LAPORAN TUGAS BESAR 02
IF2211 STRATEGI ALGORITMA

PEMANFAATAN PATTERN MATCHING DALAM MEMBANGUN SISTEM DETEKSI INDIVIDU BERBASIS BIOMETRIK MELALUI CITRA SIDIK JARI

Kelompok 48 : apaKek

Disusun oleh:

Benardo	13522055
Albert	13522081
Mesach Harmasendro	13522117

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA (STEI)

INSTITUT TEKNOLOGI BANDUNG

2024

DAFTAR ISI

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, Anda 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.

Penjelasan Implementasi

Pada tugas ini, Anda diminta untuk mengimplementasikan sebuah program yang dapat melakukan identifikasi biometrik berbasis sidik jari. Proses implementasi dilakukan dengan menggunakan algoritma Boyer-Moore dan Knuth-Morris-Pratt, sesuai dengan yang diajarkan pada materi dan salindia kuliah.

Secara sekilas, penggunaan algoritma *pattern matching* dalam mencocokkan sidik jari terdiri atas tiga tahapan utama dengan skema sebagai berikut.

Gambar 2. Skema implementasi konversi citra sidik jari.
Sumber: Dokumentasi Penulis

Gambar yang digunakan pada proses *pattern matching* kedua algoritma tersebut adalah gambar sidik jari penuh berukuran *m* x *n* pixel yang diambil sebesar 30 pixel setiap kali proses pencocokan data. Untuk tugas ini, Anda **dibebaskan** untuk mengambil jumlah pixel asalkan **didasarkan pada alasan yang masuk akal** (dijelaskan pada laporan) dan **penanganan kasus ujung yang baik** (misal jika ternyata ukuran citra sidik jari tidak habis dibagi dengan ukuran pixel yang dipilih). Selanjutnya, data pixel tersebut akan dikonversi menjadi binary. Seperti yang mungkin Anda ketahui sesuai materi kuliah *(ya kalau masuk kelas)*, karena binary hanya memiliki variasi karakter satu atau nol, maka proses *pattern matching* akan membuat proses pencocokan karakter menjadi lambat karena harus sering mengulangi proses pencocokan *pattern*. Cara yang dapat dilakukan untuk mengatasi hal tersebut adalah dengan mengelompokkan setiap baris kode biner per 8 bit sehingga membentuk karakter ASCII. Karakter ASCII 8-bit ini yang akan mewakili proses pencocokan dengan string data.

Untuk memberikan gambaran yang lebih jelas, berikut adalah contoh kasus citra sidik jari dan proses serta sampel hasil yang didapatkan. Dataset yang digunakan adalah dataset citra sidik jari yang terdapat pada bagian referensi.

Citra Sidik Jari	**Potongan nilai binary**	**Potongan ASCII 8-bits**

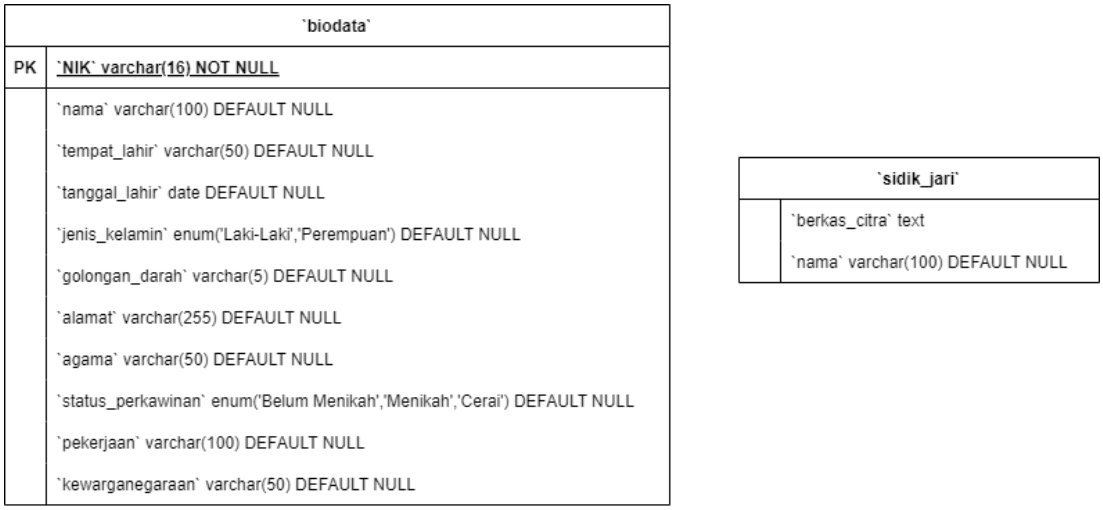
	101000000101000000101000001 010000001010000001010000010 100000010100000010100000101 000000101000000101000001010 000001010000001010000010100 000101000000101000000101000 001010000001010000001010000 010100000010100000010100000 1010000001010000001010	iÿÿÿÿÿÿÿÿÿÿÌÿÿû:Ãÿÿ<Ìÿÿu wûÿkÿý,Pþÿ¿/ÿÿ' »ÿÿÐ¾?ÿ&¼¡¤Úª†È/Zíÿÿq ¼ÿúTý~Rûût`íÿÿÿÿÿÿ iÿÿÿÿÿÿÿÿÿöÿÿÿ"býÿýÁüþÿÊ 'þ/ÿœ2ÿúYÿÿ\ ^ÿÿp fÿÿà !›ÿÿÿþÿÿÿÿÿýš <ðÿÕ ²ÿÿQûÿ Nºÿášÿÿÿÿÿÿ iÿÿÿÿÿÿÿÿÿÿõrÛõÿÿÇ½ÿÿ‡å ÿ‹ >ÿÙþþ6Gÿÿ;Úÿÿÿfäÿý

Tabel 1. Hasil konversi citra sidik jari menjadi binary dan konversi ke ASCII 8-bits.
Sumber: Dokumentasi Penulis

Pada tahap implementasi di titik ini, telah dihasilkan serangkaian karakter ASCII 8-bit yang merepresentasikan sebuah sidik jari. Hasil ini yang akan dijadikan dasar pencarian sidik jari yang sama dengan daftar sidik jari yang terdapat pada basis data. Pencarian sidik jari yang paling mirip dengan sidik jari yang menjadi masukan pengguna dilakukan dengan algoritma pencocokan *string* Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM). Jika tidak ada satupun sidik jari pada basis data yang *exact match* dengan sidik jari yang menjadi masukan melalui algoritma KMP ataupun BM, maka gunakan sidik jari paling mirip dengan kesamaan diatas nilai tertentu (*threshold*). Anda diberikan **kebebasan untuk menentukan nilai batas persentase** kemiripan ini, silakan melakukan pengujian untuk menentukan nilai *tuning* yang tepat dan **jelaskan pada laporan**. Metode perhitungan tingkat kemiripan **juga dibebaskan** kepada Anda asalkan dijelaskan di laporan. Akan tetapi, asisten sangat menyarankan untuk menggunakan salah satu dari algoritma **Hamming Distance**, **Levenshtein Distance**, ataupun **Longest Common Subsequence (LCS)**.

