Algoritma KMP

BM adalah algoritma pencocokan pola yang berdasarkan dua teknik yaitu *looking-glass* technique dan *character-jump* technique. Pada *looking-glass*, hal ini melibatkan pencarian P di T secara mundur melalui P dan mulai dari karakter yang paling terakhir. Pada *character-jump*, hal ini dilibatkan ketika terjadi ketidakcocokan dalam salah satu huruf di kata yang ingin diproses dengan kata yang akan dicari kesamaannya akan kata yang ingin di proses. Terdapat 3 kemungkinan kasus yang bisa terjadi.

Case 1

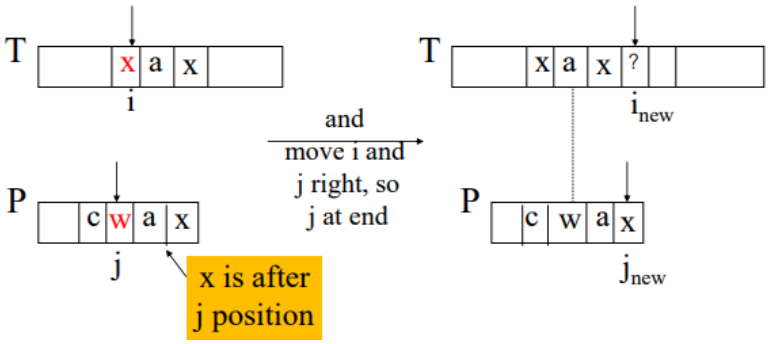
➢If P contains x somewhere, then try to *shift P* right to align the last occurrence of x in P with T[i].



Gambar 2.2.2 - Kasu 1 BM

Case 2

➢If P contains x somewhere, but a shift right to the last occurrence is *not* possible, then *shift P* right by 1 character to T[i+1].

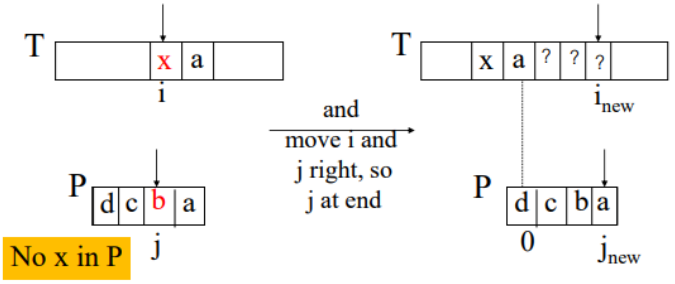


45

Gambar 2.2.3 - Kasus 2 BM

Case 3

➢If cases 1 and 2 do not apply, then *shift* P to align P[0] with T[i+1].



Gambar 2.2.4 - Kasus 3 BM

Regex adalah alat yang digunakan untuk melakukan pattern matching pada string. Alat ini digunakan pada banyak program untuk melakukan pemrosesan teks untuk mencari, mengubah, dan memanipulasi teks berdasarkan pola tertentu. Berikut adalah contohnya.

Berikut adalah pola regex

```
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
```

Penjelasan Regex adalah sebagai berikut.

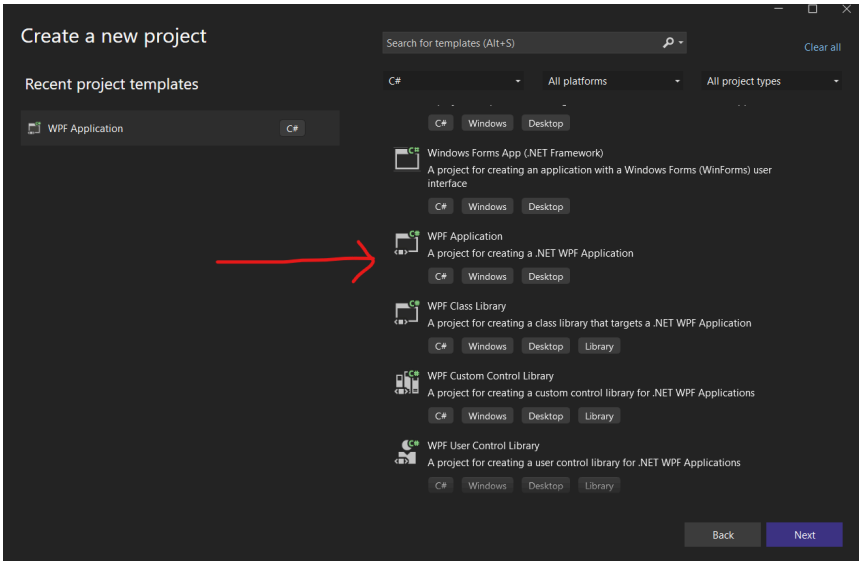
1. ^: Menegaskan posisi di awal string.
2. [a-zA-Z0-9._%+-]+: Cocok dengan satu atau lebih karakter yang dapat berupa:
    a. Huruf kecil (a-z)
    b. Huruf besar (A-z)
    c. Digit (0-9)
    d. Titik (.), garis bawah (_), tanda persen (%), tanda plus (+), atau tanda hubung (-)
3. @: Cocok dengan karakter "@" literal.
4. [a-zA-Z0-9.-]+: Cocok dengan satu atau lebih karakter yang dapat berupa:
    a. Huruf kecil (a-z)
    b. Huruf besar (A-Z)
    c. Digit (0-9)
    d. Titik (.) atau tanda hubung (-)
5. \.: Cocok dengan titik literal (.). Garis miring terbalik keluar dari titik, yang menunjukkan bahwa karakter tersebut harus dicocokkan sebagai karakter literal, bukan sebagai karakter apa pun.
6. [a-zA-Z]{2,}: Cocok dengan dua karakter atau lebih yang dapat berupa:
    a. Huruf kecil (a-z)
    b. Huruf besar (A-Z)
    c. $: Menegaskan posisi di akhir string

