

**LAPORAN TUGAS BESAR II**  
**IF2211 STRATEGI ALGORITMA**

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



**Kelompok Sangku**

Disusun oleh:

10023457	Habibi Galang Trianda
13522019	Wilson Yusda
13522021	Filbert
13522073	Juan Alfred Widjaya

**PROGRAM STUDI TEKNIK INFORMATIKA**  
**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG 2023**

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>2</b>
<b>BAB I DESKRIPSI TUGAS.....</b>	<b>3</b>
<b>BAB II LANDASAN TEORI.....</b>	<b>4</b>
2.1 Algoritma KMP.....	4
2.2 Algoritma BM.....	5
2.3 Regular Expression.....	5
2.4 Algoritma Levenshtein Distance.....	6
<b>BAB III ANALISIS PEMECAHAN MASALAH.....</b>	<b>7</b>
3.1 Langkah-langkah pemecahan masalah.....	7
3.2 Proses penyelesaian solusi dengan algoritma KMP dan BM.....	9
3.3 Fitur fungsional dan arsitektur aplikasi desktop yang dibangun.....	13
3.4 Contoh ilustrasi kasus.....	14
<b>BAB IV IMPLEMENTASI DAN PENGUJIAN.....</b>	<b>19</b>
4.1 Spesifikasi teknis program (struktur data, fungsi, dan prosedur yang dibangun).....	19
4.2 Penjelasan tata cara penggunaan program.....	28
4.3 Hasil pengujian.....	28
4.3.1 Pengujian Algoritma BM.....	28
4.3.2 Pengujian Algoritma KMP.....	31
4.4 Analisis hasil pengujian.....	34
<b>BAB V KESIMPULAN DAN SARAN.....</b>	<b>38</b>
5.1 Kesimpulan.....	38
5.2 Saran.....	39
5.3 Tanggapan dan refleksi.....	39
<b>LAMPIRAN.....</b>	<b>41</b>
Pranala Repository.....	41
Pranala Video.....	41
<b>DAFTAR PUSTAKA.....</b>	<b>42</b>

## **BAB 1**

### **DESKRIPSI TUGAS**

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.

Di dalam Tugas Besar 3 ini, Diminta untuk mengimplementasikan sistem yang dapat melakukan identifikasi individu berbasis biometrik dengan menggunakan sidik jari. Metode yang akan digunakan untuk melakukan deteksi sidik jari adalah Boyer-Moore dan Knuth-Morris-Pratt. Selain itu, sistem ini akan dihubungkan dengan identitas sebuah individu melalui basis data sehingga harapannya terbentuk sebuah sistem yang dapat mengenali identitas seseorang secara lengkap hanya dengan menggunakan sidik jari.

## BAB II

### LANDASAN TEORI

#### 2.1 Algoritma KMP

Algoritma Knuth-Morris-Pratt (KMP) adalah teknik pencocokan string yang dirancang untuk mencari sebuah pola dalam teks secara efisien tanpa mengalami backtracking yang berlebihan. Dikembangkan oleh *Donald Knuth, Vaughan Pratt, dan James H. Morris* pada tahun 1977, KMP mengoptimalkan proses pencarian dengan menggunakan tabel lompatan atau prefix table yang dibuat sebelum pencocokan dimulai. Algoritma ini menggunakan teknik yang sangat efisien untuk pencocokan string yang melibatkan penggunaan sebuah fungsi khusus yang dikenal sebagai *border function*. Fungsi ini memainkan peran penting dalam menentukan seberapa jauh pola harus digeser ketika terjadi ketidakcocokan karakter selama proses pencocokan. Border function ini dihitung sebelum proses pencocokan sebenarnya dimulai dan bertujuan untuk mengurangi jumlah pemeriksaan yang tidak perlu dengan memanfaatkan informasi dari ketidakcocokan yang terjadi.

Dalam pembuatan *border function*, algoritma mencari untuk setiap posisi dalam pola dan panjang maksimum dari *prefix* yang juga merupakan *suffix*. Misalnya, jika pola gagal cocok pada titik tertentu dalam teks, algoritma tidak perlu memulai pencocokan dari awal pola lagi, melainkan dapat melompat langsung ke bagian pola yang berpotensi bisa cocok berdasarkan informasi dari border function. Ini mengurangi waktu yang terbuang untuk memeriksa karakter-karakter yang sudah pasti tidak akan cocok, sehingga mempercepat proses pencocokan secara keseluruhan.

*Border function* efektif mengurangi kompleksitas operasi yang diperlukan dalam pencocokan string. Dengan mengetahui panjang dari bagian yang cocok dari pola yang sebelumnya telah dicocokkan, KMP dapat menghindari pengulangan pencocokan karakter yang telah gagal cocok. Proses ini menjadikan kompleksitas waktu algoritma KMP menjadi  $O(n + m)$ , di mana  $n$  adalah panjang teks dan  $m$  adalah panjang pola, memastikan bahwa setiap karakter dalam teks hanya perlu diperiksa sekali. Keuntungan utama dari KMP adalah tidak memerlukan backtracking, yang berarti bahwa setiap karakter dalam teks hanya diperiksa sekali. Hal ini menjadikan KMP sangat cocok untuk pencarian dalam teks yang panjang atau di mana pencocokan string yang sering diperlukan.

## 2.2 Algoritma BM

Algoritma Boyer-Moore (BM) merupakan salah satu metode pencocokan string yang sangat efisien, terutama diketahui karena kecepatan pencarinya yang lebih tinggi dibandingkan algoritma pencocokan string lainnya. BM menggunakan heuristik untuk mengoptimalkan jumlah pergeseran yang dilakukan selama pencocokan string dengan salah satu heuristik utamanya yaitu bad character. Heuristik ini memainkan peran penting dalam mengidentifikasi karakter yang tidak cocok dan menentukan seberapa jauh pola harus digeser dalam teks.

Heuristik bad character pada dasarnya memanfaatkan informasi dari karakter yang tidak cocok untuk mempercepat pencarian. Saat terjadi ketidakcocokan karakter dalam teks, algoritma BM akan melihat karakter yang tidak cocok tersebut dan memeriksa posisi terakhir karakter tersebut dalam pola. Jika karakter yang tidak cocok tidak ada dalam pola, maka pola dapat digeser sejauh panjang pola. Namun, jika karakter tersebut ada dalam pola, algoritma akan menggeser pola sedemikian rupa sehingga karakter terakhir yang cocok dengan karakter yang tidak cocok di teks akan berada di sebelah kanan posisi ketidakcocokan.

Pendekatan ini memungkinkan algoritma BM untuk melewati banyak karakter yang tidak mungkin menjadi awal dari kecocokan sehingga secara signifikan mengurangi jumlah perbandingan yang perlu dilakukan dibandingkan dengan metode pencocokan string lainnya. Efisiensi ini menjadikan BM pilihan yang sangat baik untuk pencarian dalam teks yang sangat panjang atau dalam database string besar karena dapat mengurangi waktu pencarian secara signifikan.

## 2.3 Regular Expression

Regular expressions (regex) adalah metodologi yang sangat efektif untuk mencari dan memanipulasi string berdasarkan pola yang didefinisikan. Alat ini memungkinkan pengembang untuk melakukan berbagai operasi teks seperti pencarian, penggantian, dan pemisahan berdasarkan pola kompleks yang ditentukan sebelumnya. Komponen utama dari regex meliputi karakter literals, yang mencari kecocokan persis sesuai karakter yang ditentukan, metacharacters yang mengubah aturan pencarian (seperti `\*`, `+`, `?`, `{}`, `[]`, `^`, `\$`, `.`), dan teknik escaping untuk menangani metacharacters sebagai *literals*.

Dalam regex, grup dan alternasi sangat penting; capturing groups `()` memungkinkan pengelompokan ekspresi dan menangkap sub-string yang cocok, sementara non-capturing groups

`((?...))` melakukan hal yang sama tanpa menyimpan substring. Alternation (`|`) digunakan untuk mencocokkan salah satu dari beberapa sub-pola. Regex juga mengandung assertions seperti lookahead dan lookbehind yang membantu melakukan pencocokan berdasarkan elemen sebelum atau setelah tanpa mengikutsertakan elemen tersebut dalam hasil akhir.

Kompleksitas waktu dalam eksekusi regex dapat bervariasi tergantung pada pola dan teks yang diberikan, dengan beberapa kasus mungkin mengalami pertumbuhan waktu eksekusi secara eksponensial, terutama ketika pola melibatkan banyak *backtracking*. Namun, library regex modern sering mengoptimalkan operasi ini melalui kompilasi pola menjadi state machines yang efisien, mengurangi overhead yang terkait.

## 2.4 Algoritma Levenshtein Distance

Algoritma Levenshtein Distance adalah metode yang digunakan untuk mengukur perbedaan antara dua string. Algoritma ini dinamai menurut ilmuwan Vladimir Levenshtein yang memperkenalkannya pada tahun 1965. Prinsip utama dari algoritma ini adalah menghitung jumlah minimum operasi penyisipan, penghapusan, atau substitusi yang diperlukan untuk mengubah satu string menjadi string lain.

Dalam praktiknya, Levenshtein Distance dihitung menggunakan teknik pemrograman dinamis (DP). Algoritma ini membangun matriks dengan ukuran yang sesuai dengan panjang kedua string, dan setiap sel dalam matriks diisi dengan jarak minimum yang diperlukan untuk mengubah substring dari string pertama menjadi substring dari string kedua hingga titik itu. Proses ini dimulai dari dasar, dengan kondisi awal yang menetapkan biaya nol untuk transformasi string kosong, dan kemudian secara bertahap membangun solusi untuk substring yang lebih besar hingga mencakup seluruh string.

Kompleksitas waktu dari algoritma Levenshtein Distance adalah  $O(mn)$ , di mana  $m$  dan  $n$  adalah panjang dari dua string yang dibandingkan. Meskipun efisien untuk string dengan panjang yang relatif kecil, kompleksitas ini bisa menjadi masalah untuk string yang sangat panjang karena memerlukan ruang dan waktu yang besar untuk komputasi.

## **BAB III**

### **ANALISIS PEMECAHAN MASALAH**

#### **3.1 Langkah-langkah pemecahan masalah**

Berikut adalah langkah-langkah pemecahan masalah untuk melakukan pencarian berdasarkan citra yang diberikan, menggunakan beberapa algoritma pattern matching:

Langkah-langkah pemecahan masalah:

1. Inisialisasi dan Preprocessing:

- a. Mengubah citra menjadi citra dengan representasi binary tiap pixel direpresentasikan dengan 0 dan 1. 0 jika pixel tersebut gelap dan 1 jika pixel tersebut terang.
- b. Mengubah citra dengan representasi binary menjadi representasi ASCII 8-bit penuh dan representasi ASCII 8-bit terbaik berdasarkan piksel. Tiap 8 pixel dijadikan satu karakter ASCII. Pemilihan ASCII 8-bit terbaik yang nantinya digunakan untuk pattern matching dilakukan dengan memilih 32 piksel secara berurutan dengan pergantian dari 0 ke 1 atau 1 ke 0 terbanyak (32 pixel terunik). 32 pixel ini merupakan representasi dari citra tersebut untuk dilakukan pattern matching terhadap basis data citra.
- c. Pemilihan representasi citra dengan 32 piksel dilakukan untuk menjaga integritas data dan efisiensi dalam proses pencocokan pola. Setiap karakter ASCII 8-bit merepresentasikan 8 piksel biner (0 atau 1), sehingga diperlukan jumlah piksel yang dapat dibagi secara sempurna oleh 8 untuk menciptakan representasi yang efisien. Dalam hal ini, 32 piksel dapat dibagi sempurna menjadi 4 karakter ASCII 8-bit, yang memastikan tidak ada data yang hilang atau terpotong saat mengonversi dari piksel biner ke representasi ASCII. Penggunaan 30 piksel, seperti yang disebutkan dalam beberapa jurnal, akan mengakibatkan representasi yang tidak sempurna karena 30 tidak dapat dibagi sempurna oleh 8, yang dapat menyebabkan kehilangan sebagian data atau representasi yang tidak akurat. Dengan memilih 32 piksel, setiap piksel dapat dikonversi secara akurat dan lengkap ke dalam karakter ASCII 8-bit, sehingga menjaga integritas data citra. Selain itu, pemilihan 32 piksel secara berurutan dengan pergantian dari 0 ke 1

atau 1 ke 0 terbanyak (32 piksel terunik) bertujuan untuk menangkap fitur unik dari citra tersebut, di mana transisi ini menunjukkan perubahan intensitas yang signifikan dalam citra, yang merupakan karakteristik penting untuk pencocokan pola. Dengan memilih blok 32 piksel yang memiliki transisi terbanyak, kita memastikan bahwa representasi tersebut mencerminkan bagian citra yang paling berbeda dan khas, sehingga meningkatkan akurasi pencocokan pola terhadap basis data citra. Blok 32 piksel memberikan ukuran yang cukup untuk menangkap detail penting dari citra tanpa terlalu banyak data yang dapat menghambat efisiensi algoritma pencocokan pola. Representasi yang ringkas namun informatif ini memungkinkan pencocokan pola yang cepat dan akurat terhadap basis data citra yang besar. Jika pengambilan blok 32 piksel tidak memungkinkan ( $< 32$  akibat ukuran tidak habis dibagi 32), maka akan diambil piksel sisa namun masih berupa kelipatan 8 untuk merepresentasikan biner.