Fungsi dari deteksi biometrik adalah mengidentifikasi seseorang, oleh sebab itu, pada proses implementasi program ini, sebuah citra sidik jari akan dicocokkan dengan biodata seseorang. Biodata yang dicocokkan terdiri atas data-data yang terdapat pada KTP, antara lain :

NIK, nama, tempat/tanggal lahir, jenis kelamin, golongan darah, alamat, agama, status perkawinan, pekerjaan, dan kewarganegaraan. Relasi ini dibuat dalam sebuah basis data dengan skema detail seperti yang tertera pada bagian di bawah ini. Sebagai tambahan, struktur relasi basis data telah disediakan, silakan gunakan dump sql berikut. Atribut berkas_citra yang disimpan pada tabel sidik_jari adalah alamat dari citra dalam repositori (test/…), lokasi penyimpanan citra dalam folder **test**, bisa dilihat pada struktur repositori di bagian Pengumpulan Tugas.



Gambar 3. Skema basis data yang digunakan.

Sumber: Dokumentasi Penulis

Seorang pribadi dapat memiliki lebih dari satu berkas citra sidik jari (**relasi *one-to-many***). Akan tetapi, seperti yang dapat dilihat pada skema relasional di atas, keduanya tidak terhubung dengan sebuah relasi. Hal ini disebabkan karena pada kasus dunia nyata, data yang disimpan bisa saja mengalami korup. Dengan membuat atribut kolom yang mungkin korup adalah atribut nama pada tabel biodata, maka atribut nama pada tabel sidik_jari **tidak dapat memiliki *foreign-key*** yang mereferensi ke tabel biodata (silakan *review* kembali materi IF2240 bagian basis data relasional). Pada tugas besar kali ini, kita akan coba melakukan simulasi implementasi data korup yang **hanya mungkin terjadi pada atribut nama di tabel biodata** (asumsikan kolom lain pada setiap tabel tidak mengalami korup). Akan tetapi, karena tujuan utama program adalah mengenali identitas seseorang secara lengkap hanya dengan menggunakan sidik jari, maka harus dilakukan sebuah skema untuk menangani data korup tersebut.

Sebuah data yang korup dapat memiliki berbagai macam bentuk. Pada tugas ini, jenis data korup adalah bahasa alay Indonesia. Terdengar lucu, tetapi para pendahulu kita sudah membuat bahasa ini untuk berkomunikasi kepada sesamanya. Dengan mengutip dari berbagai sumber, Anda akan diminta untuk menangani **kombinasi dari tiga buah variasi** bahasa alay, yaitu kombinasi huruf besar-kecil, penggunaan angka, dan penyingkatan. Contoh kasus bahasa alay dijelaskan dengan detail sebagai berikut.

Variasi	**Hasil**
Kata orisinil	Bintang Dwi Marthen
Kombinasi huruf besar-kecil	bintanG DwI mArthen
Penggunaan angka	B1nt4n6 Dw1 M4rthen
Penyingkatan	Bntng Dw Mrthen
Kombinasi ketiganya	b1ntN6 Dw mrthn

Tabel 2. Kombinasi ketiga variasi bahasa alay Indonesia.
Sumber: Dokumentasi Penulis

Cara yang dapat digunakan untuk menangani ini adalah dengan menggunakan Regular Expression (Regex). Lakukan konversi pola karakter alay hingga dapat dikembalikan ke bentuk alfabetik yang bersesuaian. Setelah menggunakan Regex, Anda akan diminta kembali untuk melakukan *pattern matching* antara nama yang bersesuaian dengan algoritma KMP dan BM dengan ketentuan yang sama seperti saat pencocokan sidik jari. Sebagai referensi bahasa alay, Anda dapat menggunakan *website* alay generator yang terdapat pada bagian referensi.

Setelah menemukan nama yang bersesuaian, Anda dapat menggunakan basis data untuk mengembalikan detail biodata lengkap individu. Jika seluruh prosedur diatas diimplementasikan dengan baik, maka sebuah sistem deteksi individu berbasis biometrik dengan sidik jari telah berhasil untuk diimplementasikan.

Penggunaan Program

Pada Tugas Besar kali ini, sistem yang dibangun akan diimplementasikan dengan basis ***desktop-app*** menggunakan **bahasa C#**. Saran dari asisten, Anda dapat menggunakan kakas **WinForm atau WPF** untuk membangun GUI. Masukan yang akan diberikan oleh pengguna

saat menggunakan aplikasi adalah **sebuah citra sidik jari**. Selain itu program perlu untuk **memiliki basis data SQL** dengan struktur relasional seperti yang telah dijelaskan sebelumnya. Tampilan layout dari aplikasi yang akan dibangun adalah sebagai berikut.

Gambar 4. Referensi tampilan UI *desktop-app*.
Sumber: Dokumentasi Penulis

Anda dapat menambahkan menu lainnya seperti gambar, logo, dan sebagainya. Tampilan *front-end* dari *desktop-app* **tidak harus sama persis** dengan *layout* yang diberikan di atas, tetapi dibuat semenarik mungkin dan **wajib mencakup komponen-komponen berikut**:

- 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.

Secara umum, berikut adalah cara umum penggunaan program:

1. Pengguna terlebih dahulu memasukkan citra sidik jari yang ingin dicari biodatanya.
2. Setelah citra dimasukkan, pilih opsi pencarian, ingin melakukan pencarian apakah akan menggunakan algoritma KMP atau BM.
3. Tekan tombol search, program kemudian akan memproses, mencari citra sidik jari dari basis data yang memiliki kemiripan dengan citra sidik jari yang menjadi masukan.
4. Program akan menampilkan list biodata jika ditemukan citra sidik jari yang memiliki kemiripan dengan batas persentase tertentu. Program juga dapat mengeluarkan informasi tidak ada sidik jari yang mirip jika semua citra dalam basis data tidak memiliki kemiripan dengan masukan.
5. Terdapat informasi terkait waktu eksekusi program dan persentase kemiripan citra.