2.2 Teknik pengukuran persentase kemiripan

Pada tugas ini, kami tidak serta merta langsung membandingkan gambar secara langsung sehingga kami perlu melakukan konversi akan gambar tersebut supaya perbandingan gambar bisa dilakukan dengan lebih mudah. Konversi yang kami lakukan pada gambar tersebut adalah konversi ke ascii code. Dan pada gambar-gambar yang disimpan di database, kami hanya mengonversi titik tengah binary code dari gambar tersebut ke ascii code.

2.3 Rancangan Aplikasi Desktop

Pada tugas ini, kami menggunakan bahasa C# untuk membuat aplikasi pencocokan sidik jari. C# mempunyai fitur yang bisa melakukan setup akan aplikasi berbasis GUI. Pada tugas ini, kami menggunakan template yang bernama WPF (Windows Presentation Foundation) sebagai GUI dari aplikasi ini.



Gambar 2.3.1 - Pembuatan WPF Application

Aplikasi yang dibuat diharapkan mempunyai komponen-komponen sebagai berikut agar aplikasi bisa berjalan dengan lancar.

- Judul aplikasi
- Tombol Insert citra sidik jari, beserta display citra sidik jari yang ingin dicari
- Toggle Button untuk memilih algoritma yang ingin digunakan (KMP atau BM)
- Tombol Search untuk memulai pencarian
- Display sidik jari yang paling mirip dari basis data
- Informasi mengenai waktu eksekusi

- Informasi mengenai tingkat kemiripan sidik jari dengan gambar yang ingin dicari, dalam persentase (%)
- List biodata hasil pencarian dari basis data. Keluarkan semua nilai atribut dari individu yang dirasa paling mirip. Perlu diperhatikan pendefinisian batas kemiripan dapat memunculkan kemungkinan tidak ditemukan list biodata yang memiliki sidik jari paling mirip.

BAB III

Analisis Pemecahan Masalah

3.1 Langkah-Langkah Pemecahan Masalah

Permasalahan terkait pencarian solusi kecocokan terdekat dari sidik jari bisa diselesaikan dengan algoritma pattern matching. Pattern matching mempunyai berbagai jenis algoritma yang digunakan agar penyelesaian pattern matching tersebut bisa menjadi lebih efisien yaitu KMP dan BM. Namun pada bagian ini, kami hanya akan menjelaskan tahapan pemecahan masalah secara umum saja.

1. Gambar masukan yang berupa fingerprint dikonversi menjadi bentuk bitmap
2. Bitmap dikonversikan ke binary string
3. Binary string dikonversikan ke ASCII string dan mencari titik tengahnya sebesar 128 pixel
4. Gambar-gambar yang akan diperiksa juga dikonversikan ke binary string
5. Setelah itu, konversi gambar-gambar menjadi ascii code agar bisa dilakukan perbandingan dengan gambar masukkan.
6. Hasil yang benar adalah yang persentase kecocokannya paling tinggi.
7. Usai mendapat nama pemilik fingerprint, nama akan dicocokan dengan database biodata menggunakan regex

3.2 Proses penyelesaian solusi dengan algoritma KMP dan BM

1. KMP
    a. Konversi gambar-gambar menjadi ascii code sesuai dengan spesifikasi di pemecahan masalah
    b. Dapatkan fungsi tepi dari ascii code milik gambar-gambar dataset
    c. Fungsi tepi didapat dengan mencocokan prefix dan suffix dari setiap subset string yang ada
    d. Cocokan ascii code gambar dataset dengan gambar masukan
    e. Jika menemukan kesalahan saat pencocokan, gunakkan fungsi tepi untuk menggeser ascii code milik gambar dataset
    f. Cari hamming distance terdekat untuk mendapatkan solusi
2. BM

a. Konversi gambar-gambar menjadi ascii code sesuai dengan spesifikasi di pemecahan masalah

b. Algoritma BM ini mengikuti 2 aturan yaitu bad character rule dan good suffix rule

c. Bandingkan karakter P dengan T mulai dari huruf yang paling kiri

d. Lakukan pengecekan string mulai dari karakter yang paling kanan dari P

e. Dalam algoritma bad character rule, terdapat 3 skenario

    i. P ditemukan dalam T karena seluruh karakter P bersesuaian

    ii. Terdapat mismatch, maka geser P (skip) hingga terdapat karakter dalam P yang bersesuaian dengan mismatch

    iii. Jika karakter mismatch tidak dalam karakter P, maka geser P hingga satu karakter di sebelah kanan T terakhir yang diperiksa

f. Dalam algoritma good suffix rule, jika terdapat mismatch, dalam substring yang telah bersesuaian dalam T, akan diambil substring yang juga terdapat dalam P lalu geser P hingga karakter substring tersebut bersesuaian.

g. Jika tidak terdapat substring yang bersesuaian maka geser P hingga satu karakter di sebelah kanan karakter T terakhir yang diperiksa

h. Untuk setiap pencocokan string, hitung hamming distance dengan membandingkan setiap huruf yang berkorespondensi

i. Pada akhir algoritma, perbandingan hamming distance terkecil akan menjadi solusi