- d. Mendapatkan daftar sidik jari dari basis data yang sudah didekripsi dan diproses. Daftar sidik jari dari basis data juga diproses menjadi representasi ASCII 8-bit penuh.
2. Pencarian Menggunakan Algoritma KMP:
  - a. Jika algoritma yang diminta adalah KMP (Knuth-Morris-Pratt), lakukan pencarian menggunakan KMP dengan pattern merupakan representasi ASCII terbaik dari citra masukan dan text merupakan representasi ASCII citra dari basis data.
  - b. Jika ditemukan, lanjut ke pencarian biodata.
  - c. Jika tidak ditemukan, lanjutkan dengan pencarian menggunakan LD (Levenshtein Distance).
3. Pencarian Menggunakan Algoritma BM:
  - a. Jika algoritma yang diminta adalah BM (Boyer-Moore), lakukan pencarian menggunakan BM dengan pattern merupakan representasi ASCII terbaik dari citra masukan dan text merupakan representasi ASCII citra dari basis data .
  - b. Jika ditemukan, lanjut ke pencarian biodata.
  - c. Jika tidak ditemukan, lanjutkan dengan pencarian menggunakan LD (Levenshtein Distance).
4. Pencarian Menggunakan Algoritma Levenshtein Distance:

- a. Jika algoritma yang diminta tidak ditemukan dengan KMP atau BM, lakukan pencarian menggunakan LD dengan representasi ASCII penuh.
  - b. Hitung jarak Levenshtein antara citra yang diberikan dan setiap sidik jari.
  - c. Jika ditemukan kemiripan yang cukup tinggi (minimal 60%), lanjut ke pencarian biodata.
5. Mencari Biodata:
    - a. Mencari biodata berdasarkan nama dari hasil pencarian dengan nama di biodata yang didapat dari basis data menggunakan pencocokan regex Alay, jika tidak ditemukan maka ulangi pencarian dengan algoritma Levenshtein Distance dengan tingkat kemiripan terbesar.
    - b. Jika biodata ditemukan, lanjut ke pengembalian hasil.
  6. Mengembalikan Hasil:
    - a. Kembalikan hasil pencarian, termasuk biodata, algoritma yang digunakan, persentase kemiripan, waktu eksekusi, dan path citra hasil.

### 3.2 Proses penyelesaian solusi dengan algoritma KMP dan BM.

#### 1. Proses penyelesaian solusi dengan algoritma KMP

Inisialisasi pencarian dengan algoritma KMP dilakukan dengan membuat *border function*. *Border function* mencatat apakah kemunculan dari tiap karakter pada *Pattern* mengalami repetisi jika dilihat dari *index* awal pada *pattern*. Untuk setiap kecocokkan akan dilakukan penambahan nilai sebanyak 1. Contohnya yaitu *pattern* ABABCD akan memiliki nilai 001200 karena AB sudah mengalami repetisi dan CD tidak terkandung pada karakter sebelumnya.

Konsep pencarian KMP dimulai dengan mengecek dari index 0 *pattern* dengan index pencarian pada *text* (tidak selalu 0, namun dimulai dari 0). Untuk setiap kesamaan, akan dilakukan pengecekan pada *index* setelahnya. Namun, jika ditemukan ketidakcocokan, maka akan melihat nilai *border function* dari karakter sebelum karakter ditemukan kesalahan. Nilai dari *border function* pada *index* sebelum *index* kesalahan ditemukan akan dianggap sebagai *index pattern* pada posisi tersebut. Ilustrasi kasus dapat dilihat pada *text* ABCAAB dan *pattern* ABACAB. *Index pattern* dan *index text* dimulai dari 0. Pada *index* ke-5 pada *pattern* dan *text*, terdapat ketidakcocokan. Oleh karena itu akan dilihat *border function* *index* ke-4 yaitu 1. Oleh karena itu, *index pattern* menjadi 1.

Pada program ini, *shifting* tidak ditunjukkan secara terang-terangan, namun dilakukan dengan terus  *mengupdate* posisi *index pattern* sehingga akan bekerja layaknya perubahan posisi *pattern* terhadap *text*, contohnya pada ilustrasi sebelumnya akan mengubah *index pattern* menjadi 1, sesuai dengan nilai *index* ke-4 pada *border function*. Jika *index pattern* dideklarasikan sebagai 0 atau ditemukan ketidakcocokan pada *index 0*, maka akan melanjutkan *index text* karena dalam implementasi sebenarnya akan meng-*shifting pattern* sebanyak + 1 dan mempertahankan *index pattern*.

Secara gambaran umum, mungkin bisa diasumsikan terjadi *shifting* sehingga *index* ke-1 dari *pattern* memiliki posisi yang sama dengan *index-5* pada *text*. Secara bertahap, proses pencarian dengan algoritma KMP dapat dilihat pada tahapan dibawah:

1. Inisialisasi *border function* dari *pattern* yang dijadikan parameter fungsi
  2. Dilakukan perulangan sampai didapat kecocokan atau fungsi menyatakan tidak ada kecocokan akibat jumlah karakter *pattern* > jumlah karakter *text* yang perlu diperiksa.
  3. Untuk setiap perulangan, dengan kedua *index* pencarian dimulai dari 0, akan dilakukan pengecekan dari *index-0*. Jika sampai *index-(Panjang pattern-1)* masih cocok, maka akan dianggap sesuai atau ditemukan pola.
  4. Jika ditemukan ketidakcocokan, akan melihat pada *border function* pada *index* sebelum *index* ditemukan kesalahan. Nilai dari posisi tersebut akan dijadikan acuan untuk *index pattern*. Pencocokan akan dilakukan mulai dari *index pattern* yang baru , yaitu nilai yang diambil dari *border function* pada *index* sebelum *index* ditemukan kesalahan. Jika nilainya bernilai 0, maka *index* pencarian *text* akan dilanjutkan karena 0 tidak dapat di-*shifting* lebih mundur lagi.
2. Proses penyelesaian solusi dengan algoritma BM

Inisialisasi dimulai dengan mengubah kedua parameter input menjadi array of character serta menginisialisasi heuristic dengan *generateBadCharArray*. Pengecekan karakter dimulai dari index paling belakang dan akan semakin maju jika cocok. Jika ada ketidakcocokan, akan dilakukan shifting sesuai *BadCharArray*. *BadCharArray* digunakan untuk mencari *LastOccurrence* untuk tiap karakter yang ada ( setiap karakter ASCII ). Untuk karakter yang tidak ada pada pattern akan dianggap memiliki indeks -1

karena inisialisasi array untuk 255 karakter ASCII diawali dengan nilai -1 . Perhitungan shifting dilakukan dengan formula:

$$shift = \max(1, index_{ketidakcocokkan} - badChar[karaktersalah])$$

Secara analogi, *shifting* yaitu pemindahan posisi pencocokan pattern terhadap *index* dari *text*. Contohnya jika suatu pencarian dilakukan dari *index* ke-7 dari pattern dan *index* ke-7 dari *text*, dan dilakukan *shifting* + 1 , maka *index* ke-7 dari *text* maka sekarang akan memiliki posisi sama dengan *index* ke -6 *pattern*, karena posisi mulai dari *pattern* akan dimajukan sebanyak 1 karakter.

Dalam konteks formula diatas, jika ditemukan karakter tidak cocok, dilakukan shifting sebanyak 1 dan akan dicek apakah sudah serupa. Jika belum akan dilakukan shifting lagi, sampai ditemukan karakter yang cocok dengan karakter salah atau sampai keseluruhan pattern sudah didepan karakter salah tadi.

Jika setelah suatu pencarian , didapatkan bahwa namaAlay bersifat null, maka akan dilakukan pencarian dengan *threshold* 60% dengan menggunakan algoritma LD. Secara umum, konsep pencarian dengan menggunakan algoritma BM yaitu sebagai berikut:

1. Memulai inisialisasi data yang akan digunakan, yaitu dengan mengubah *pattern* serta *text* menjadi *array of char*, dan membuat *badChar* yang bertindak sebagai *Last Occurrence Table*. *badChar* dibuat dengan menandakan karakter yang pernah muncul pada *text* dengan nilai index terakhir kemunculan dan karakter yang tidak pernah muncul dengan nilai -1
2. Dilakukan pengecekan kecocokan dengan dari *pattern* dengan *text* sesuai dengan *index*. *Index* dimulai dari panjang *pattern* dan pelan pelan menjadi 0 ( dari belakang ke depan) sesuai dengan implementasi pencarian dengan BM. Untuk setiap kecocokan, akan dilakukan pengurangan *index* , dan jika *index* mencapai 0, maka akan dianggap menemukan kecocokan.
3. Jika selama pengecekan, ditemukan ketidakcocokan, maka akan dilakukan *shifting*. Minimal shifting yaitu 1 untuk memastikan tidak ada nilai minus jika sampai posisi karakter pada *pattern* yang cocok dengan *text* pada *index* ketidakcocokan ada pada posisi paling ujung.

4. *Shifting* dilakukan dengan mencari apakah untuk karakter *text* yang tidak cocok tersebut ada pada *pattern* dengan *index* lebih kecil dari *index* kesalahan ditemukan. Oleh karena itu, *shifting* akan dilakukan sampai bertemu salah satu dari 2 kemungkinan, antara karakter pada *index* kesalahan tersebut menjadi cocok akibat bertemu dengan karakter pada *pattern* yang sesuai, ataupun keseluruhan *pattern* dimulai pada *index* setelah *index* kesalahan tersebut. Cara pencarian apakah karakter ketidakcocokan ada pada *pattern* dapat dilakukan dengan mengecek *LastOccurrence* pada *badChar*.
3. Proses penyelesaian solusi dengan algoritma LD

Metode Get menginisialisasi array 2D dp di mana dp[i, j] merepresentasikan jumlah minimum edit yang diperlukan untuk mengubah i karakter pertama dari text1 menjadi j karakter pertama dari text2. Metode ini mengisi array dp dengan mengiterasi setiap karakter dari kedua string. Jika salah satu string kosong, jaraknya adalah panjang string lainnya (semua operasi insert atau delete). Untuk setiap perbandingan karakter, jika karakter-karakter tersebut cocok, tidak ada biaya yang ditambahkan; sebaliknya, jika karakter tidak cocok, biaya sebesar 1 ditambahkan. Nilai dp[i, j] ditentukan dengan mengambil nilai minimum dari tiga operasi yang mungkin: insert (menambah karakter), delete (menghapus karakter), dan substitute (mengganti karakter). Akhirnya, metode ini mengembalikan nilai dp[text1Length, text2Length], yang merupakan jarak Levenshtein antara text1 dan text2 secara keseluruhan. Operasi cost hanya mengembalikan 2 nilai , yaitu 0 atau 1 dan mengembalikan nilai 1 jika text1 pada *index* i-1 tidak sama dengan text2 pada *index* j-1 pada iterasi i dan j.