Spesifikasi Tugas Besar

Pada Tugas Besar ini, buatlah sebuah sistem yang dapat melakukan identifikasi individu berbasis biometrik dengan menggunakan sidik jari dengan detail sebagai berikut.

1. Sistem dibangun dalam **bahasa C#** dengan kakas Visual Studio .NET yang mengimplementasikan **algoritma KMP, BM,** dan **Regular Expression** dalam mencocokkan sidik jari dengan biodata yang berpotensi rusak. Pelajarilah C# desktop development, **disarankan untuk menggunakan Visual Studio** untuk mempermudah pengerjaan.
2. Program dapat memiliki basis data **SQL** yang telah mencocokkan berkas citra sidik jari yang telah ada dengan seorang pribadi. Basis data yang digunakan dibebaskan asalkan **bukan No-SQL** (sebagai contoh, MySQL, PostgreSQL, SQLite).
3. Program dapat menerima masukan sebuah citra sidik jari yang ingin dicocokkan. Apabila citra tersebut memiliki kecocokan di atas batas tertentu (silakan lakukan *tuning* nilai yang tepat) dengan citra yang sudah ada, maka tunjukkan biodata orang tersebut. Apabila di

bawah nilai yang telah ditentukan tersebut, memunculkan pesan bahwa sidik jari tidak dikenali.

4. Program memiliki keluaran yang **minimal** mengandung seluruh data yang terdapat pada contoh antarmuka pada bagian penggunaan program.
5. Pengguna dapat memilih algoritma yang ingin digunakan antara KMP atau BM.
6. Biodata yang ditampilkan harus biodata yang memiliki nama yang benar (gunakan Regex untuk memperbaiki nama yang rusak dan gunakan KMP atau BM untuk mencari orang yang paling sesuai).
7. Program memiliki antarmuka yang *user-friendly*. Anda juga dapat menambahkan fitur lain untuk menunjang program yang Anda buat (unsur kreativitas).

BAB 2
LANDASAN TEORI

A. Algoritma KMP (Knuth-Morris-Pratt)

Algoritma Knuth-Morris-Pratt (KMP) adalah algoritma efisien untuk menemukan semua kemunculan dari suatu pola (substring) dalam teks (string) utama. Algoritma ini dikembangkan oleh Donald Knuth, Vaughan Pratt, dan James H. Morris pada tahun 1977. Keunggulan utama dari algoritma KMP adalah kemampuannya untuk melakukan pencarian dengan kompleksitas waktu linear, yaitu O(n + m), di mana n adalah panjang teks dan m adalah panjang pola. KMP mencapai efisiensi ini dengan menghindari perbandingan berulang karakter dalam teks yang sudah diketahui tidak cocok dengan pola. Hal ini dilakukan melalui penggunaan tabel "prefix function" atau "failure function" yang mengandung informasi tentang panjang prefiks terpanjang dari pola yang juga merupakan sufiks.

Proses algoritma KMP terdiri dari dua tahap utama: tahap pra-pemrosesan dan tahap pencarian. Pada tahap pra-pemrosesan, algoritma membangun tabel prefix function berdasarkan pola yang diberikan. Tabel ini membantu dalam menentukan langkah pergeseran yang optimal ketika terjadi ketidakcocokan karakter saat pencarian. Pada tahap pencarian, algoritma menggunakan tabel prefix function untuk menggeser pola dengan cepat tanpa harus mengulang pemeriksaan dari awal teks. Setiap kali terjadi ketidakcocokan antara karakter dalam teks dan pola, tabel ini digunakan untuk menentukan posisi baru pola, memungkinkan pencarian untuk melanjutkan dari posisi yang lebih maju daripada kembali ke awal pola. Dengan demikian, KMP mampu mencari pola dalam teks secara efisien dan menghindari redudansi perbandingan, membuatnya sangat efektif untuk aplikasi pencarian string dalam teks panjang.

B. Algoritma BM (Boyer-Moore)

Algoritma Boyer-Moore (BM) adalah metode pencarian string yang sangat efisien dan salah satu algoritma pencarian substring yang paling umum digunakan dalam aplikasi praktis. Dikembangkan oleh Robert S. Boyer dan J Strother Moore pada tahun 1977, algoritma ini

dikenal dengan kemampuannya melakukan pencarian lebih cepat daripada algoritma pencarian lainnya seperti Knuth-Morris-Pratt (KMP) terutama dalam aplikasi dengan alfabeta yang besar atau string yang sangat panjang. Uniknya, Boyer-Moore melakukan pencarian dari arah akhir pola menuju awal pola, berbeda dengan algoritma pencarian string lain yang biasanya bekerja dari awal ke akhir. Kecepatannya berasal dari dua heuristik yang digunakannya, yaitu heuristik ‘bad character’ dan ‘good suffix’, yang secara signifikan mengurangi jumlah perbandingan karakter yang diperlukan dalam proses pencarian dengan cara cerdas melompati bagian-bagian dari teks yang tidak mungkin menjadi awal kecocokan dengan pola.

Heuristik 'bad character' pada algoritma Boyer-Moore menyarankan pergeseran pola melewati beberapa karakter yang tidak cocok dengan pola berdasarkan informasi terakhir di mana karakter yang tidak cocok itu muncul di dalam pola. Jika karakter yang tidak cocok tidak ada dalam pola, pola dapat bergeser sepenuhnya melewati karakter yang tidak cocok tersebut. Heuristik 'good suffix', di sisi lain, berfokus pada ketika pencocokan karakter gagal setelah beberapa karakter di akhir pola sudah cocok. Berdasarkan ini, algoritma mencari di pola untuk kemunculan terakhir dari sufiks yang baik ini dan menggunakan informasi tersebut untuk menggeser pola ke posisi yang memaksimalkan kesesuaian sufiks ini tanpa perlu membandingkan ulang karakter yang sudah cocok. Kombinasi kedua heuristik ini membuat Boyer-Moore jauh lebih cepat daripada metode pencarian tradisional karena mengurangi jumlah perbandingan yang diperlukan secara dramatis, terutama pada teks yang panjang dengan banyak karakter non-pola. Algoritma ini sangat efektif dalam pencarian pada teks yang besar atau ketika pola relatif panjang dan beragam.