3.3 Pencarian Nama pada Database Nama Kotor Menggunakan Regex

Dalam database yang berisi biodata setiap orang, terdapat nama - nama yang didefinisikan sebagai nama korup. Untuk menentukan dan menentukan apakah nama korup tersebut sama dengan nama yang kita inginkan, maka akan digunakan regular expression. Penerapan regex yang kami gunakan akan diterapkan atau di-generate dari nama - nama korup pada biodata dan bukan nama asli pada berkas citra. Hal ini dikarenakan akan lebih mudah mengekspan nama korup dibandingkan nama asli. Cara yang kami terapkan adalah sebagai berikut

1. Pada awal setiap nama akan dimasukan regex [a-zA-Z]*?, untuk mengenerate karakter awal.

2. Untuk setiap karakter selanjutnya akan dibuat regex yang berkorespondensi dengan karakter tersebut, misalnya, a menjadi [Aa4@], L menjadi [Ll1!], $ menjadi [Ss5$], dan sebagainya.

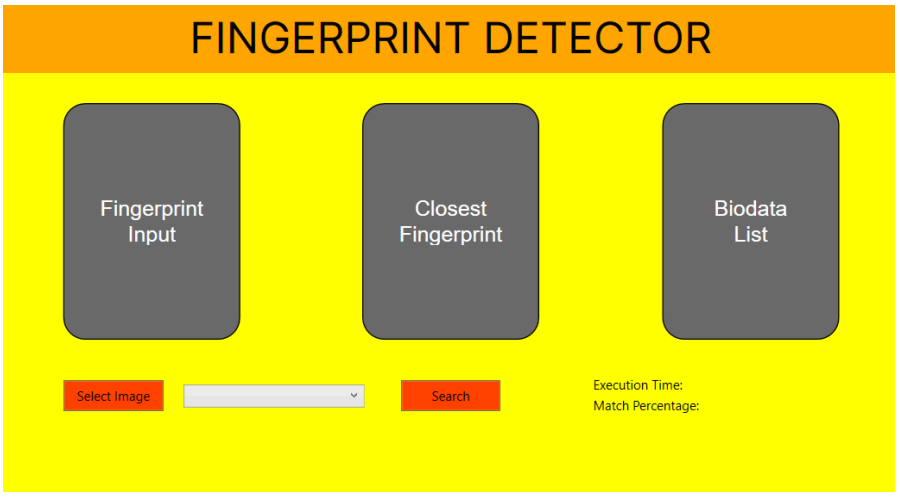
3. Untuk setiap karakter yang dikonversi menjadi regex, akan ditambahkan regex berikut [a-zA-Z]*?, untuk memastikan karakter yang dihilangkan di antara tiap huruf kembali di-generate.

4. Pada akhirnya karakter akhir juga akan ditambahkan [a-zA-Z]*?, untuk memastikan karakter akhir tergenerate.

3.4 Fitur fungsional dan arsitektur aplikasi desktop yang dibangun

Aplikasi desktop ini menggunakan bahasa pemrograman C# dan dikerjakan di sebuah IDE yang bernama Visual Studio. Pada Visual Studio, terdapat tempat untuk membangun WPF. WPF adalah basis dari aplikasi kami. Kami menamai aplikasi ini "Fingerprint Detector".

Bentuk dari aplikasi kami masih sederhana dan beberapa tidak terlalu responsif jika ukuran dari aplikasi desktop diganti. Oleh karena itu, tidak terlalu disarankan untuk maximize jika tidak ingin merubah tampak depan dari aplikasi ini.



Gambar 3.3.1 - Tampak depan dari aplikasi Fingerprint Detector

Aplikasi ini memiliki background yang berwarna kuning dan judul background yang berwarna jingga. Nama judulnya adalah "Fingerprint Detector". Terdapat 3 kotak yang terdiri dari masukkan fingerprint, fingerprint yang terdekat dengan masukan dan biodata dari orang yang bersangkutan. Untuk melakukan input, pengguna bisa klik tombol "Select Image" dan memilih gambar .bmp. Pengguna bisa memilih opsi algoritma di combobox dan bisa melakukan pencarian dengan klik tombol "Search". Pada aplikasi akan ditunjukkan juga waktu eksekusi yang dibutuhkan untuk menjalankan program dan persentase kecocokan dari gambar masukan dengan gambar hasil.

3.5 Contoh Ilustrasi Kasus

1. Pembukaan aplikasi

   Aplikasi biasanya dibuka dengan klik dua kali file executable-nya. File executable dalam bentuk .exe karena tugas ini dibangun dengan menggunakan C# yang notabene selalu menggunakan .exe setelah melakukan kompilasi. Setelah dibuka maka aplikasi desktop akan menunjukkan GUI-nya.

2. Pemilihan gambar

   Pengguna klik tombol “Search Image” untuk mencari gambar yang diinginkan. File yang dipilih disarankan memiliki format .BMP atau .bmp tetapi aplikasi juga memiliki opsi untuk memilih file lain dengan opsi (All file) file-file gambar lainnya seperti .png dan .jpg bisa dicoba meskipun tidak disarankan.

3. Pemilihan algoritma

   Pengguna klik combobox yang terletak di sebelah tombol “Search Image” untuk memilih opsi algoritma yang diinginkan oleh pengguna. Terdapat dua opsi yang dipilih yaitu KMP dan BM.

4. String matching

   Masukan gambar yang didapat akan diproses untuk dicari gambar sidik jari yang mirip dengan algoritma yang diinginkan. Hasil yang didapat adalah solusi yang memiliki hamming distance yang paling kecil.