Secara umum, konsep pencarian dengan algoritma LD yaitu sebagai berikut:

1. Inisialisasi text1Length dan text2Length ( panjang tiap *text* )
2. Dilakukan iterasi dengan membentuk isi dari matriks dengan jumlah row sebanyak panjang *text* 2 dan kolom sebanyak panjang *text* 1.
3. Pengisian dilakukan dengan mengecek apakah index dari i atau j sewaktu perulangan adalah 0, jika bernilai 0 , maka akan diambil nilai dari j dan i secara berurutan pada saat i ataupun j tersebut bernilai 0.
4. Jika tidak, maka akan ada perhitungan *cost*. Nilai dp[i, j] ditentukan dengan mengambil nilai minimum dari tiga operasi yang mungkin: *insert*

(menambah karakter), *delete* (menghapus karakter), dan *substitute* (mengganti karakter), dimana cost akan ditambahkan dalam proses *substitute*.

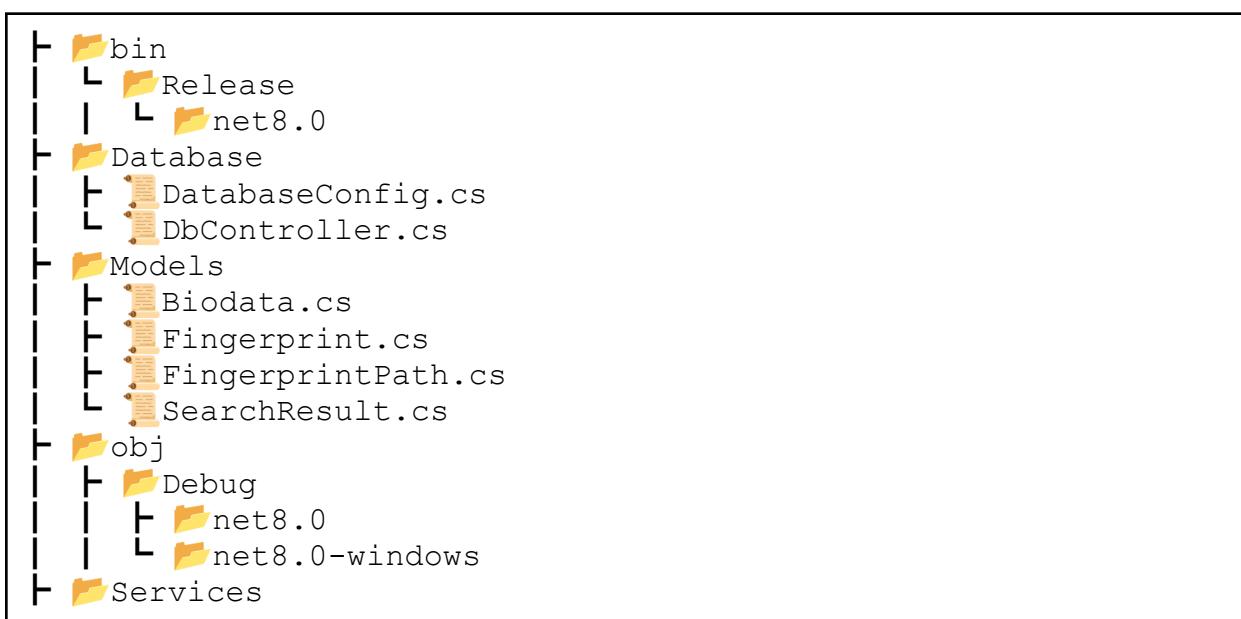
5. Dikembalikan nilai matriks pada *index* (*text1Length* , *text2Length*)

### 3.3 Fitur fungsional dan arsitektur aplikasi desktop yang dibangun.

Perancangan aplikasi menggunakan framework WPF dan didukung dengan bahasa pemrograman C# dalam menerapkan algoritma dalam proses kalkulasi. Fitur yang dimiliki yakni berupa:

- Fitur penerimaan inputan gambar dengan tipe file .BMP yang didukung dengan penampilan gambar dari hasil inputan pengguna
- Fitur pemilihan algoritma dengan menggunakan radio button dan tersedianya 2 pilihan yakni algoritma BM dan KMP
- Fitur pencarian yang didukung dengan adanya database sebagai basis pencarian. Pencarian memanggil fungsi pencarian dan menggunakan gambar dari hasil inputan untuk diproses dengan bahasa pemrograman C#.
- Fitur penampilan hasil pencarian , baik dari informasi lengkap hasil pencarian serta penampilan gambar yang dianggap memiliki kemiripan yang tinggi dengan gambar inputan pengguna.

Adapun arsitektur ( folder src ) dari aplikasi yang dibangun yaitu:





### 3.4 Contoh ilustrasi kasus

Berikut adalah ilustrasi kasus jika gambar masukan seperti berikut:

Gambar memiliki nilai RGB (8x8 piksel)

255, 0, 0	255, 0, 0	255, 0, 0	255, 0, 0	0, 255, 0	0, 255, 0	0, 255, 0	0, 255, 0
255, 0, 0	255, 0, 0	255, 0, 0	255, 0, 0	0, 255, 0	0, 255, 0	0, 255, 0	0, 255, 0
0, 0, 255	0, 0, 255	0, 0, 255	0, 0, 255	255, 255, 0	255, 255, 0	255, 255, 0	255, 255, 0
0, 0, 255	0, 0, 255	0, 0, 255	0, 0, 255	255, 255, 0	255, 255, 0	255, 255, 0	255, 255, 0
255, 0, 255	255, 0, 255	255, 0, 255	255, 0, 255	0, 255, 255	0, 255, 255	0, 255, 255	0, 255, 255
255, 0, 255	255, 0, 255	255, 0, 255	255, 0, 255	0, 255, 255	0, 255, 255	0, 255, 255	0, 255, 255
128, 128, 128	128, 128, 128	128, 128, 128	128, 128, 128	0, 0, 0	0, 0, 0	0, 0, 0	0, 0, 0
128, 128, 128	128, 128, 128	128, 128, 128	128, 128, 128	0, 0, 0	0, 0, 0	0, 0, 0	0, 0, 0

Berikut merupakan proses pencarinya:

1. Mengubah Gambar Menjadi Representasi Biner:

Konversi setiap piksel ke grayscale dan kemudian menjadi biner berdasarkan ambang batas (misalnya 128). Piksel yang lebih terang dari 128 menjadi 1 dan yang lebih gelap menjadi 0.

Berikut adalah hasil nya

11110000

00001111

00001111

11110000

11110000

11110000

11110000

2. Mengubah Representasi Biner Menjadi ASCII 8-bit:

Gabungkan setiap 8 piksel menjadi satu karakter ASCII.

Representasi Biner:

"11110000" = ASCII: ð

"11110000" = ASCII: ð

"00001111" = ASCII: ? (Hanya contoh)

"00001111" = ASCII: ?

"11110000" = ASCII: ð

"11110000" = ASCII: ð

"11110000" = ASCII: ð

"11110000" = ASCII: ð

ASCII Penuh: ðððððð

3. Mengubah Representasi Biner Menjadi ASCII 8-bit Terbaik (32 Piksel Terunik):

Pilih 32 piksel dengan transisi terbanyak.

Hasil Representasi Biner Terbaik:

"11110000" = ASCII: ð

"11110000" = ASCII: ð

"00001111" = ASCII: ?

"00001111" = ASCII: ?

ASCII Terbaik: ðð??

#### 4. Melakukan Pencocokan Pola (Pattern Matching):

Gunakan algoritma pencocokan pola untuk membandingkan representasi ASCII 8-bit terbaik (ðð??) dengan basis data.

Misalkan basis data memiliki representasi berikut:

Fingerprint 1: "ððð?ððð?" = Nama: Alice

Fingerprint 2: "ð?ðð??ðð" = Nama: Bob

Fingerprint 3: "?ðððððð" = Nama: Charlie

Pencocokan dilakukan dengan algoritma KMP ataupun BM, dan ditemukan pada Fingerprint 2 dengan Nama Asli: Bob

Jika tidak ditemukan dengan KMP atau BM, gunakan LD untuk mengukur kesamaan dengan representasi ASCII penuh.

Jika menggunakan BM, berikut alurnya:

Proses pencocokan Fingerprint 1:

ð	ð	ð	?	ð	ð	ð	?
ð	ð	?	?				
	ð	ð	?	?			
			ð	ð	?	?	

Karena tidak bisa shift dua kali lagi pada akhir step, maka pattern tidak ditemukan.

Proses pencocokan Fingerprint 2:

ð	?	ð	ð	?	?	ð	ð
---	---	---	---	---	---	---	---

ð	ð	?	?				
		ð	ð	?	?		

Pattern ditemukan pada fingerprint 2.

#### 5. Mencari Biodata Berdasarkan Nama:

Nama asli yang ditemukan (Bob) dicocokkan dengan nama alay dalam basis data menggunakan regex atau LD.

Nama alay dalam basis data:

4lic, b0B, C4rl1

Pertama, Regex untuk pencocokan akan dibuat. Karena Pencocokan ditemukan pada Fingerprint 2 dengan nama asli Bob, maka buat regex dari Bob, yakni [bB8][oO0][bB8] (spasi diabaikan). Dari regex tersebut dicari pencocokan dengan data nama alay pada basis data. Setelah pencocokan ditemukan nama alay yang cocok dengan regex, yakni b0B. Jika ditemukan nama alay yang cocok, maka kembalikan hasil biodata yang mempunyai nama alay tersebut.

#### 6. Mengembalikan Hasil:

Biodata yang sesuai ditemukan (data contoh):

Nama Asli: Bob

Nama Alay: b0B

NIK: 123456789

Tempat Lahir: Jakarta

Tanggal Lahir: 01-01-1990

Jenis Kelamin: Laki-laki

Golongan Darah: O

Alamat: Jalan Merdeka No. 1

Agama: Islam

Status Perkawinan: Belum Kawin

Pekerjaan: Programmer

Kewarganegaraan: Indonesia

Ilustrasi ini menunjukkan bagaimana gambar 8x8 piksel dikonversi menjadi representasi biner, lalu menjadi representasi ASCII 8-bit penuh dan terbaik, kemudian dilakukan pencocokan pola menggunakan algoritma pencarian, dan akhirnya menemukan biodata yang sesuai dalam basis data dengan nama asli Bob dan nama alay b0B.