C. Regex

Regular Expressions (Regex) merupakan suatu teknik dalam teori bahasa formal dan linguistik komputasional yang digunakan untuk mencari dan manipulasi string berdasarkan pola tertentu. Regex banyak digunakan dalam pemrograman untuk pencarian (searching), penggantian (substitution), dan manipulasi teks. Pada dasarnya, Regex menyediakan sarana untuk pencocokan string terhadap suatu pola yang fleksibel dan ekspresif. Pola ini ditentukan menggunakan berbagai simbol yang menentukan "apa yang harus dicocokkan" dalam teks.

Misalnya, dalam Python dan banyak bahasa pemrograman lain, `\d` dapat digunakan untuk mencocokkan digit numerik, `.` (titik) digunakan untuk mencocokkan karakter apa pun, dan `*` (bintang) menunjukkan pengulangan nol atau lebih dari elemen sebelumnya. Dengan menggunakan kombinasi dari berbagai simbol ini, pengguna dapat membangun ekspresi yang sangat spesifik yang mampu menargetkan hampir semua urutan atau pola teks yang dapat dibayangkan.

Regex memanfaatkan beberapa prinsip utama untuk melakukan pencocokan string. Misalnya, "brackets" (`[]`) digunakan untuk mendefinisikan suatu kelas karakter di mana salah satu karakter di dalam bracket tersebut harus cocok. Penggunaan tanda `-` di antara dua karakter mendefinisikan range karakter. Selain itu, penggunaan caret (`^`) dalam konteks bracket (misalnya, `[^a-z]`) menciptakan negasi, yang artinya mencocokkan karakter apa pun yang tidak terdaftar di dalam bracket. Dalam konteks pengembangan perangkat lunak, Regex menjadi sangat berguna untuk validasi input, seperti memeriksa format email atau nomor telepon dalam formulir, mencari dan mengganti teks dalam dokumen, atau untuk scraping data dari situs web. Alat seperti RegexPal atau library `re` dalam Python menyediakan sarana interaktif dan programatik untuk menggunakan regex, yang memperkuat utilitasnya dalam berbagai aplikasi, mulai dari pengembangan web hingga analisis data dan kecerdasan buatan, seperti yang digunakan dalam pembangunan chatbot berbasis pola seperti ELIZA.

D. Teknik Pengukuran Persentase Kemiripan

Pada tugas ini digunakan 2 algoritma utama pattern matching yaitu KMP dan BM serta satu algoritma tambahan lain yaitu perhitungan Levenshtein Distance. Jika pencarian dan pencocokan dapat dilakukan dan ditemukan dengan menggunakkan algoritma pattern matching yaitu KMP dan BM maka persentase kemiripan sidik jari akan langsung dianggap 100%. Ketika algoritma KMP dan BM tidak bisa menemukan gambar sidik jari yang sesuai maka proses pencocokan dan pencarian akan berganti dengan menggunakan algoritma Levenshtein Distance. Perhitungan persentase kemiripan pada algoritma Levenshtein Distance adalah sebagai berikut:

$$\left(1 - \frac{Levenshetin\ Distance}{Panjang\ Maksimal}\right) x\ 100\%$$

E. Aplikasi Desktop

Aplikasi desktop pada tugas ini dibangun dengan menggunakan bahasa C# dengan menggunakan bantuan dari library WPF (Windows Presentation Foundation), yang merupakan salah satu framework modern untuk pembuatan aplikasi desktop pada platform Windows. WPF menyediakan sistem pengaturan layout yang kaya serta dukungan untuk grafik vektor, sehingga memungkinkan pengembangan antarmuka pengguna yang lebih dinamis dan menarik secara visual. Dengan WPF, aplikasi ini memanfaatkan XAML (Extensible Application Markup Language) untuk mendefinisikan elemen UI secara deklaratif, yang memisahkan logika tampilan dari logika bisnis, sehingga memudahkan dalam maintenance dan pengembangan lebih lanjut. Aplikasi ini juga mengintegrasikan teknologi seperti data binding, animasi, dan template, yang semuanya dapat diatur dalam XAML, sementara C# digunakan untuk menangani logika di balik aplikasi. Hal ini tidak hanya meningkatkan efisiensi dalam pengembangan tetapi juga memberikan performa yang kuat dan pengalaman pengguna yang mulus. Penggunaan teknologi ini menjamin bahwa aplikasi dapat berkembang sesuai dengan kebutuhan pengguna dan dapat dengan mudah diadaptasi untuk memenuhi tantangan baru yang mungkin muncul seiring berjalannya waktu.

BAB 3
ANALISIS PEMECAHAN MASALAH

A. Langkah - Langkah Pemecahan Masalah

1. Pendefinisian Masalah

Pada era digital ini, keamanan data dan akses menjadi semakin penting. Kami mengembangkan aplikasi pencocokan sidik jari menggunakan C# dan WPF. Aplikasi ini bertujuan untuk memungkinkan pengguna mengunggah gambar sidik, memilih algoritma pencocokan antara Boyer-Moore (BM) atau Knuth-Morris-Pratt (KMP), dan mendapatkan hasil pencocokan yang berupa biodata dari pemilik sidik jari dengan cepat dan akurat. Aplikasi ini juga dapat menggunakan algoritma Levenshtein Distance jika tidak ditemukan kecocokan dengan BM atau KMP, untuk mencari gambar dengan persentase kemiripan tertinggi.

2. Menetapkan Desain Solusi

Setelah pendefinisian masalah, kami menetapkan desain sistem yang akan kami gunakan, dimulai dari framework yang akan digunakan, bahasa pemrograman yang akan digunakan, dan alur dari aplikasi. Aplikasi ini akan menggunakan WPF untuk membangun antarmuka pengguna dan bahasa pemrograman yang digunakan adalah C#. UI terdiri dari sebuat input field box yang bisa menerima image dari pengguna (yang kemudian akan ditampilkan setelah di upload), toggle button (yang bisa digunakan oleh user untuk memilih algoritma yang akan digunakan), lalu tombol search (untuk melakukan pencarian). Kemudian, akan ada juga sebuah display box yang akan menampilkan biodata hasil pencarian, beserta waktu pencarian dan persentase kemiripannya. Saat pengguna menekan tombol pencarian, aplikasi akan memproses gambar yang diunggah dengan memilih 32 bit terbaik dan mengubahnya ke format ASCII. Aplikasi akan mengakses database untuk mengambil seluruh file path gambar sidik jari yang terdaftar dan mencocokkan pola ASCII menggunakan algoritma yang dipilih oleh pengguna.