5. Pengembalian nilai akhir

   Setelah didapat hamming distance, program akan mencocokan nama dengan nama alay yang ada di biodata sehingga bisa diidentifikasi orangnya. Kembalian berupa semua informasi yang ada di biodata, waktu eksekusi, dan persentase kecocokan.

6. Penunjukkan gambar dan daftar biodata

   Setelah semua hasil didapatkan, program akan menunjukkan semua hasil yang didapat ke GUI.

BAB IV

IMPLEMENTASI & EKSPRIMEN

4.1 Spesifikasi teknis program (struktur data, fungsi, dan prosedur yang dibangun)

1. KMP.cs

   a. Kelas

```
public class KMP: IStringSearchAlgorithm
{ Melakukan string matching dengan menggunakan algoritma KMP}
```

   b. Atribut

```
private int[] borderfunction;
```

   c. Fungsi dan Prosedur

```
public KMP() {
        borderfunction = new int[0];
}
{ Constructor kelas KMP }
```

```
public string getPrefix(string kata, int length, int idx)
    {
        int count = 0;
        string result = "";
        while (result.Length < length)
        {
            result = result + kata[count];
            count += 1;
        }
        /*Console.WriteLine("Result: " + result);*/
        return result;
    }
{ Mendapatan prefix dari short word untuk mencari fungsi tepi / pinggiran}
```

```
public string getSuffix(string kata, int length, int idx)
    {
        int count = idx;
        string result = "";
        while (result.Length < length)
        {
            result = kata[count] + result;
            count -= 1;
        }
        /*Console.WriteLine("Result: " + result);*/
        return result;
    }
```

{ Mendapatkan suffix dari short word untuk mencari fungsi tepi / pinggiran }

```
public int getSize(string kata, int idx)
    {
        for (int i = idx; i > 0; i--)
        {
            string prefix = getPrefix(kata, i, idx);
            string suffix = getSuffix(kata, i, idx);
            Console.WriteLine("Prefix: " + prefix);
            Console.WriteLine("Suffix: " + suffix);
            if (prefix == suffix)
            {
                return prefix.Length;
            }
        }

        return 0;
    }
```

{ Mendapatkan fungsi tepi / pinggiran untuk setiap indeks }

```
public void getborderfunction(string pattern) {
        borderfunction = new int[pattern.Length - 1];
        for (int i = 1; i < pattern.Length; i++)
        {
            Console.WriteLine("Index: " + (i - 1));
            borderfunction[i - 1] = getSize(pattern, i - 1);
        }
    }
```

```
public List<(int Position, int HammingDistance, double ClosenessPercentage)> Search(string text, string pattern)
    {
        if (string.IsNullOrEmpty(text))
                throw new ArgumentException("Text cannot be null or empty.");
        if (string.IsNullOrEmpty(pattern))
             throw new ArgumentException("Pattern cannot be null or empty.");

        getborderfunction(pattern);

        string katalengkap = text;
        string kata = pattern;

                List<(int Position, int HammingDistance, double ClosenessPercentage)> results = new List<(int Position, int HammingDistance, double ClosenessPercentage)>();
        int i = 0;
        int idxkata = 0;
        int idxkatalengkap = 0;
        int position = 0;
        int hammming = kata.Length - 1;
        bool found = false;
```

```
                while (found == false && idxkatalengkap < katalengkap.Length)
        {
            if (kata[idxkata] == katalengkap[idxkatalengkap])
            {
                if (idxkata == kata.Length - 1)
                {
                        /*Console.WriteLine("Huruf kata lengkap: " + katalengkap[idxkatalengkap]);
                                Console.WriteLine("Huruf kata: " + kata[idxkata]);
                    Console.WriteLine("Kata ditemukan di index: " + (idxkatalengkap));*/
                    position = idxkatalengkap - idxkata;
                    found = true;
                    results.Add((position, 0, 100.0));
                    return results;
                }
                else
                {
                        /*Console.WriteLine("Huruf kata lengkap: " + katalengkap[idxkatalengkap]);
                                Console.WriteLine("Huruf kata: " + kata[idxkata])*/;
                    idxkata += 1;
                    idxkatalengkap += 1;
                }
            }
            else
            {
                     /*Console.WriteLine("Huruf kata lengkapa: " + katalengkap[idxkatalengkap]);
                            Console.WriteLine("Huruf kataa: " + kata[idxkata]);*/
                if (idxkata == 0)
```

```
                {
                    idxkatalengkap += 1;
                }
                else
                {
                    int gethammming = kata.Length - idxkata;
                    if (gethammming < hammming) {
                        hammming = gethammming;
                        position = idxkatalengkap - idxkata;
                    }
                    idxkata = borderfunction[idxkata - 1];
                }
            }
        }
        if (found == false)
        {
            results.Add((position, hammming, 100 - (double)hammming
/ kata.Length * 100));
        }
        return results;
    }
{Menemukan kemiripan antara substring dan string}
```