## **BAB IV**

### **IMPLEMENTASI DAN PENGUJIAN**

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

Program dibagi menjadi berbagai entitas yang dapat saling dihubungkan untuk mendukung kinerja pembuatan tugas besar ini. Terdapat folder Models yang berfungsi sebagai deklarasi tipe untuk berbagai tipe data buatan. Berbagai tipe data buatan yang digunakan yaitu :

##### **1. Biodata**

Atribut dari kelas ini:

- a. NIK (string)
- b. NamaAlay (string)
- c. TempatLahir (string)
- d. TanggalLahir (string)
- e. JenisKelamin (string)
- f. Golongan Darah (string)
- g. Alamat (string)
- h. Agama (string)
- i. StatusPerkawinan (string)
- j. Pekerjaan (string)
- k. Kewarganegaraan (string)

##### **2. Fingerprint**

Atribut dari kelas ini:

- a. PixelAscii (string)
- b. ImagePath (string)
- c. Nama (string)

##### **3. FingerprintPath**

Atribut dari kelas ini:

- a. ImagePath (string)
- b. Nama (string)

##### **4. SearchResult**

Atribut dari kelas ini:

- a. Biodata (Biodata)
- b. Algoritma (string)
- c. *Similarity* (int)
- d. *ExecTime* (double)
- e. *imagePath* (string)

Adapun fungsi yang dideklarasikan dalam kelas ini yaitu:

- a. `getNotFoundResult ( Input:(Double: execTime); Output:(SearchResult) )`

Mengembalikan sebuah kelas `SearchResult` dengan `similarity` 0 dan `execTime` sesuai parameter serta atribut lain bernilai null sebagai pertanda hasil pencarian tidak ditemukan.

Setelah menyelesaikan model dari berbagai struktur data yang dibutuhkan, selanjutnya dikelompokkan berdasarkan fitur yang dibutuhkan. Dalam folder Services terdapat beberapa sub-folder yang dikelompokkan berdasarkan utilitas mereka masing-masing.

Folder pertama yaitu folder Database yang berisi semua deklarasi kelas terkait *database*.

Berbagai deklarasi kelasnya yaitu:

### 1. DatabaseConfig

Atribut yang dimiliki yaitu:

- 1. Server (string)
- 2. User (string)
- 3. Database (string)
- 4. Password (string)
- 5. Port (int)
- 6. ConnectionString (string)

Terdapat pula prosedur yang dideklarasikan yaitu:

- 1. `setConnection (Input: (String:server, user, database, password, Integer:port))`

Fungsi ini melakukan inisialisasi penghubungan kepada database.

### 2. DbController

Kelas ini menyediakan berbagai metode untuk mengelola data yang disimpan dalam database. Kelas ini memiliki 1 atribut bertipe array of uint yang dinamakan `encryptionKey` yang memiliki nilai = { 0x01234567, 0x89ABCDEF, 0xFEDCBA98, 0x76543210 }. Adapun fungsi yang dideklarasikan yaitu:

1. GetDecryptedBiodataList (Output: Array of Biodata)

Fungsi ini bertugas untuk mengambil data dari tabel biodata di database, mendekripsi setiap kolom yang dienkripsi, dan mengembalikannya sebagai daftar objek Biodata. Prosedur ini membuka koneksi ke database, menjalankan perintah SQL untuk mengambil semua baris dari tabel biodata, kemudian membaca hasilnya menggunakan MySqlDataReader. Setiap nilai yang dienkripsi di dekripsi menggunakan algoritma SimpleXTEA dengan kunci enkripsi yang telah ditentukan, dan hasilnya disimpan dalam objek Biodata yang ditambahkan ke daftar hasil.

2. GetDecryptedFingerprintList (Output: Array of FingerprintPath)

Fungsi ini berfungsi untuk mengambil data dari tabel sidik\_jari di database, mendekripsi jalur gambar dan nama yang dienkripsi, dan mengembalikannya sebagai daftar objek FingerprintPath. Prosedur ini membuka koneksi ke database, menjalankan perintah SQL untuk mengambil semua baris dari tabel sidik\_jari, kemudian membaca hasilnya menggunakan MySqlDataReader. Nilai jalur gambar dan nama yang dienkripsi didekripsi menggunakan algoritma SimpleXTEA dan hasilnya disimpan dalam objek FingerprintPath yang ditambahkan ke daftar hasil.

3. GetBiodataList (Output: Array of Biodata)

Fungsi ini mengambil data dari tabel biodata di database tanpa mendekripsi kolom yang dienkripsi. Prosedur ini membuka koneksi ke database, menjalankan perintah SQL untuk mengambil semua baris dari tabel biodata, kemudian membaca hasilnya menggunakan MySqlDataReader. Setiap nilai disimpan dalam objek Biodata yang ditambahkan ke daftar hasil. Metode ini berguna untuk mendapatkan data asli dari database tanpa dekripsi.

4. GetFingerprintList (Output: Array of FingerprintPath)

Fungsi ini mengambil data dari tabel sidik\_jari di database tanpa mendekripsi kolom yang dienkripsi. Prosedur ini membuka koneksi ke database, menjalankan perintah SQL untuk mengambil semua baris dari tabel sidik\_jari, kemudian membaca hasilnya menggunakan MySqlDataReader. Setiap nilai disimpan dalam objek FingerprintPath yang ditambahkan ke daftar hasil. Metode ini berguna untuk mendapatkan data asli dari database tanpa dekripsi.

### 5. EncryptAndSaveBiodata

Prosedur ini mengenkripsi data dari daftar Biodata yang diambil menggunakan GetBiodataList, kemudian menyimpannya kembali ke database setelah dienkripsi. Prosedur ini membuka koneksi ke database, melakukan iterasi melalui setiap objek Biodata, mengenkripsi setiap kolom yang sesuai menggunakan algoritma SimpleXTEA, dan kemudian memperbarui baris dalam tabel biodata menggunakan perintah SQL UPDATE.

### 6. EncryptAndSaveFingerprint

Prosedur ini mengenkripsi data dari daftar FingerprintPath yang diambil menggunakan GetFingerprintList, kemudian menyimpannya kembali ke database setelah dienkripsi. Prosedur ini membuka koneksi ke database, melakukan iterasi melalui setiap objek FingerprintPath, mengenkripsi jalur gambar dan nama menggunakan algoritma SimpleXTEA, menghapus entri lama dari tabel sidik\_jari menggunakan perintah SQL DELETE, dan kemudian menyisipkan entri yang telah dienkripsi menggunakan perintah SQL INSERT.

### 7. EncryptAndSaveData

Prosedur ini adalah wrapper yang memanggil dua metode lainnya: EncryptAndSaveBiodata dan EncryptAndSaveFingerprint. Prosedur ini memastikan bahwa semua data biodata dan sidik jari dienkripsi dan disimpan kembali ke database. Metode ini memberikan cara yang mudah dan terintegrasi untuk mengenkripsi dan menyimpan ulang semua data sensitif dalam tabel biodata dan sidik jari.

Folder kedua yaitu folder Encryption. Folder ini berisi algoritma enkripsi dan dekripsi. Algoritma enkripsi menggunakan algoritma XTEA. Algoritma XTEA (eXtended Tiny Encryption Algorithm) adalah algoritma enkripsi blok yang dirancang untuk mengatasi beberapa kelemahan pada algoritma TEA (Tiny Encryption Algorithm). XTEA menggunakan kunci 128-bit dan bekerja dengan melakukan 32 putaran enkripsi atau dekripsi untuk memproses data dalam blok 64-bit. Algoritma tersebut diimplementasikan secara simpel dalam kelas SimpleXTEA.

SimpleXTEA dimulai dengan mendefinisikan konstanta NumRounds sebanyak 32 putaran dan Delta sebagai nilai konstanta yang digunakan dalam proses enkripsi dan dekripsi.

Pada metode enkripsi, teks input diubah menjadi byte menggunakan encoding UTF-8, kemudian ditambahkan padding agar panjang data menjadi kelipatan 8 byte. Data ini kemudian diproses dalam blok 64-bit, dimana setiap blok dienkripsi menggunakan algoritma XTEA dalam 32 putaran. Hasil enkripsi dikembalikan dalam bentuk string heksadesimal.

Proses enkripsi dimulai dengan mengubah teks asli menjadi byte menggunakan encoding UTF-8. Data tersebut kemudian diproses dalam blok 64-bit, dengan setiap blok dienkripsi dalam 32 putaran. Dalam setiap putaran, dua nilai 32-bit ( $v_0$  dan  $v_1$ ) diubah menggunakan operasi aritmatika dan logika berdasarkan kunci enkripsi dan nilai Delta. Hasil akhir dari setiap blok dienkripsi kemudian digabungkan dan dikembalikan sebagai string heksadesimal.

Metode dekripsi mengubah teks heksadesimal terenkripsi kembali menjadi byte array, kemudian memprosesnya dalam blok 64-bit. Setiap blok didekripsi menggunakan algoritma XTEA dalam 32 putaran. Hasil dekripsi dikembalikan sebagai string UTF-8 setelah menghapus karakter padding yang tidak diperlukan. Proses dekripsi dimulai dengan mengubah string heksadesimal yang terenkripsi kembali menjadi byte array. Data tersebut kemudian diproses dalam blok 64-bit, dengan setiap blok didekripsi dalam 32 putaran. Dalam setiap putaran, dua nilai 32-bit ( $v_0$  dan  $v_1$ ) diubah menggunakan operasi aritmatika dan logika secara terbalik berdasarkan kunci enkripsi dan nilai Delta. Hasil akhir dari setiap blok didekripsi kemudian digabungkan dan dikembalikan sebagai string UTF-8, setelah menghapus karakter padding yang tidak diperlukan.

Selain itu, terdapat metode pembantu yang mengubah string heksadesimal menjadi byte array yang digunakan dalam proses dekripsi. Metode ini memastikan setiap dua karakter heksadesimal diubah menjadi satu byte. Kode ini mencakup semua fungsi dasar untuk melakukan enkripsi dan dekripsi menggunakan algoritma XTEA, memastikan data aman dan dapat diproses dalam format yang sesuai.

Folder ketiga yaitu folder Matcher. Folder ini berisi semua algoritma yang akan dipakai dalam mencocokkan gambar inputan pengguna dengan target. Untuk penjelasan cara kerja dapat dilihat pada penjelasan algoritma pada bagian 3.2 diatas. Adapun kelas kelas dideklarasikan dari folder Matcher ini adalah:

1. AlayPatternMatcher

Kelas ini bertujuan untuk memeriksa apakah sebuah string yang telah diubah menjadi bahasa "alay" cocok dengan pola string asli. Bahasa "alay" adalah bentuk tulisan yang

sering digunakan di media sosial atau pesan singkat, di mana huruf-huruf digantikan dengan angka atau karakter lainnya yang mirip.

Adapun fungsi-fungsi yang dideklarasikan pada kelas ini yaitu:

1. IsMatch ( Input : (String:original, alay) ; Output : boolean )

Fungsi ini bertujuan untuk memeriksa apakah string inputan sesuai dan string "alay" dapat dianggap sama. Fungsi ini digabungkan dengan fungsi MatchPattern dan BuildRegexPattern sehingga untuk memeriksa apakah suatu kata "alay" sama dengan kata original diperlukan pembangunan pola regex berdasarkan kata original dengan BuildRegexPattern, dan dicocokkan pola regex tersebut dengan MatchPattern.

2. MatchPattern ( Input: (String : str, regex) ; Output : boolean)

Fungsi ini bertujuan apakah string inputan sesuai dengan pola regex tertentu. Fungsi ini sebagai perantara dengan fungsi *built-in* pada *regex.cs*

3. BuildRegexPattern ( Input: (String: original); Output: string)

Fungsi ini berfungsi untuk membangun pola regex yang sesuai dengan string asli yang dapat mengenali varian bahasa "alay". Metode ini menerima sebuah string dan membangun sebuah pola regex yang bisa mendeteksi variasi penulisan dalam bahasa alay. Awalnya, string input dibagi menjadi array kata-kata berdasarkan spasi, dengan mengabaikan spasi ekstra. Pola regex diinisialisasi dengan "^" untuk menandakan awal string. Untuk setiap kata dalam array, fungsi memasukkan spasi opsional antar kata menggunakan "\s\*". Setiap karakter dalam kata diuji apakah termasuk vokal atau konsonan. Vokal diubah menjadi varian regex yang mencakup bentuk asli, bentuk besar, dan varian alaynya, sementara konsonan diganti dengan bentuk huruf besar dan kecilnya, dan beberapa diubah menjadi bentuk alay. Karakter ini dibuat opsional dalam regex dengan menambahkan "?". Pola ini untuk setiap kata kemudian digabungkan kembali menjadi satu string pola regex, dan ditutup dengan "\$" untuk menandakan akhir string.