3. **Perancangan Database**

Aplikasi ini akan menggunakan basis data yang menyimpan informasi biodata pengguna serta file path gambar sidik jari pengguna. Kami membuat dua tabel dalam basis data kami, yaitu tabel biodata dan tabel sidik_jari. Kami kemudian melakukan seeding database dengan data pengguna, dan untuk sidik jari, kami menyimpan file path dari gambar sidik jari tersebut. Setiap data (selain tanggal dan jenis kelamin) akan dienkripsi terlebih dahulu sebelum dimasukkan ke dalam database. Proses enkripsi ini bertujuan untuk mengamankan data dari ancaman pihak ketiga. Hal ini penting karena data yang kami simpan merupakan data kredensial yang mencakup biodata dan sidik jari. Untuk mempermudah pengguna dalam melakukan perancangan basis data ini, kami menyediakan Docker yang dapat mempermudah proses setup tanpa perlu konfigurasi basis data yang rumit. Aplikasi akan mengakses seluruh file path gambar yang ada di database dan mencocokkan gambar tersebut dengan gambar yang di input oleh pengguna. Setelah ditemukan pencocokan yang berhasil, biodata dari pemilik gambar sidik jari tersebut akan dicari dan ditampilkan.

4. **Pemrosesan dan Pengoptimalan**

Setelah pengguna menekan tombol pencarian, gambar yang di input oleh pengguna akan diproses. Proses ini dimulai dengan mengubah gambar menjadi format biner, kemudian dari biner tersebut akan dipilih 64 bit terbaik yang kemudian dikonversi ke format ASCII. Selanjutnya, aplikasi akan memuat semua file path sidik jari yang ada di database. Untuk setiap path, gambar akan diakses dan disimpan dalam format bitmap, lalu bitmap tersebut akan diubah menjadi format biner dan kemudian menjadi format ASCII.Pencocokan gambar akan dilakukan dengan menggunakan algoritma Boyer-Moore (BM) atau Knuth-Morris-Pratt (KMP) sesuai dengan pilihan pengguna. Proses ini akan diulang untuk semua file path yang tersedia di database. Namun, jika ditemukan kecocokan pada salah satu gambar, proses pencocokan akan dihentikan.

Untuk optimasi, kami menggunakan multithreading untuk mempercepat proses pengubahan gambar menjadi format ASCII. Proses multithreading ini dilakukan dengan membagi gambar per baris sehingga dapat dikerjakan oleh beberapa thread sekaligus.Jika

setelah menelusuri seluruh path di database tidak ditemukan kecocokan, maka algoritma Levenshtein Distance akan digunakan untuk kembali membandingkan setiap gambar dan mencari persentase kemiripannya. Gambar dengan persentase kemiripan tertinggi akan ditampilkan sebagai hasil. Untuk algoritma Levenshtein Distance, kami juga menggunakan multi-threading sehingga bisa melakukan pencocokan terhadap beberapa gambar sekaligus.

5. **Respon ke Antarmuka Pengguna**

   Hasil pencocokan sidik jari, termasuk waktu pencarian dan persentase kemiripan, akan ditampilkan di antarmuka pengguna. Aplikasi akan menampilkan hasil pencocokan yang berupa biodata dari pemilik sidik jari dalam bentuk teks yang mudah dipahami di dalam suatu display box, kemudian juga akan ditampilkan gambar dari database yang cocok dengan gambar yang di input oleh pengguna. Hal ini memungkinkan pengguna untuk melihat hasil pencocokan dengan jelas dan detail.

6. **Pengujian**

   Setelah aplikasi selesai dikembangkan, kami melakukan pengujian terhadap aplikasi kami. Pengujian dilakukan dengan menggunakan beberapa test case untuk memastikan aplikasi dapat mencocokkan sidik jari dengan akurat dan cepat. Pengujian juga dilakukan untuk mengevaluasi performa aplikasi dalam hal kecepatan pencocokan dan penggunaan sumber daya. Pengujian berulang dan revisi dilakukan untuk memastikan aplikasi dapat memberikan hasil yang konsisten dan memenuhi kebutuhan pengguna.

B. Proses penyelesaian solusi dengan algoritma KMP dan BM

Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM) adalah dua algoritma pencocokan pola yang digunakan untuk mencari substring dalam string yang lebih panjang. KMP menggunakan pendekatan untuk menghindari perbandingan ulang dengan membangun tabel border function dari pola yang menunjukkan sejauh mana pola harus digeser jika terjadi ketidakcocokan. Ini memungkinkan KMP untuk melanjutkan pencarian tanpa harus memeriksa ulang karakter yang sudah diketahui. Sementara itu, BM bekerja dari kanan ke kiri dan menggunakan dua aturan utama: looking glass dan character jump. Dalam looking

glass, pencocokan dimulai dari kanan ke kiri dari pola. Dalam character jump, BM dapat melakukan pelompatan jika terdapat ketidakcocokan, dengan menggunakan tabel last occurrence untuk menentukan lompatan dari indeks i dengan rumus $i = i + m - \min(j, 1 + \text{last occurrence})$, di mana m m adalah panjang pola, j adalah posisi karakter yang tidak cocok dalam pola, dan last occurrence last occurrence adalah posisi terakhir karakter yang muncul dalam pola.

Berikut adalah proses penyelesaian solusi:

1. **Mengubah Gambar Menjadi ASCII**

   Gambar sidik jari yang diunggah oleh pengguna akan diubah menjadi terlebih dahulu ke dalam bentuk grayscale , baru kemudian akan diganti ke binary. Setelah mendapatkan binary dari gambar, baru akan dilakukan pemilihan 64 bit terbaik berdasarkan perubahan terbesar. Setelah mendapatkan 64 bit terbaik ini, kemudian akan diubah ke dalam bentuk ASCII dengan mengambil setiap 8 bit dari 64 bit ini. Kode Ascii ini akan dijadikan sebagai pola, dengan ini untuk memudahkan pencocokan dengan menggunakan ASCII.

2. **Pencocokan dengan Algoritma KMP atau BM**

   Aplikasi kemudian akan melakukan pencocokan antara pola dengan gambar sidik jari yang ada di database (yang sudah diubah semuanya menjadi ASCII). Proses ini dimulai dengan memuat semua path gambar sidik jari dari database , kemudian akan dilakukan iterasi untuk setiap path tersebut , dan kemudian mengubah gambar dari path tersebut menjadi ASCII. Setelah itu , akan menggunakan antara KMP atau BM untuk mencocokkan pola ASCII gambar yang diunggah dengan pola ASCII gambar dari path database. Jika ditemukan kecocokan , maka proses iterasi ini akan dihentikan.