2. BM.cs

a. Kelas

```
public class BoyerMoore : IStringSearchAlgorithm
{Melakukan string matching dengan menggunakan BoyerMoore}
```

b. Atribut

```
private readonly Dictionary<char, int> _badCharacterShift;
private int[] _goodSuffixShift;
```

```
private int[] _suffixes;
```

c. Fungsi dan Prosedur

```
public BoyerMoore()
    {
        _badCharacterShift = new Dictionary<char, int>();
    }
{Inisiasi Algoritma Boyer Moore}
```

```
private void PreprocessBadCharacter(string pattern)
    {
        _badCharacterShift.Clear();
        for (int i = 0; i < pattern.Length; i++)
        {
            char currentChar = pattern[i];
            if (!_badCharacterShift.ContainsKey(currentChar))
            {
                 _badCharacterShift[currentChar] = pattern.Length -
i - 1;
            }
        }
    }
{Gerak ketika terdapat huruf yang tidak ada pada last occurence}
```

```
private void PreprocessGoodSuffix(string pattern)
    {
        _goodSuffixShift = new int[pattern.Length + 1];
        _suffixes = new int[pattern.Length + 1];

        int i = pattern.Length;
        int j = pattern.Length + 1;
        _suffixes[i] = j;
```

```
        while (i > 0)
        {
                while (j <= pattern.Length && pattern[i - 1] !=
pattern[j - 1])
            {
                if (_goodSuffixShift[j] == 0)
                    _goodSuffixShift[j] = j - i;
                j = _suffixes[j];
            }
            i--;
            j--;
            _suffixes[i] = j;
        }

        j = _suffixes[0];
        for (i = 0; i <= pattern.Length; i++)
        {
            if (_goodSuffixShift[i] == 0)
                _goodSuffixShift[i] = j;
            if (i == j)
                j = _suffixes[j];
        }
    }
```

```
public IEnumerable<(int Position, int HammingDistance, double
ClosenessPercentage)> Search(string text, string pattern)
    {
        if (string.IsNullOrEmpty(text))
               throw new ArgumentException("Text cannot be null or
empty.");
        if (string.IsNullOrEmpty(pattern))
             throw new ArgumentException("Pattern cannot be null or
empty.");
```

```
        PreprocessBadCharacter(pattern);

        PreprocessGoodSuffix(pattern);


                List<(int Position, int HammingDistance, double
ClosenessPercentage)> results = new List<(int Position, int
HammingDistance, double ClosenessPercentage)>();

        int i = 0;

        while (i <= text.Length - pattern.Length)

        {

            int j = pattern.Length - 1;

            while (j >= 0 && pattern[j] == text[i + j])

                j--;

            if (j < 0)

            {

                results.Add((i, 0, 100.0)); // Exact match

                i += _goodSuffixShift[0];

            }

            else

            {

                char badChar = text[i + j];

                                                int   badCharShift   =
_badCharacterShift.ContainsKey(badChar)                              ?
_badCharacterShift[badChar] : pattern.Length;



                                               int   hammingDistance   =
CalculateHammingDistance(pattern,                text.Substring(i,
pattern.Length));

                                         double  closenessPercentage  =
CalculateClosenessPercentage(hammingDistance, pattern.Length);

                                      results.Add((i,  hammingDistance,
closenessPercentage));



                i += Math.Max(_goodSuffixShift[j + 1], badCharShift
- pattern.Length + 1 + j);
```

```
            }
        }


                    int minHammingDistance = results.Min(r =>
r.HammingDistance);
                results = results.Where(r => r.HammingDistance ==
minHammingDistance).ToList();


        return results;
    }
```

{Melakukan perhitungan}

```
private int CalculateHammingDistance(string str1, string str2)
    {
        if (str1.Length != str2.Length)
                throw new ArgumentException("Strings must be of the
same length");


        int distance = 0;
        for (int i = 0; i < str1.Length; i++)
        {
            if (str1[i] != str2[i])
                distance++;
        }
        return distance;
    }
```

{Menghitung hamming distance}

```
private double CalculateClosenessPercentage(int hammingDistance,
int length)
    {
        return (1 - (double)hammingDistance / length) * 100;
    }
```

{Menghitung closeness Percentage}

3. AlaiRegex.cs

    a. Atribut

```
private List<string> listRegex = new List<string>
    {
        "[Aa4@]",
        "[Bb8]",
        "[Cc\\(]",
        "[Dd]",
        "[Ee3]",
        "[Ff]",
        "[Gg6]",
        "[Hh]",
        "[Ii1!]",
        "[Jj]",
        "[Kk]",
        "[Ll1!]",
        "[Mm]",
        "[Nn]",
        "[Oo0]",
        "[Pp]",
        "[Qq]",
        "[Rr]",
        "[Ss5$]",
        "[Tt7]",
        "[Uu]",
        "[Vv]",
        "[Ww]",
        "[Xx]",
        "[Yy]",
        "[Zz2]",
        "[\\s]*"
```

```
    };
```

b. Fungsi

```
public string StringToRegex(string input)
    {
        string result = "[a-zA-Z0-9]*?";
        int i = 0;
        foreach (char c in input)
        {
            if (c == ' ')
            {
                result += listRegex[26];
            }
            else if (c == '4' || c == '@')
            {
                result += listRegex[0]; // A
            }
            else if (c == '8')
            {
                result += listRegex[1]; // B
            }
            else if (c == '(')
            {
                result += listRegex[2]; // C
            }
            else if (c == '3')
            {
                result += listRegex[4]; // E
            }
            else if (c == '6' || c == '9')
            {
```

```
            result += listRegex[6]; // G
        }
        else if (c == '1' || c == '!')
        {
            result += listRegex[8]; // I
        }
        else if (c == '0')
        {
            result += listRegex[14]; // O
        }
        else if (c == '5' || c == '$')
        {
            result += listRegex[18]; // S
        }
        else if (c == '7')
        {
            result += listRegex[19]; // T
        }
        else if (char.IsLetter(c))
        {
            result += listRegex[char.ToUpper(c) - 65];
        }
        else
        {
            result += Regex.Escape(c.ToString());
        }

        if (i != input.Length - 1)
        {
            result += "[a-zA-Z0-9]*?";
        }
        i++;
```

```
        }
        result += "[a-zA-Z0-9]*?";
        return result;
    }
```

```
public bool TestString2(string input, string pattern)
    {
        Regex regex = new Regex(StringToRegex(pattern));
        Match match = regex.Match(input);
        return match.Success;
    }
```