2. BoyerMoore

Kelas ini ditujukan untuk semua fungsi yang berkaitan dengan penyelesaian permasalahan dengan menggunakan algoritma Boyer Moore. Adapun fungsi yang dideklarasikan pada kelas ini yaitu:

1. max ( Input : (Integer: a,b) ; Output (Integer))

Fungsi ini mencari nilai maksimum dari a dan b.

2. generateBadCharArray ( Input : (array of char:str, Integer, size); Output(array of integer))

Fungsi ini menghasilkan array bad character heuristic yang digunakan dalam algoritma Boyer-Moore.

3. Search ( Input : (String: pattern, text); Output: boolean)

Fungsi ini menerapkan konsep pencarian Boyer Moore.

3. KnuthMorrisPratt

Kelas ini digunakan untuk menyelesaikan pencarian dengan menggunakan algoritma KMP. Adapun fungsi yang dideklarasikan yaitu:

1. Search ( Input : (String : pattern, text); Output : Boolean)

Fungsi ini digunakan untuk menerapkan pencarian dengan konsep KnuthMorrisPratt

2. generateLPSArray ( Input: (String : pat, Integer: patternLength); Output: array of integer)

Fungsi ini ditujukan untuk menghasilkan *border function*

4. LevenshteinDistance

Kelas ini digunakan untuk menyelesaikan pencarian dengan LD. Adapun fungsi yang dideklarasikan yaitu:

1. Get( Input: (String : text1, text2); Output: Integer)

Fungsi ini berfungsi untuk mendapatkan nilai LD dari 2 *text*.

Folder keempat yaitu folder *Processor*, dimana folder ini berisi *file* yang ditujukan dalam melakukan pemrosesan data. Adapun kelas yang dideklarasikan dalam folder ini yaitu:

1. FingerprintProcessor

Kelas ini bertugas untuk memproses daftar jalur gambar sidik jari menjadi objek sidik jari. Adapun fungsi yang dideklarasikan yaitu:

1. Process ( Input: Array of fingerprintPath ; Output : Array of fingerprint)

Fungsi ini bertugas untuk memproses daftar jalur gambar sidik jari menjadi objek sidik jari.

## 2. ImageProcessor

Kelas menyediakan metode untuk memproses gambar menjadi representasi ASCII dengan berbagai pendekatan, seperti konversi seluruh piksel gambar atau hanya memilih blok terbaik berdasarkan jumlah transisi piksel. Adapun fungsi yang dideklarasikan yaitu:

1. ConvertImageToBinary ( Input: (String:imagePath) ; Output : string)

Fungsi ini mengkonversi gambar menjadi string biner. Fungsi ini membaca setiap piksel gambar, mengkonversinya menjadi skala abu-abu, dan kemudian mengubahnya menjadi '1' (untuk piksel terang) atau '0' (untuk piksel gelap).

2. ConvertBinaryToAscii ( Input: (String:imagePath) ; Output : string)

Fungsi ini mengkonversi string biner menjadi representasi ASCII. Fungsi ini bekerja dengan mengambil setiap 8 karakter dari string biner, mengkonversinya menjadi byte, dan kemudian mengubah byte tersebut menjadi karakter ASCII.

3. ReadAllPixelToAscii ( Input: (String:imagePath) ; Output : string)

Fungsi ini digunakan untuk membaca seluruh piksel gambar dan mengkonversinya menjadi representasi ASCII. Metode ini pertama-tama memanggil ConvertImageToBinary untuk mengkonversi gambar menjadi string biner, lalu menggunakan ConvertBinaryToAscii untuk mengonversi string biner tersebut menjadi string ASCII.

4. CountTransition ( Input: (String:binaryString) ; Output : Integer)

Fungsi ini menghitung jumlah transisi antara '0' dan '1' dalam string biner. Transisi terjadi ketika karakter berubah dari '0' ke '1' atau dari '1' ke '0'.

5. FindBestTransitionBlock ( Input: (String:binaryString, Integer: blockSize) ; Output : String)

Fungsi ini mencari blok string biner dengan jumlah transisi terbanyak. Metode ini memindai string biner dalam blok-blok dengan ukuran tertentu dan mengembalikan blok dengan jumlah transisi terbanyak.

6. ReadBestPixelToAscii ( Input: (String:imagePath) ; Output : string)

Fungsi ini digunakan untuk membaca blok terbaik dari gambar berdasarkan jumlah transisi dan mengkonversinya menjadi representasi ASCII. Metode ini

pertama-tama memanggil ConvertImageToBinary untuk mengkonversi gambar menjadi string biner, kemudian menggunakan FindBestTransitionBlock untuk menemukan blok dengan transisi terbanyak, dan akhirnya menggunakan ConvertBinaryToAscii untuk mengonversi blok tersebut menjadi string ASCII.

Terdapat pula file MainWindow.xaml.cs yang berfungsi untuk memberikan UX yang sesuai dengan tampilan yang sudah dibuat pada file MainWindow.xaml. Kelas MainWindow memiliki beberapa deklarasi yaitu:

1. Inisialisasi konstruktor yang melakukan perhubungan dengan Database dan inisialisasi komponen yang dibutuhkan.
2. Prosedur fileButton\_Click yang berfungsi untuk menerima inputan file yang sudah dibuat untuk selalu berupa file .BMP.
3. Prosedur searchButton\_Click yang berfungsi untuk memanggil fungsi pencarian serta menampilkan hasil pencarian sesuai bagian masing-masing pada layar.
4. Prosedur toggleKMP\_Checked

Berfungsi dalam pengubahan tampilan UI serta memaksimalkan UX pengguna dalam pemilihan algoritma KMP.

5. Prosedur toggleBM\_Checked

Berfungsi dalam pengubahan tampilan UI serta memaksimalkan UX pengguna dalam pemilihan algoritma BM.

6. Prosedur ClearResults

Membersihkan nilai dari berbagai komponen pada aplikasi.

7. Deklarasi struktur data ResultItem

Deklarasi dari *frontend*, yaitu berupa *label* dan *value*.

Terakhir, terdapat file Searcher.cs yang berfungsi layaknya *backend* pada aplikasi ini. Adapun fungsi yang dideklarasikan yaitu:

1. SearchByLD ( Input: (String: fullPixelAscii , array of Fingerprint:fingerprints) ; Output : string, int, string)

Fungsi ini memanggil pencarian dengan menggunakan metode LD.

2. GetBiodataByRegexAlayOrLD ( Input: (String:nama) ; Output : Biodata)

Mengecek apakah nama yang didapat ada pada *database*, dan jika tidak ada akan digunakan algoritma LD untuk mencari biodata dengan batasan *threshold* tertentu.

3. SearchByKMP ( Input: (String: pattern , array of Fingerprint:fingerprints) ; Output : string, int, string)

Fungsi ini memanggil pencarian dengan menggunakan metode KMP,

4. SearchByBM ( Input: (String: pattern , array of Fingerprint:fingerprints) ; Output : string, int, string)

Fungsi ini memanggil pencarian dengan menggunakan metode BM.

5. GetResult ( Input: (String: imagePath, algo) ; Output : SearchResult)

Fungsi ini bekerja layaknya *backend* dan memanggil fungsi pencarian sesuai inputan algoritma dan mengembalikan hasil dalam bentuk entitas SearchResult. Untuk tiap metode pencarian diatas akan mengembalikan nama hasil, kemiripan, dan tautan gambar. Dalam fungsi ini akan dihitung juga *exec time*.

## 4.2 Penjelasan tata cara penggunaan program

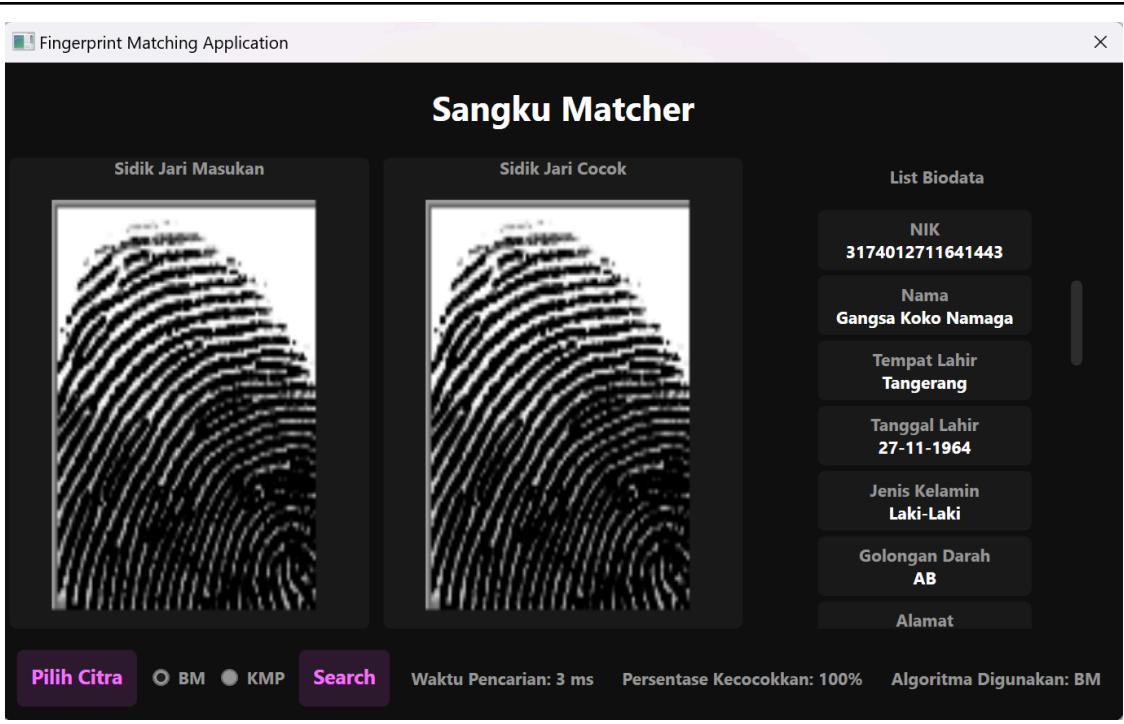
Program dimulai dengan mengarahkan *directory* pada folder src. Pemanggilan program utama dilakukan dengan memanggil *command dotnet run*. Dalam tampilan aplikasi, terdapat 3 *container* beserta dengan berbagai tombol serta opsi pemilihan algoritma. Tahapan pertama yaitu menggunakan tombol penambahan foto untuk menginisialisasi pencarian. Tampilan foto akan ditampilkan pada *container* pertama. Selanjutnya, dilakukan pemilihan algoritma yang ingin digunakan dalam mencari informasi dengan menekan tombol pada *radio button* yang sesuai. Algoritma yang disediakan yaitu algoritma BM (Boyer-Moore) dan KMP (Knuth-Morris-Pratt).

Setelah melewati tahapan inisialisasi pencarian, dapat dilakukan pencarian dengan menekan tombol *search*. Tunggu beberapa lama sampai tampilan *loading* hilang dan dalam *container* kedua akan ditampilkan foto yang sesuai dari hasil pencarian. *Container* ketiga akan menampilkan informasi dari target pemilik sidik jari yang dihitung dan dianggap sesuai dengan input pengguna.