3. **Menggunakan Levenshtein Distance (LD) Jika Tidak Ada Kecocokan**

   Jika tidak ada kecocokan yang ditemukan dengan algoritma KMP atau BM, aplikasi menggunakan algoritma Levenshtein Distance untuk mencari gambar dengan persentase kemiripan tertinggi. Levenshtein Distance mengukur perbedaan antara dua string dengan menghitung jumlah operasi penyuntingan (penghapusan, penyisipan, atau penggantian) yang diperlukan untuk mengubah satu string menjadi string lainnya. Gambar dengan persentase kemiripan tertinggi ditampilkan sebagai hasil pencarian. Hasil

yang akan ditampilkan dibatasi untuk hasil dengan persentase kemiripan lebih dari 50%, karena gambar dapat mengalami korupsi, dan menurut kelompok kami, pembatasan 50% sangat tepat, karena tidak terlalu tinggi dan juga tidak terlalu rendah.

4. **Optimasi dengan Multi-Threading**

   Untuk meningkatkan kecepatan proses pencocokan, aplikasi menggunakan multi-threading. Multi-threading memungkinkan beberapa operasi dilakukan secara paralel, memanfaatkan beberapa thread CPU untuk bekerja pada bagian-bagian berbeda dari tugas yang sama secara bersamaan. Berikut adalah penjelasan mengenai bagaimana multithreading diterapkan dalam aplikasi:

   1. **Konversi Gambar ke ASCII**: Proses mengubah gambar ke format ASCII dilakukan secara paralel dengan membagi gambar menjadi beberapa baris. Setiap baris gambar diubah menjadi biner dan kemudian ASCII oleh thread yang berbeda, sehingga seluruh proses konversi menjadi lebih cepat dibandingkan dengan jika dilakukan secara berurutan.
   2. **Pencocokan Pola pada Levenshtein Distance (LD)**: Proses pencocokan pola dengan menggunakkan dengan pola Levenshtein Distance juga dilakukan secara paralel. Aplikasi memanfaatkan beberapa thread untuk membandingkan pola ASCII dari gambar yang diunggah dengan pola ASCII dari gambar-gambar di database secara simultan. Jika salah satu thread menemukan kecocokan, thread tersebut akan menghentikan proses pencocokan lebih lanjut, sehingga mengurangi waktu yang diperlukan untuk menemukan kecocokan.

C. Fitur fungsional dan arsitektur aplikasi desktop yang dibangun

1. **Fitur Fungsional**

   Ada beberapa fitur fungsional yang terdapat di aplikasi ini:

   1. **Toggle Button untuk Memilih Algoritma:** Aplikasi ini memiliki toggle button yang memungkinkan pengguna memilih algoritma pencocokan yang akan digunakan, yaitu antara Boyer-Moore (BM) atau Knuth-Morris-Pratt (KMP).
   2. **Button Search untuk Melakukan Pencarian**: Terdapat tombol pencarian yang ketika ditekan oleh pengguna, aplikasi akan memulai proses pencocokan sidik jari berdasarkan algoritma yang telah dipilih oleh pengguna.

3. **Fitur Upload Image:** Pengguna dapat mengunggah gambar sidik jari yang ingin dicocokkan melalui antarmuka aplikasi. Gambar ini akan diproses dan dibandingkan dengan data yang ada di database.
4. **Display Box untuk Menunjukkan Hasil Pencarian:** Hasil pencarian, termasuk biodata dari pemilik sidik jari yang cocok, akan ditampilkan di dalam display box. Informasi yang ditampilkan meliputi nama, alamat, dan detail lainnya yang relevan.
5. **Docker untuk Memudahkan Setup Database:** Untuk mempermudah setup basis data, aplikasi ini menggunakan Docker. Dengan file docker-compose.yaml, pengguna dapat dengan mudah mengatur dan menjalankan database tanpa perlu melakukan konfigurasi yang rumit.

2. Arsitektur Aplikasi

```
.
├── init-sql
│       └── init.sql
├── src
│   └── Tubes3_apaKek
│       ├── .vs
│       ├── bin
│       ├── db
│       │   └── Database.cs
│       ├── Models
│       │   ├── Biodata.cs
│       │   └── ResultData.cs
│       ├── Services
│       │   ├── Algo
│       │   │   ├── BoyerMooreSearch.cs
│       │   │   ├── KmpSearch.cs
│       │   │   └── LevenshteinDistance.cs
│       │   ├── Hash
│       │   │   ├── Blowfish.cs
│       │   │   └── Vigenere.cs
│       │   └── Tools
```

```
│       │       ├── Alay.cs
│       │       ├── EntropyProcess.cs
│       │       ├── ImagetoAscii.cs
│       │       └── Variance.cs
│       ├── Logic.cs
│       ├── App.config
│       ├── App.xaml
│       ├── App.xaml.cs
│       ├── AssemblyInfo.cs
│       ├── MainWindow.xaml
│       ├── MainWindow.xaml.cs
│       ├── pat.txt
│       ├── Tubes3_apaKek.csproj
│       └── Tubes3_apaKek.sln
├── test
│   └── Real
├── .gitignore
├── docker-compose.yaml
├── README.md
├── seedingBiodata.py
├── seedingBiodataHash.py
├── seedingSidikJari.py
└── seedingSidikJariHash.py
```

Untuk aplikasi ini, kami menggunakan WPF sebagai framework untuk GUI aplikasi ini. Untuk fungsi dan logic aplikasi ini, kami menggunakan bahasa pemrograman C#. Dalam root project kami terdapat beberapa folder init-sql , test , dan src. Selain itu juga beberapa file seperti .gitignore , docker-compose.yml (untuk setup basis data) , README.md (untuk informasi bagaimana menjalankan program) , seedingBiodata.py seedingBiodatHash.py , seedingSidikJari.py. seedingSidikJariHash.py (keempat file tersebut untuk melakukan seeding terhadap database yang dibuat).

Untuk folder init-sql berisi file sql yang dapat di dump sebagai inisiasi dari database (terdiri dari tabel-table).

Untuk folder test, berisi sebuah folder Real yang didalamnya terdapat file gambar sidik jari yang akan digunakan sebagai data gambar untuk melakukan pencocokan.