4.2 Penjelasan tata cara penggunaan program

1. Clone repository git

   clone https://github.com/scifo04/Tubes3_Six-Degrees-Of-Tubes-IF.git

2. Pindah direktori

   cd Tubes3_Six-Degrees-Of-Tubes-IF/src/WpfApp1/WpfApp1

3. Masukkan command

   Install-Package System.Data.SQLite

   Install-Package System.Drawing.Common

   Install-Package MySql.Data

4. Masukkan command

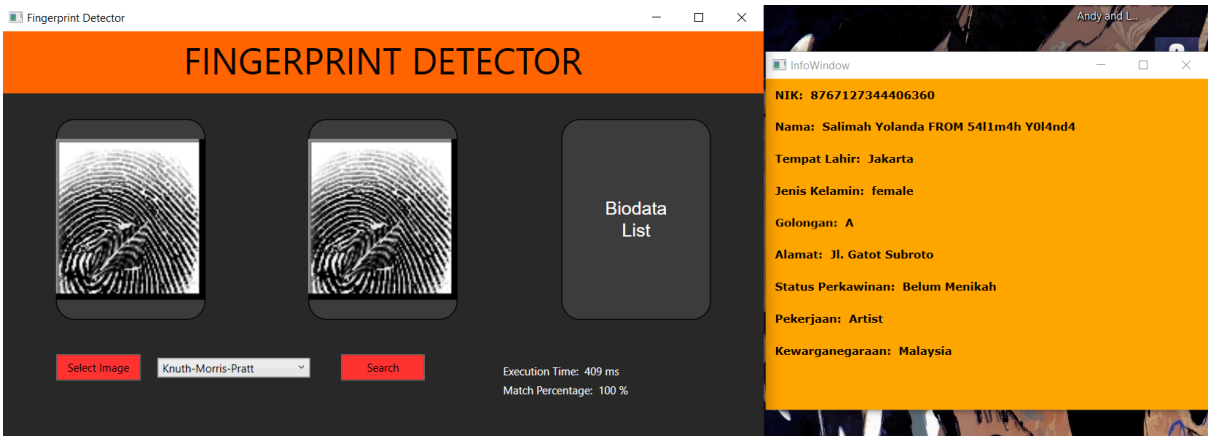
   dotnet run

5. Masukkan gambar yang ingin dicari sidik jarinya
6. Pilih algoritma yang ingin digunakan
7. Klik tombol _search_
8. Tunggu untuk mendapatkan hasil
9. Hasil akan muncul ketika box biodata bila ditekan