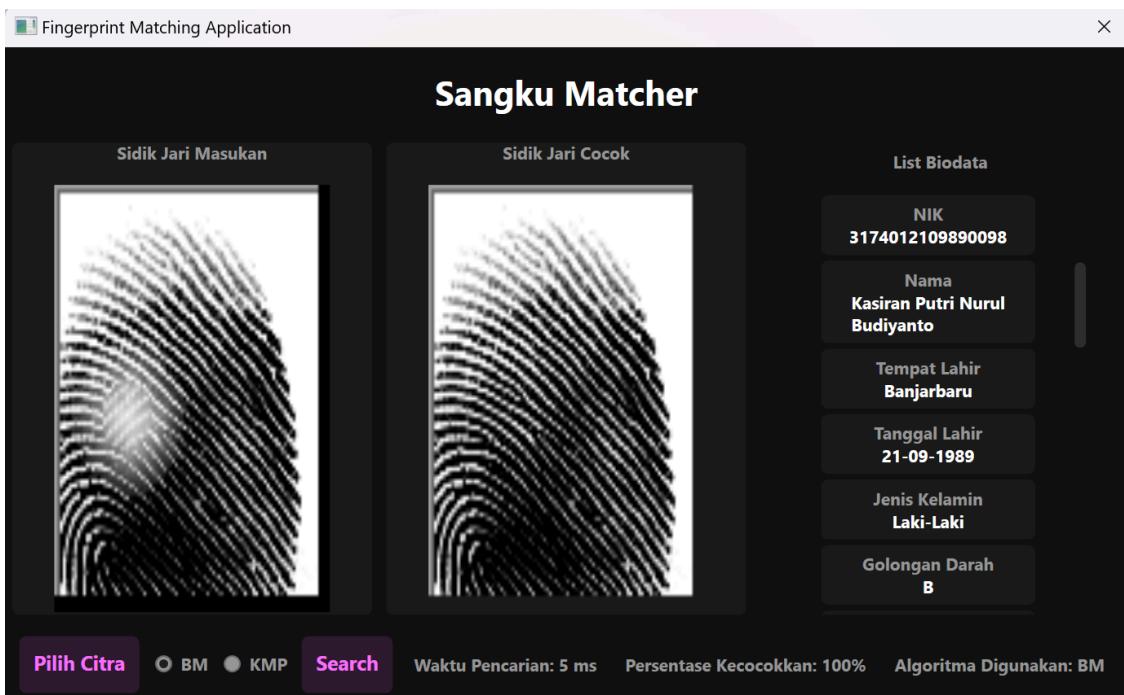
## 4.3 Hasil pengujian

### 4.3.1 Pengujian Algoritma BM

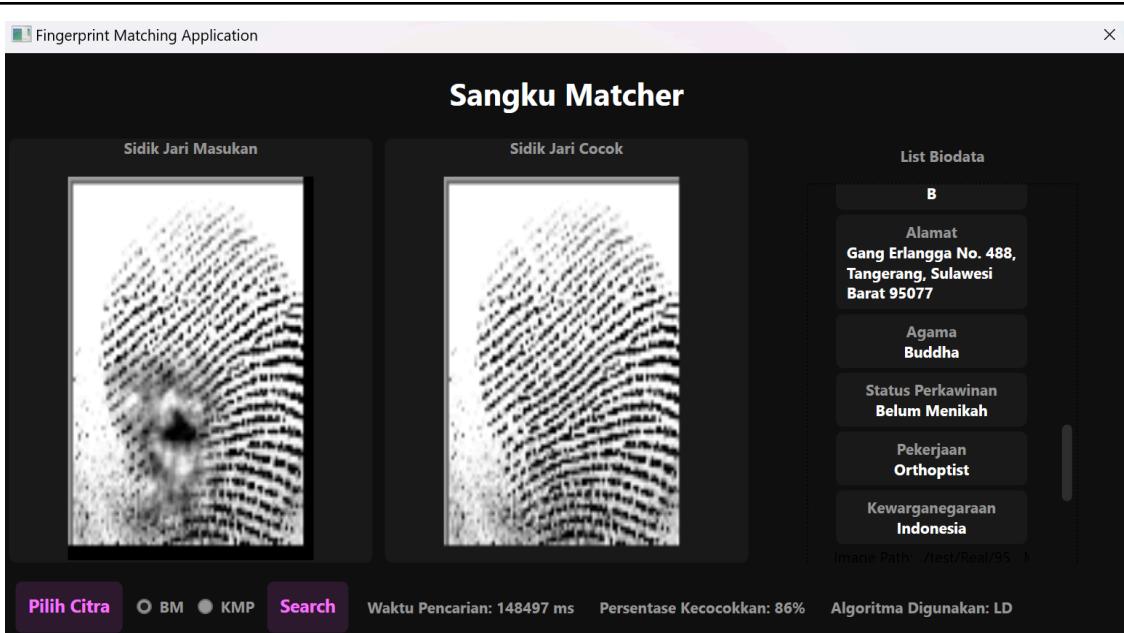
No.	Hasil Pengujian
1	real



2 altered easy



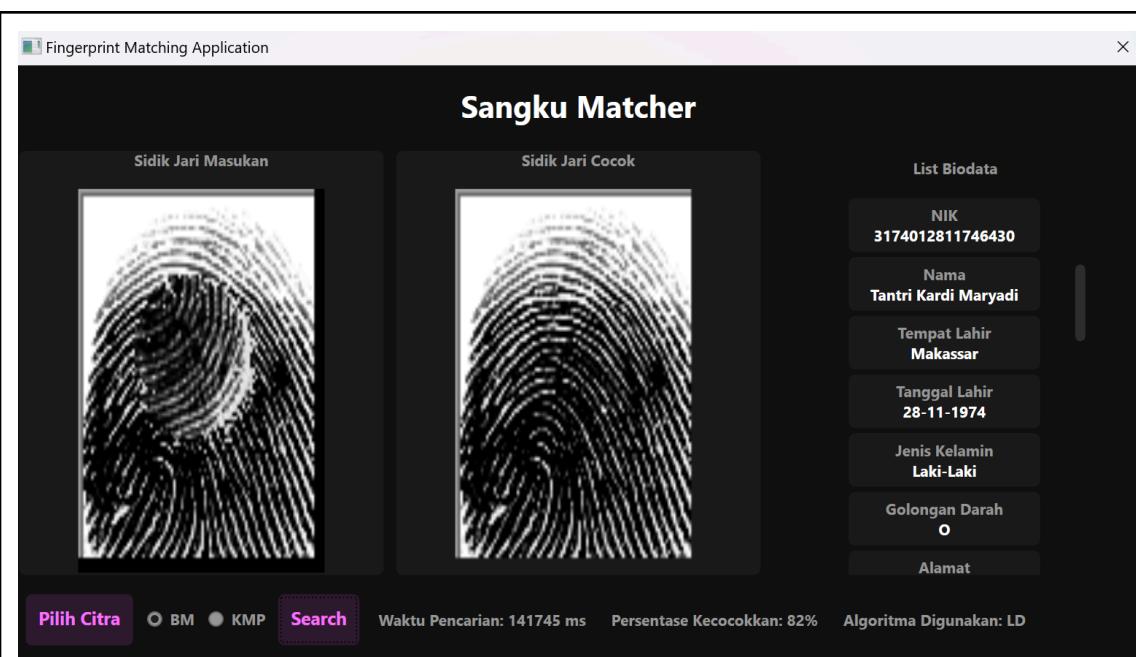
3 altered hard



4 altered medium

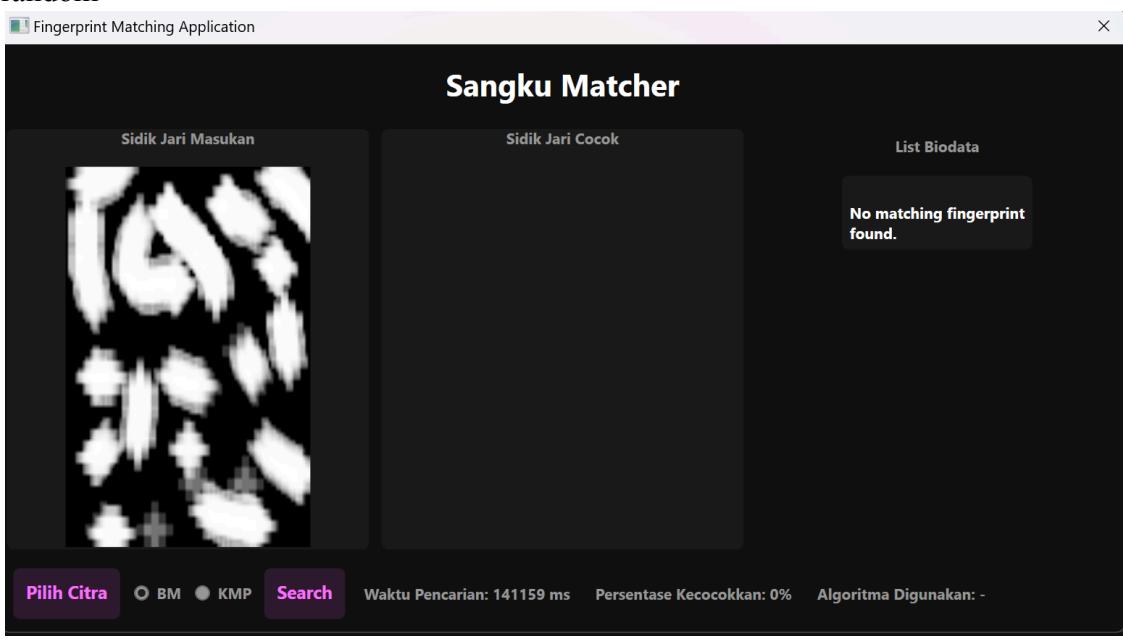


5 altered medium



6

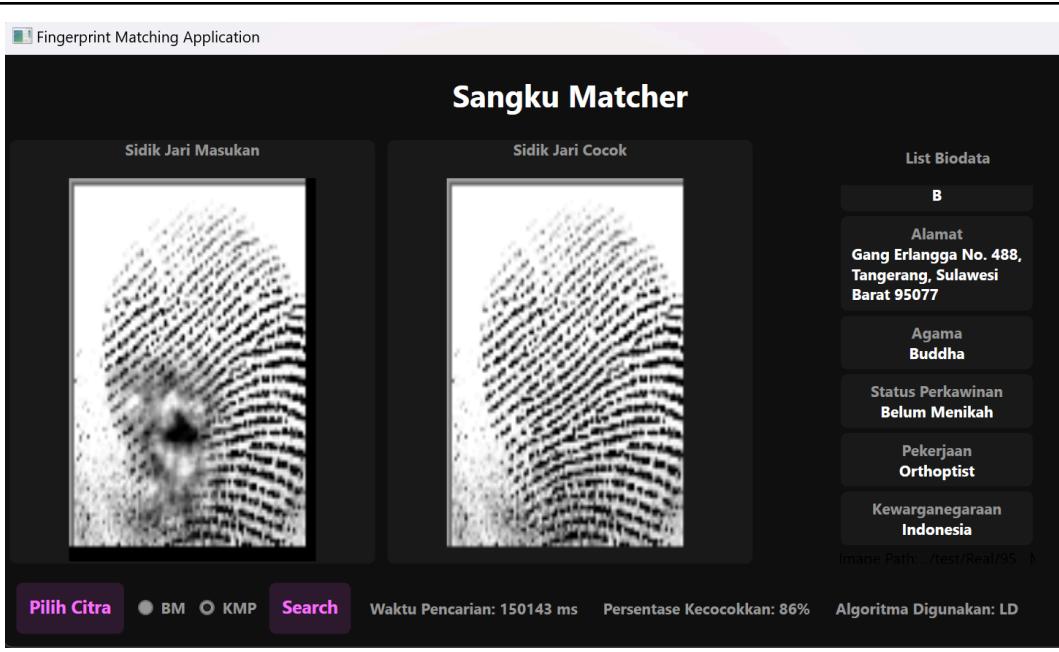
random



### 4.3.2 Pengujian Algoritma KMP

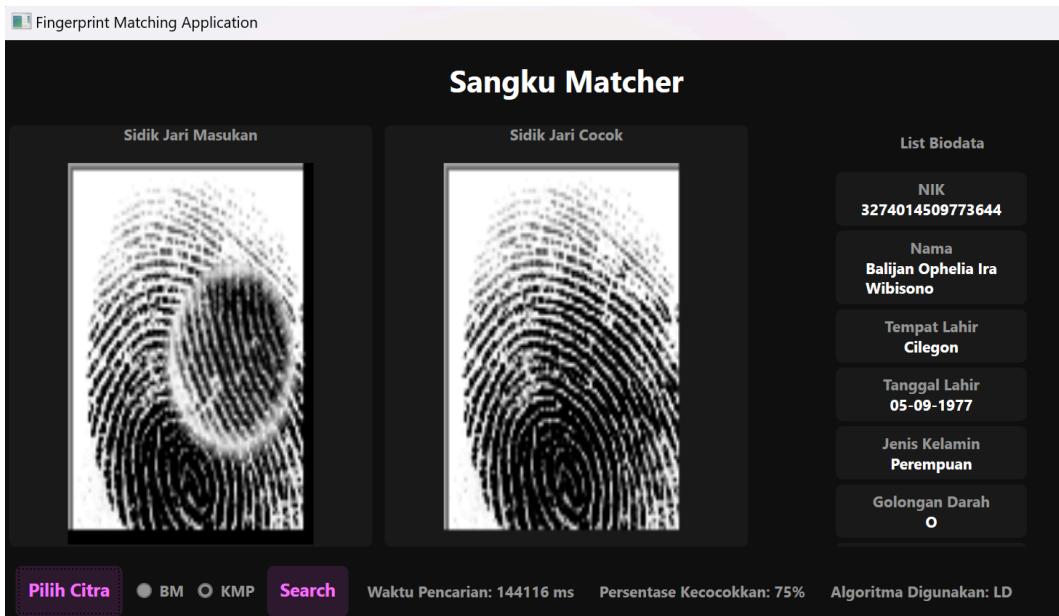
No.	Hasil Pengujian

1	real
	<p>Fingerprint Matching Application</p> <h3>Sangku Matcher</h3> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <p><b>Sidik Jari Masukan</b></p> </div> <div style="text-align: center;"> <p><b>Sidik Jari Cocok</b></p> </div> <div style="text-align: center;"> <p><b>List Biodata</b></p> <ul style="list-style-type: none"> <li>NIK <b>3174012711641443</b></li> <li>Nama <b>Gangsa Koko Namaga</b></li> <li>Tempat Lahir <b>Tangerang</b></li> <li>Tanggal Lahir <b>27-11-1964</b></li> <li>Jenis Kelamin <b>Laki-Laki</b></li> <li>Golongan Darah <b>AB</b></li> <li>Alamat</li> </ul> </div> </div> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <span><b>Pilih Citra</b></span> <span><input checked="" type="radio"/> BM   <input type="radio"/> KMP</span> <span><b>Search</b></span> <span>Waktu Pencarian: 9 ms</span> <span>Persentase Kecocokkan: 100%</span> <span>Algoritma Digunakan</span> </div>
2	altered easy
	<p>Fingerprint Matching Application</p> <h3>Sangku Matcher</h3> <div style="display: flex; justify-content: space-around;"> <div style="text-align: center;"> <p><b>Sidik Jari Masukan</b></p> </div> <div style="text-align: center;"> <p><b>Sidik Jari Cocok</b></p> </div> <div style="text-align: center;"> <p><b>List Biodata</b></p> <ul style="list-style-type: none"> <li>NIK <b>3174012109890098</b></li> <li>Nama <b>Kasiran Putri Nurul Budiyanto</b></li> <li>Tempat Lahir <b>Banjarbaru</b></li> <li>Tanggal Lahir <b>21-09-1989</b></li> <li>Jenis Kelamin <b>Laki-Laki</b></li> <li>Golongan Darah <b>B</b></li> </ul> </div> </div> <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <span><b>Pilih Citra</b></span> <span><input checked="" type="radio"/> BM   <input type="radio"/> KMP</span> <span><b>Search</b></span> <span>Waktu Pencarian: 18 ms</span> <span>Persentase Kecocokkan: 100%</span> <span>Algoritma Digunakan</span> </div>