Untuk folder src, berisi sebuah folder Tuber3_apakek (didalamnya berisi semua source code untuk aplikasi ini) yang didalamnya berisi:

1. bin/ : Direktori ini untuk menyimpan file hasil kompilasi
2. db/ : Direktori ini berisi sebuah file database.cs yang akan digunakan sebagai kode untuk interaksi dengan dengan database.
3. Models/ : Direktori ini berisi :
    - Biodata.cs: Model data untuk menyimpan informasi biodata pengguna.
    - ResultData.cs: Model data untuk menyimpan hasil pencocokan sidik jari.
4. Services/ : Direktori ini berisi :
    - Algo/ : Direktori ini untuk menyimpan file yang berisi algoritma yang digunakan yang terdiri dari :
        1. BoyerMooreSearch.cs: File ini berisi algoritma pencocokan string dengan Boyer Moore.
        2. KmpSearch.cs: File ini berisi algoritma pencocokan string dengan Knuth–Morris–Pratt .
        3. LevensteinDistance.cs: File ini berisi algoritma pencocokan string dengan Levenshtein Distance.
    - Hash/ : Direktori ini untuk menyimpan file yang berisi :
        1. Blowfish.cs : Implementasi algoritma Blowfish untuk enkripsi.
        2. Vigenere.cs : Implementasi algoritma Vigenere untuk enkripsi.
    - Tools/ :
        1. Alay.cs: File yang berisi kode untuk memproses teks "bahasa alay".
        2. EntropyProcess.cs: File yang berisi kode untuk memproses entropi gambar.
        3. ImagetoAscii.cs: File yang berisi kode untuk mengkonversi gambar ke format ASCII.
        4. Variance.cs: File yang berisi kode untuk menghitung variansi gambar.
    - Logic.cs : Berisi logika utama aplikasi.
5. App.config: File konfigurasi aplikasi.
6. App.xaml: File XAML untuk aplikasi WPF.
7. App.xaml.cs: Kode C# untuk App.xaml.

8. AssemblyInfo.cs: Informasi tentang assembly.
9. MainWindow.xaml: File XAML untuk jendela utama aplikasi WPF.
10. MainWindow.xaml.cs: Kode C# untuk MainWindow.xaml.
11. Tubes3_apaKek.csproj: File proyek untuk Visual Studio.
12. Tubes3_apaKek.sln: File solusi untuk Visual Studio.

D. Contoh ilustrasi kasus.

Pada kasus ini, Perusahaan XYZ ingin mengimplementasikan sistem absensi berbasis sidik jari untuk meningkatkan keamanan dan efisiensi proses absensi karyawan. Setiap karyawan diharuskan untuk memverifikasi identitas mereka menggunakan sidik jari ketika masuk dan keluar dari kantor. Perusahaan memilih aplikasi pencocokan sidik jari berbasis WPF yang telah Anda kembangkan karena kemampuan aplikasinya dalam mencocokkan sidik jari dengan akurasi tinggi dan kecepatan yang efisien.

Admin IT perusahaan memulai dengan menjalankan docker-compose.yaml untuk mengatur dan menginisiasi database menggunakan Docker, diikuti dengan inisialisasi tabel-tabel yang diperlukan menggunakan skrip init.sql. Database kemudian diisi dengan data biodata karyawan dan file path gambar sidik jari yang sudah ada menggunakan skrip seedingBiodataHash.py dan seedingSidikJariHash.py.

Ketika seorang karyawan, misalnya Udin, tiba di kantor, ia mendekati perangkat absensi yang menjalankan aplikasi pencocokan sidik jari. Udin membuka aplikasi dan mengunggah gambar sidik jarinya melalui fitur upload image yang tersedia di antarmuka aplikasi. Setelah itu, Udin memilih algoritma pencocokan yang diinginkan melalui toggle button, misalnya Boyer-Moore (BM), dan menekan tombol Search untuk memulai proses pencocokan.Kemudian , akan keluar hasil pencocokan dan akan mengisi daftar absen untuk pemilik sidik jari tersebut.

Aplikasi kemudian mengonversi gambar sidik jari yang diunggah ke format biner, memilih 64 bit terbaik, dan mengkonversinya ke format ASCII. Selanjutnya, aplikasi mengakses database dan membuat semua file path gambar sidik jari karyawan. Untuk setiap file path, aplikasi mengkonversi gambar sidik jari yang tersimpan ke format biner, kemudian

ke format ASCII, dan melakukan pencocokan menggunakan algoritma BM. Jika ditemukan kecocokan, aplikasi menghentikan proses pencarian lebih lanjut dan menampilkan biodata John di display box, termasuk nama, jabatan, dan informasi lain yang relevan. Jika tidak ditemukan kecocokan, aplikasi menggunakan algoritma Levenshtein Distance untuk mencari gambar dengan persentase kemiripan tertinggi dan menampilkan hasilnya.

Proses pencocokan dipercepat menggunakan multi-threading, di mana gambar dipecah per baris dan diproses oleh beberapa thread sekaligus. Selain itu, data sensitif seperti biodata dan sidik jari dienkripsi sebelum disimpan di database menggunakan algoritma Blowfish dan Vigenere, memastikan keamanan data dari ancaman pihak ketiga.

Dengan fitur-fitur fungsional seperti toggle button untuk memilih algoritma, tombol search untuk melakukan pencarian, fitur upload image untuk mengunggah gambar sidik jari, dan display box untuk menunjukkan hasil pencarian, serta penggunaan Docker untuk setup database, aplikasi ini memberikan solusi absensi berbasis biometrik yang cepat, akurat, dan aman. Ilustrasi kasus ini menunjukkan bagaimana aplikasi pencocokan sidik jari dapat digunakan dalam lingkungan perusahaan untuk memverifikasi identitas karyawan secara efisien dan aman.

BAB 4
IMPLEMENTASI DAN PENGUJIAN

A. Spesifikasi Teknis Program

1. Struktur Data Program

a. Algoritma KMP

Algoritma KMP (Knuth-Morris-Pratt) adalah salah satu algoritma pencarian string yang efisien, terutama digunakan untuk mencari pola dalam teks. Struktur data utama yang digunakan dalam algoritma ini adalah array integer (int[] lps) yang menyimpan panjang dari prefix yang juga merupakan suffix dari pola. Array ini dikenal sebagai tabel LPS (Longest Prefix Suffix). Tabel LPS adalah array yang menyimpan panjang dari substring terpanjang yang merupakan prefix dari pola hingga posisi ke-i dan juga merupakan suffix dari substring tersebut. Tabel ini bertujuan untuk menghindari perbandingan ulang karakter dalam pola ketika ditemukan ketidakcocokan selama proses pencarian dalam teks.