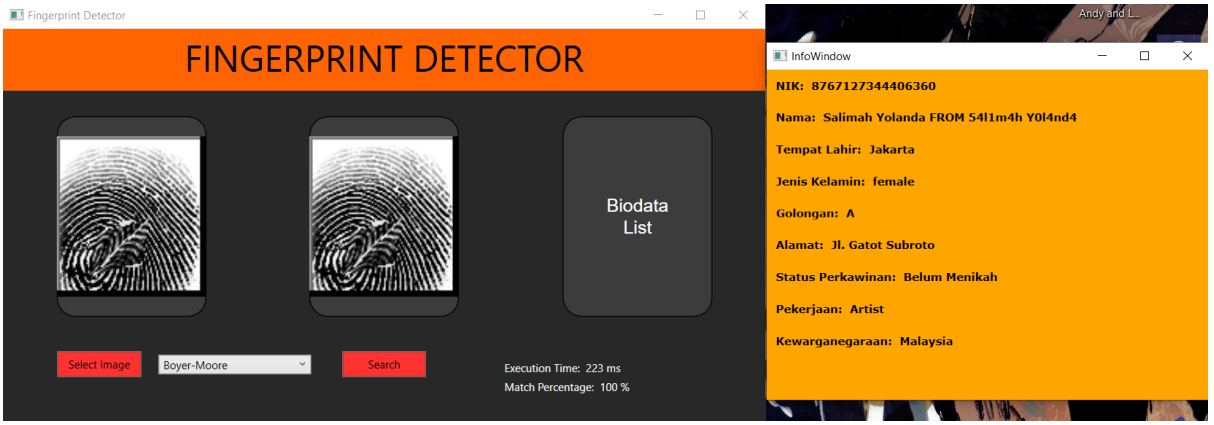
4.3 Hasil pengujian

1. Test Case 1

    - KMP

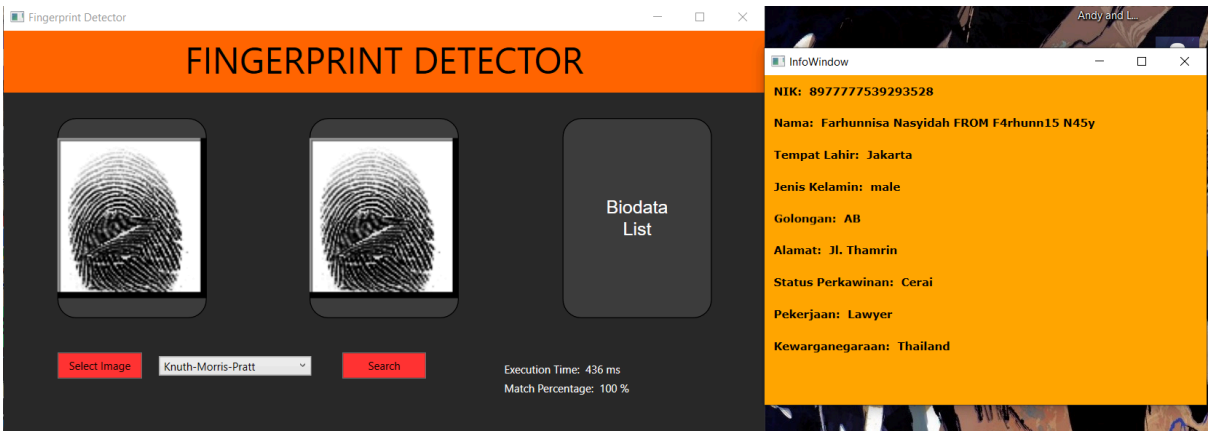


    - BM

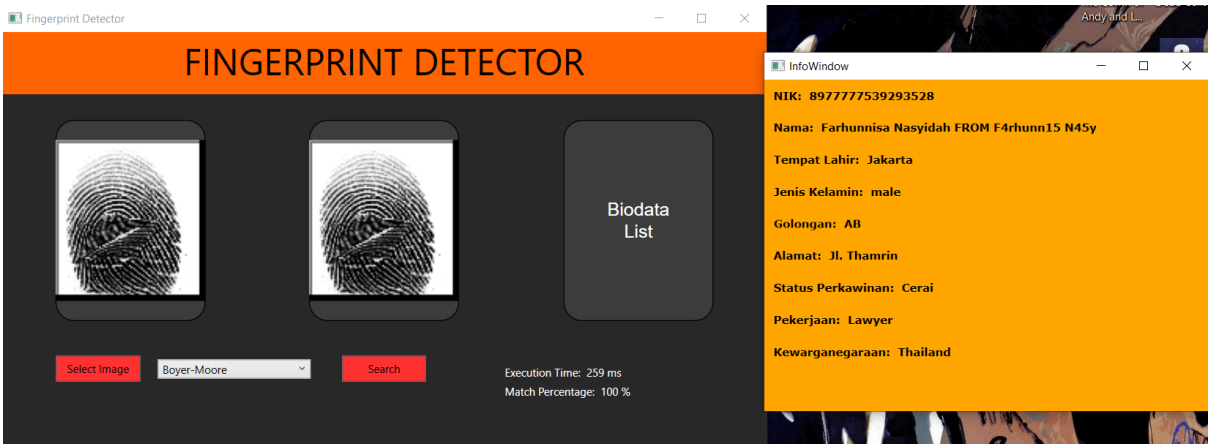


2. Test Case 2

    - KMP



- BM



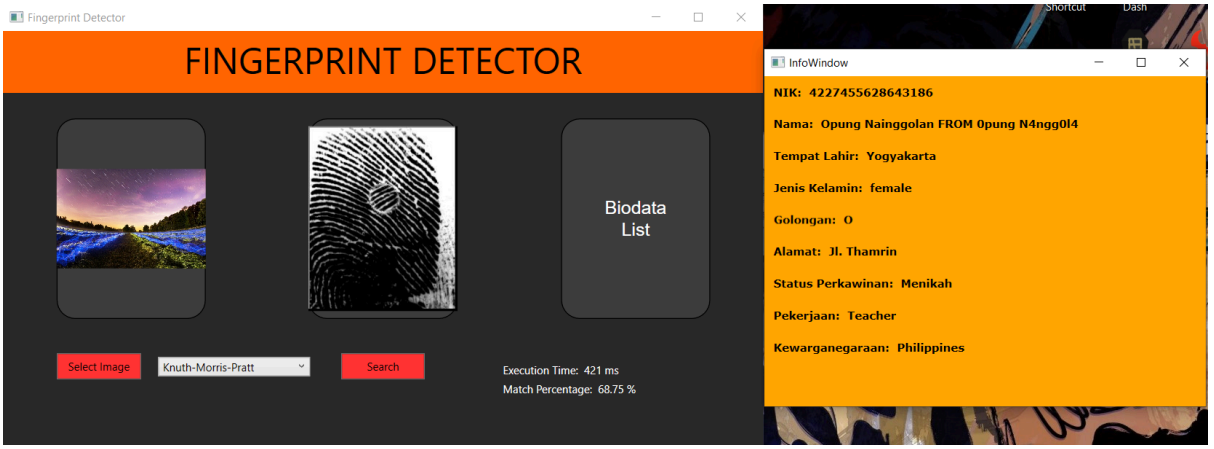
3. Test Case 3

- KMP

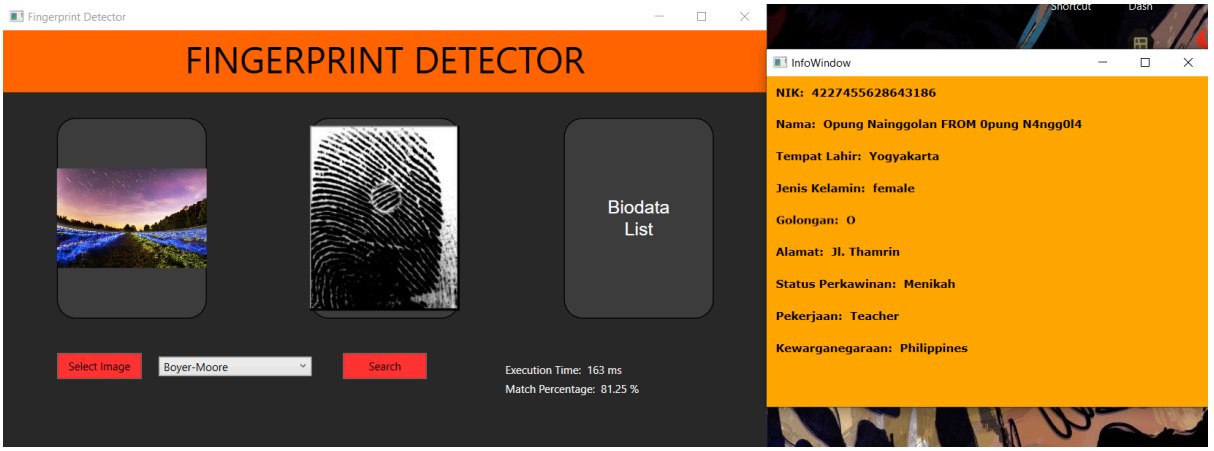
- BM

4. Test Case 4

- KMP



- BM



4.4 Analisis hasil pengujian

Dari hasil pengujian yang telah kami uji, didapatkan untuk gambar yang sangat mirip atau sama dengan gambar yang terdapat dalam basis data, kedua algoritma Knuth Morris Pratt dan Boyer Moore mendapatkan exact match dengan kedekatan 100% untuk gambar

gambar tersebut. Tetapi terdapat sedikit perbedaan yaitu waktu eksekusi, di mana algoritma Knuth Morris Pratt sedikit lebih lambat dibandingkan Boyer Moore. Hal ini dikarenakan karakter yang digunakan adalah karakter ASCII dan pattern yang cukup panjang (128 pixel), sehingga algoritma yang lebih efektif adalah algoritma Boyer Moore.

Hasil pengujian kedua dan ketiga, telah digunakan kambar yang tidak terdapat dalam database, dan gambar yang bukanlah sebuah sidik jari. Berdasarkan uji, didapatkan bahwa algoritma Knuth Morris Pratt dan Boyer Moore dapat memiliki hasil yang berbeda, atau dapat memiliki hasil yang sama, namun nilai hamming distance yang berbeda. Hal ini dikarenakan algoritma Knuth Morris Pratt dan algoritma Boyer Moore keduanya memiliki cara perbandingan dan skip yang berbeda, yang menyebabkan perhitungan hamming distance yang dilakukan untuk setiap comparison, tidak terjadi untuk beberapa kasus pada Boyer Moore yang seharusnya terjadi pada Knuth Morris Pratt dan sebaliknya.

BAB V

Analisis

5.1 Kesimpulan

Tugas Besar 3 IF2211 Strategi Algoritma adalah tugas untuk melakukan pencocokan sidik jari. Untuk menyelesaikan permasalahan ini, algoritma pattern-matching seperti KMP dan BM bisa menjadi algoritma yang cocok untuk diimplementasikan ke permasalahan tersebut. Aplikasi pencocokan sidik jari dikembangkan berbasis desktop. Frontend yang digunakan adalah WPF (C#) sedangkan Backend yang digunakan adalah C#. Aplikasi web menerima masukan berupa pilihan gambar dan pilihan algoritma.