3	altered hard



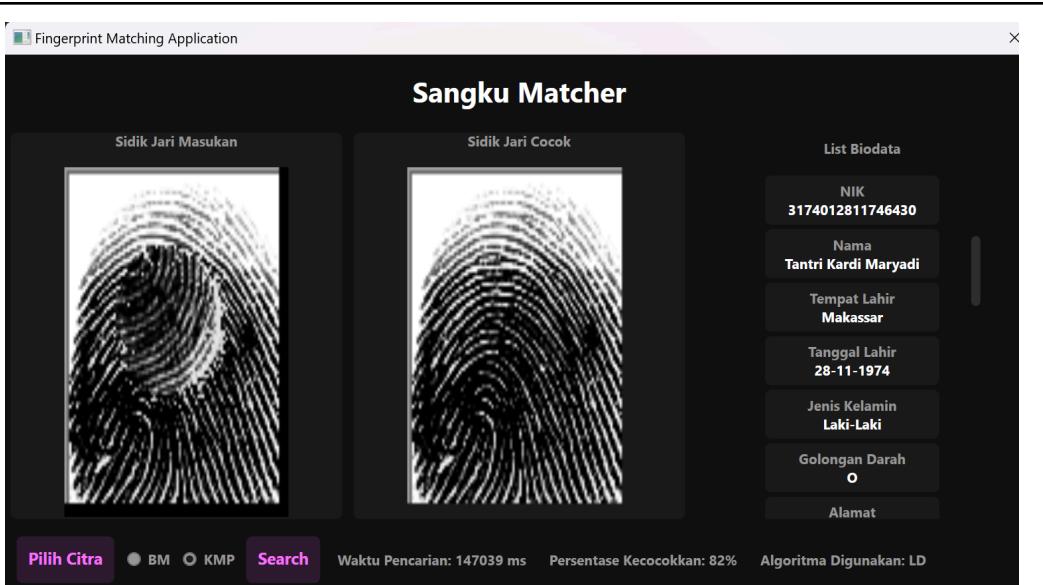
4

altered medium

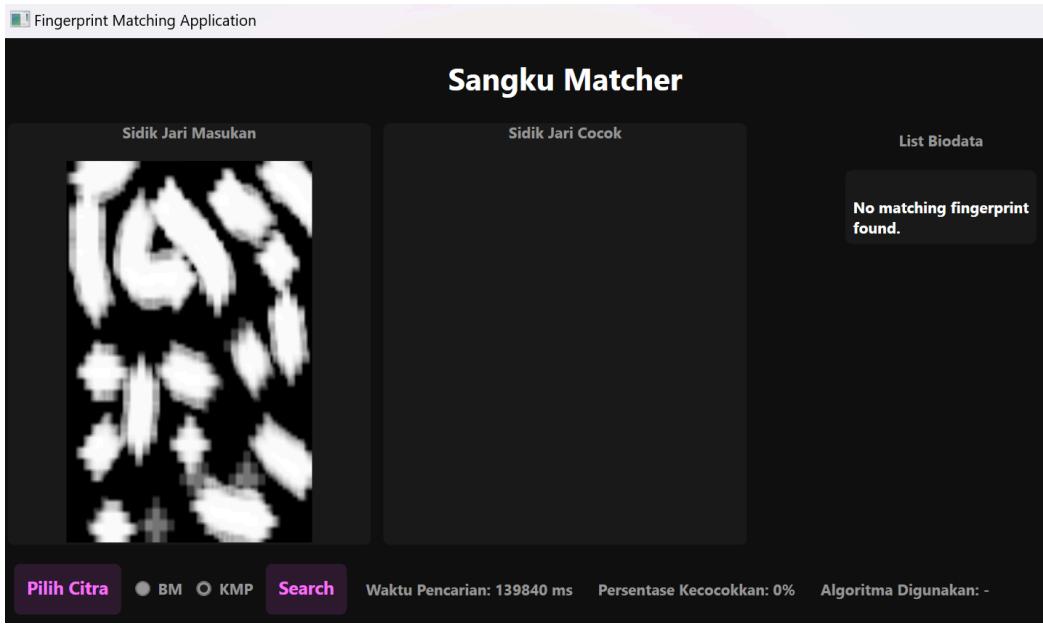


5

altered medium



6 random



#### 4.4 Analisis hasil pengujian

Algoritma KMP merupakan salah satu teknik pencocokan string yang efisien dan dirancang untuk mengurangi jumlah perbandingan yang tidak perlu selama proses pencocokan dengan memanfaatkan informasi yang telah diperoleh dari upaya pencocokan sebelumnya.

Algoritma ini mengembangkan sebuah tabel pra-komputasi yang dikenal sebagai border function yang memungkinkan pencarian untuk melompati segmen teks yang pasti tidak cocok, tanpa perlu memeriksa ulang karakter-karakter tersebut. Tabel ini direkayasa berdasarkan pola itu sendiri, mencatat indeks di mana pencarian dapat dilanjutkan setelah ketidakcocokan terjadi. Hal ini sangat mengurangi waktu yang diperlukan dalam pencocokan, terutama dalam kasus-kasus di mana teks mengandung banyak pengulangan.

Untuk kompleksitas algoritma dari KMP, untuk kasus terbaiknya terjadi dimana ketika karakter pertama dari pola tidak cocok dengan karakter di teks, atau setiap karakter di teks cocok sempurna dengan pola. Dalam skenario ini karena karakter pertama dari pola tidak cocok, algoritma segera melompat ke karakter berikutnya di teks, tanpa mengulangi pengecekan untuk karakter-karakter yang sudah dicocokkan. Kompleksitas waktu untuk kasus terbaik adalah  $O(n)$ , dengan  $n$  adalah panjang teks. Ini menunjukkan bahwa setiap karakter di teks hanya perlu diperiksa satu kali. Sedangkan untuk kasus terburuknya terjadi ketika pola ditemukan hampir di akhir teks atau setiap kegagalan cocok terjadi di akhir pola. Contoh dari kasus terburuk adalah ketika pola dan teks terdiri dari karakter yang sama kecuali karakter terakhir. Dalam kasus ini, KMP akan mencocokkan seluruh pola kecuali karakter terakhir berulang kali sampai mencapai akhir teks. Namun, berkat penggunaan tabel pra-komputasi, algoritma tidak perlu memulai pencocokan dari awal pola setelah setiap kegagalan. Kompleksitas waktu untuk kasus terburuk KMP adalah  $O(n + m)$ , dengan  $m$  adalah panjang pola. Meskipun kompleksitasnya terlihat hamper sama dengan kompleksitas dari kasus terbaik, kebutuhan untuk kembali beberapa langkah untuk mengecek kecocokan dapat membuatnya sedikit lebih lambat dibandingkan kasus terbaik.