Dari semua yang kami kerjakan di Tugas Besar ini, kami bisa menyimpulkan bahwa algoritma KMP dan GM bisa digunakan untuk menyelesaikan pencocokan sidik jari. KMP melakukan pencocokan dengan melihat fungsi tepi. BM melakukan pencocokan dengan melihat bad character rule dan good suffix rule.

5.2 Saran

Tugas Besar 3 IF2211 Strategi Algoritma Semester II menjadi pelajaran yang baik bagi seluruh anggota kelompok karena ini Tugas Besar yang menggunakan bahasa C# dan salah satu tugas besar yang wajib merancang desktop. Berdasarkan pengalaman anggota kelompok, kami menyarankan beberapa hal ini untuk para pembaca.

1. Mempelajari penggunaan Visual Studio dan WPF agar pengerjaan aplikasi bisa menjadi lebih mudah
2. Mempelajari cara membuat program dengan menggunakan C#

5.3 Tanggapan

Tugas Besar ini menjadi tempat yang baik untuk melakukan eksplorasi C# dan aplikasi berbasis GUI. Namun, dengan banyaknya tugas besar yang dikerjakan, terlalu sulit untuk mempelajari hal-hal tersebut lebih dalam. Harapannya setelah tugas besar ini, kami bisa termotivasi untuk membuat aplikasi GUI yang lebih baik lagi.

5.4 Refleksi

Ada beberapa hal yang kami sarankan untuk pengerjaan Tugas Besar selanjutnya agar pengerjaan menjadi lebih optimal.

1. Mengatur waktu agar kelompok bisa lebih maksimal pengerjaan
2. Lebih sering membaca dokumentasi terkait WPF agar bisa menghasilkan aplikasi desktop yang lebih baik

3. Menyempatkan waktu untuk mengerjakan tugas besar meskipun terdapat tubes yang lain untuk meringankan pekerjaan kelompok saat hari-hari pengerjaan tugas besar yang efektif.

DAFTAR PUSTAKA

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/String-Matching-dengan-Regex-2019.pdf

LAMPIRAN

Link Repository:

https://github.com/scifo04/Tubes3_Six-Degrees-Of-Tubes-IF

Link Video:

https://youtu.be/6JJUEmsRgus