Berbeda dengan KMP, algoritma Boyer-Moore (BM) memaksimalkan jumlah karakter yang dapat di-skip selama pencocokan dengan memanfaatkan informasi heuristiknya yang membuatnya algoritma BM secara umum lebih cepat dari KMP khususnya pada teks yang panjang atau pola yang panjang. Algoritma BM menggunakan strategi heuristic utama yaitu tabel "bad character". Tabel "bad character" menyediakan informasi tentang seberapa jauh pola harus digeser ketika terjadi ketidakcocokan karakter. Jika karakter tidak ada di dalam pola, maka pola bisa langsung digeser sejauh panjangnya. Ini mengurangi jumlah perbandingan secara signifikan karena karakter yang tidak ada dalam pola menunjukkan tidak adanya kesempatan cocok dalam geseran itu. Tabel heuristic ini memungkinkan BM untuk melewati sejumlah besar teks yang

tidak mungkin cocok, yang sering menghasilkan peningkatan dramatis dalam kecepatan pencarian dibandingkan dengan KMP. Dalam kasus terbaiknya, kompleksitas waktu algoritma dari BM adalah  $O(n/m)$  dimana n adalah panjang teks dan m adalah panjang polanya. Ini terjadi ketika pola tidak ditemukan dalam teks dan heuristik bad character memungkinkan algoritma untuk melompati sebagian besar teks dan sering kali melompati lebih dari panjang pola itu sendiri yang membuat algoritma BM ini efisien dan lebih cepat dari algoritma KMP.

Dalam aplikasi pencocokan sidik jari, perbedaan kinerja antara KMP dan BM tidak terlalu terdapat perbedaan yang terlalu besar. KMP memberikan kinerja yang lebih konsisten dan dapat diandalkan dalam kasus teks atau pola dengan struktur berulang, karena kecenderungannya untuk menghindari pemeriksaan ulang karakter yang sudah di-match. Sebaliknya, BM cenderung lebih unggul dalam skenario di mana teks yang dianalisis cukup besar dan pola yang dicari juga panjang. Algoritma ini sangat efektif dalam mengurangi waktu pencarian dengan meminimalkan komparasi berkat strategi shiftingnya yang efisien. Oleh karena itu, dalam praktiknya, BM sering menjadi pilihan yang lebih baik untuk pencocokan sidik jari dalam database besar, di mana efisiensi waktu adalah prioritas. Akan tetapi, kedua algoritma ini hanya bisa mengecek dan memberikan hasil jika tingkat kecocokannya benar-benar 100%, jika tidak maka kedua algoritma ini tidak akan memberikan hasil sehingga digunakanlah algoritma Levenshtein Distance untuk mengatasi masalah tersebut.

Ketika pencocokan pola menggunakan KMP dan BM tidak menghasilkan kecocokan yang sempurna (100%), Levenshtein Distance menjadi sangat berguna untuk mengukur dan menganalisis tingkat kemiripan antara sidik jari yang dicocokkan. Algoritma ini menghitung jumlah minimum operasi (penyisipan, penghapusan, atau substitusi) yang diperlukan untuk mengubah satu string menjadi string lain sehingga sangat berguna untuk mengukur tingkat kemiripan antara dua sidik jari, terutama dalam kasus di mana sidik jari mungkin rusak atau gambar tidak lengkap. Dalam kasus di mana sidik jari rusak atau kualitas gambar tidak optimal, kepercayaan terhadap hasil yang diberikan oleh KMP dan BM bisa berkurang, membuat LD sangat penting.

Dalam program ini, setiap gambar sidik jari diubah menjadi representasi biner berdasarkan ambang batas grayscale. Gambar yang rusak atau kualitas gambar yang buruk dapat menghasilkan transisi yang lebih acak dan sering (dari 0 ke 1 atau sebaliknya) dalam representasi biner tersebut. Ketika sidik jari yang rusak dianalisis, blok 32-pixel yang digunakan mungkin

tidak akurat mencerminkan pola sidik jari asli karena kehilangan informasi atau distorsi. Hal ini menghasilkan representasi biner yang signifikan berbeda dari versi asli atau dari sidik jari yang ada dalam database, menyebabkan KMP dan BM gagal menemukan kecocokan yang tepat. Sebaliknya, LD mampu mengukur tingkat perbedaan yang diperlukan untuk menyamai kedua sidik jari, memberikan kegunaan yang krusial dalam kondisi tersebut.

Levenshtein Distance dapat menilai tingkat kemiripan berdasarkan perubahan minimal yang diperlukan, memberikan hasil dalam bentuk persentase kesamaan yang lebih detail dan granular. Namun, LD memerlukan waktu eksekusi yang lebih lama dibandingkan dengan KMP dan BM. Proses perhitungan LD melibatkan pembuatan matriks ukuran  $(m+1) \times (n+1)$ , di mana  $m$  dan  $n$  adalah panjang dari dua string yang dibandingkan. Setiap sel di matriks ini diisi dengan biaya minimum untuk mengubah substring dari string pertama menjadi substring dari string kedua sehingga proses ini membutuhkan waktu yang signifikan terutama untuk string yang panjang karena memerlukan penghitungan yang kompleks untuk setiap perubahan karakter.

## **BAB V**

### **KESIMPULAN DAN SARAN**

#### **5.1 Kesimpulan**

Dalam tugas besar ini, kami telah berhasil mengimplementasikan Sistem Deteksi Individu berbasis Biometrik melalui citra sidik jari dengan menggunakan beberapa algoritma pattern matching, yaitu Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM), serta Levenshtein Distance (LD) untuk mengatasi kasus-kasus di mana kecocokan tidak sempurna ditemukan.

Algoritma KMP dan BM terbukti efisien dalam pencocokan pola sidik jari. KMP bekerja dengan baik dalam teks atau pola yang memiliki struktur berulang, menghindari pemeriksaan ulang karakter yang sudah dicocokkan, sehingga memberikan kinerja yang konsisten dan dapat diandalkan. BM, di sisi lain cenderung lebih unggul dalam skenario di mana teks yang dianalisis cukup besar dan pola yang dicari juga panjang, berkat strategi heuristiknya yang memungkinkan pengurangan jumlah perbandingan secara signifikan yang membuat algoritma BM lebih cepat dari KMP.

LD juga penting ketika KMP dan BM tidak menghasilkan kecocokan yang sempurna. Algoritma ini menghitung jumlah minimum operasi yang diperlukan untuk mengubah satu string menjadi string lain, yang sangat berguna untuk mengukur tingkat kemiripan antara dua sidik jari. Dalam kasus sidik jari yang rusak atau gambar yang tidak lengkap, LD mampu mengukur tingkat perbedaan yang diperlukan untuk menyamai kedua sidik jari, memberikan kegunaan yang krusial dalam kondisi tersebut.

Dengan menggabungkan teknologi identifikasi sidik jari dan pattern matching, kami dapat membangun sistem identifikasi biometrik yang aman, andal, dan mudah digunakan. Sistem ini dapat diaplikasikan di berbagai bidang seperti kontrol akses, absensi karyawan, dan verifikasi identitas dalam transaksi keuangan. Penerapan sistem ini diharapkan dapat memberikan solusi keamanan yang efektif dan efisien dalam berbagai aspek kehidupan sehari-hari.

Secara keseluruhan, proyek ini membuktikan bahwa algoritma *pattern matching* dapat secara efektif digunakan untuk mengembangkan sistem deteksi individu berbasis biometrik. Dengan potensi untuk pengembangan lebih lanjut, seperti implementasi pemrosesan paralel dan pengintegrasian algoritma enkripsi yang lebih kuat, sistem ini bisa lebih ditingkatkan untuk memenuhi tantangan keamanan masa depan. Selain itu, penelitian lebih lanjut dalam optimasi

algoritma dan teknologi sensor sidik jari dapat membawa peningkatan signifikan dalam efisiensi dan akurasi sistem.

## 5.2 Saran

Secara umum, algoritma Boyer-Moore (BM) dan Knuth-Morris-Pratt (KMP) sudah cukup cepat untuk menangani kasus uji pencocokan pola. Oleh karena itu, tidak diperlukan perbaikan tambahan pada implementasi mereka, seperti penambahan pemrosesan paralel dan sebagainya. Namun, untuk lebih mempercepat proses pencarian, dapat dipertimbangkan penambahan pemrosesan paralel pada saat menggunakan algoritma Levenshtein Distance (LD). Hal ini disebabkan oleh waktu pemrosesan yang sangat lama ketika menggunakan LD untuk kuantitas data yang besar. Dengan pemrosesan paralel, waktu yang diperlukan untuk menghitung tingkat kemiripan antara dua sidik jari dapat dikurangi secara signifikan.

Selain itu, untuk mempercepat waktu akses data, pemrosesan paralel juga dapat diterapkan pada proses dekripsi data dari basis data. Proses dekripsi ini memakan waktu cukup lama, terutama ketika menangani sejumlah besar data. Dengan menggunakan pemrosesan paralel, dekripsi data dapat dilakukan secara bersamaan, sehingga mengurangi waktu keseluruhan yang dibutuhkan untuk mengakses dan memproses data.

Untuk meningkatkan keamanan, sebaiknya algoritma enkripsi yang lebih kuat seperti Advanced Encryption Standard (AES) digunakan. AES menawarkan tingkat keamanan yang lebih tinggi dibandingkan dengan algoritma enkripsi sederhana dan telah menjadi standar enkripsi yang diakui secara luas. Implementasi AES akan memastikan bahwa data sensitif tetap terlindungi dari akses yang tidak sah.

Dengan melakukan penambahan pemrosesan paralel pada algoritma Levenshtein Distance dan proses dekripsi data, serta menggunakan algoritma enkripsi yang lebih kuat seperti AES, efisiensi dan keamanan sistem pencocokan sidik jari dapat ditingkatkan secara signifikan.

## 5.3 Tanggapan dan refleksi

Kami merasa bahwa tugas besar ini cukup mudah dipahami dan memberikan kami pemahaman yang lebih mendalam mengenai penerapan algoritma KMP (Knuth-Morris-Pratt) dan BM (Boyer-Moore) dalam konteks pencocokan pola. Proyek ini juga telah memperluas

wawasan kami tentang penggunaan bahasa pemrograman C#, serta ekosistem .NET dan berbagai fiturnya.

Melalui tugas ini, kami dapat mengeksplorasi lebih dalam tentang bagaimana algoritma KMP dan BM dapat diterapkan secara efektif untuk memecahkan masalah nyata, khususnya dalam identifikasi biometrik berbasis citra sidik jari. Penggunaan algoritma-algoritma ini memungkinkan kami untuk memahami cara kerja mereka secara lebih komprehensif, termasuk kekuatan dan kelemahan masing-masing dalam berbagai skenario.

Selain itu, pengalaman menggunakan C# dan .NET memberikan kami wawasan baru tentang kekuatan dan fleksibilitas platform ini dalam mengembangkan aplikasi desktop yang kompleks. Kami belajar tentang berbagai aspek pemrograman dalam C#, mulai dari manajemen data hingga implementasi algoritma dan integrasi dengan database. Tugas ini juga memberikan kami kesempatan untuk mempraktikkan teknik-teknik pemrograman yang efisien dan aman, seperti penggunaan enkripsi data dan pemrosesan paralel.

Secara keseluruhan, tugas besar ini tidak hanya memperkuat pemahaman kami tentang algoritma pencocokan pola, tetapi juga memperkaya pengetahuan kami tentang pengembangan perangkat lunak menggunakan teknologi modern. Kami merasa bahwa pengalaman ini sangat berharga dan akan bermanfaat bagi kami dalam proyek-proyek masa depan.

## **LAMPIRAN**

### **Pranala Repository**

[https://github.com/Filbert88/Tubes3\\_Sangku](https://github.com/Filbert88/Tubes3_Sangku)

### **Pranala Video**

<https://youtu.be/LwMHSXUywe8?feature=shared>

## DAFTAR PUSTAKA

Munir, Rinaldi. n.d. “Pencocokan string (String matching/pattern matching)” Informatika.

Diakses 24 April 2024.

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf>

Munir, Rinaldi. n.d. “Pencocokan string dengan Regular Expression (Regex)” Informatika.

Diakses 24 April 2024.

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

GeeksforGeeks. n.d. “Introduction to Levenshtein Distance.” Diakses 24 April 2024.

<https://www.geeksforgeeks.org/introduction-to-levenshtein-distance/>