Fungsi ComputeLPSArray bertanggung jawab untuk mengisi array LPS. Proses ini dimulai dengan menginisialisasi array lps dengan panjang yang sama dengan pola dan menetapkan nilai awal: lps[0] = 0 karena tidak ada proper prefix yang juga suffix untuk satu karakter. Dua variabel penting digunakan dalam proses ini: length untuk panjang dari substring terpanjang yang merupakan prefix dan suffix, serta i untuk iterasi melalui pola. Iterasi dimulai dari posisi kedua dalam pola (i = 1). Jika karakter pada posisi i cocok dengan karakter pada posisi length, maka length ditambahkan dan lps[i] diset ke length, lalu iterasi dilanjutkan ke karakter berikutnya. Jika karakter tidak cocok dan length tidak sama dengan nol, length diset ke lps[length - 1] tanpa mengubah i. Jika length sama dengan nol, lps[i] diset ke nol dan iterasi dilanjutkan ke karakter berikutnya. Sebagai contoh, untuk pola "ABABCABAB", tabel LPS untuk pola tersebut akan terlihat seperti ini: [0, 0, 1, 2, 0, 1, 2, 3, 4]. Angka pada indeks ke-i dalam tabel LPS menunjukkan panjang dari substring terpanjang yang merupakan prefix dari pola hingga posisi ke-i dan juga merupakan suffix dari substring tersebut.

Setelah tabel LPS terbangun, algoritma KMP menggunakan tabel ini untuk melakukan pencarian pola dalam teks. Proses pencarian dimulai dengan mencocokkan karakter dari pola dan teks dari kiri ke kanan. Ketika terjadi kecocokan, pencocokan dilanjutkan dengan karakter berikutnya. Namun, ketika terjadi ketidakcocokan, algoritma menggunakan nilai dalam tabel LPS untuk menentukan posisi berikutnya yang harus diperiksa dalam pola, menghindari perbandingan karakter yang tidak perlu. Misalnya, jika pola tidak cocok di posisi tertentu dalam teks, algoritma tidak perlu kembali ke awal pola, tetapi dapat melompat ke posisi yang ditentukan oleh tabel LPS. Ini mengurangi jumlah operasi yang harus dilakukan saat membandingkan karakter, karena algoritma dapat melanjutkan pencarian dari posisi yang lebih jauh tanpa harus memulai kembali dari awal. Dengan menggunakan tabel LPS, algoritma KMP dapat mencari pola dalam teks dengan efisiensi yang lebih tinggi karena mengurangi jumlah operasi yang harus dilakukan ketika karakter yang dicocokkan tidak sesuai.

b. Algoritma BM

Pada algoritma BM (Boyer-Moore), struktur data utama yang digunakan adalah dictionary (Dictionary<char, int>) yang menyimpan posisi kemunculan terakhir dari setiap karakter dalam pola. Struktur data ini dikenal sebagai tabel kemunculan terakhir (last occurrence table). Fungsi GetLastOccurrenceTable bertanggung jawab untuk membangun tabel ini, yang kemudian digunakan oleh algoritma selama proses pencarian. Tabel ini berfungsi untuk mencatat posisi terakhir setiap karakter dalam pola, yang akan digunakan untuk menentukan lompatan saat terjadi ketidakcocokan selama proses pencarian dalam teks.

Algoritma BM bekerja dengan cara mencari pola dari kanan ke kiri dan menggunakan tabel kemunculan terakhir untuk menentukan berapa banyak posisi yang dapat dilompati ketika terjadi ketidakcocokan. Misalnya, jika karakter dalam teks tidak cocok dengan karakter dalam pola, algoritma akan melompat sejauh mungkin ke kanan berdasarkan posisi terakhir karakter yang tidak cocok dalam

pola. Misalnya, jika kita mencari pola "EXAMPLE" dalam teks dan menemukan ketidakcocokan pada karakter 'A', algoritma akan menggunakan tabel kemunculan terakhir untuk menentukan bahwa 'A' terakhir kali muncul di posisi kedua dalam pola. Oleh karena itu, algoritma dapat melompat ke posisi setelah 'A' terakhir kali muncul dalam teks. Ini dilakukan dengan menghitung lompatan sebagai m - Math.Min(j, 1 + lastOccurrence.GetValueOrDefault(text[i], -1)), di mana m adalah panjang pola, j adalah indeks saat ini dalam pola, dan lastOccurrence.GetValueOrDefault(text[i], -1) memberikan posisi terakhir kemunculan karakter teks saat ini dalam pola.

Proses pencarian dimulai dengan menyelaraskan akhir pola dengan karakter dalam teks. Algoritma kemudian membandingkan karakter dalam pola dengan karakter dalam teks dari kanan ke kiri. Jika karakter cocok, algoritma melanjutkan ke karakter berikutnya di sebelah kiri. Jika terjadi ketidakcocokan, algoritma menggunakan tabel kemunculan terakhir untuk menentukan lompatan. Dengan cara ini, algoritma dapat melewati sejumlah besar karakter dalam teks yang tidak mungkin cocok dengan pola, sehingga mempercepat proses pencarian.

Keuntungan utama dari algoritma BM adalah kemampuannya untuk membuat lompatan besar saat mencari pola, yang sangat berguna untuk teks yang panjang. Kemampuan untuk melompati karakter yang tidak perlu dibandingkan membuat algoritma ini sangat efisien, terutama dalam situasi di mana pola jarang ditemukan dalam teks. Algoritma BM meminimalkan jumlah perbandingan karakter yang diperlukan, sehingga mempercepat keseluruhan proses pencarian. Ini menjadikan algoritma Boyer-Moore salah satu algoritma pencarian string yang paling efisien dan banyak digunakan dalam berbagai aplikasi yang memerlukan pencarian teks cepat.

c. Algoritma LD

Algoritma LD (Levenshtein Distance) menggunakan matriks dua dimensi (int[,] distance) untuk menghitung jarak Levenshtein antara dua string. Matriks ini menyimpan biaya transformasi dari satu karakter ke karakter lainnya, memungkinkan perhitungan jarak edit antara dua string. Setiap sel dalam matriks

ini merepresentasikan biaya minimum untuk mengubah substring dari satu string menjadi substring dari string lainnya. Proses dimulai dengan inisialisasi baris dan kolom pertama dari matriks, yang diisi dengan nilai yang meningkat dari 0 hingga panjang string target, mewakili biaya transformasi dari string kosong menjadi substring dari string target. Demikian pula, kolom pertama diisi dengan nilai yang meningkat dari 0 hingga panjang string sumber, mewakili biaya transformasi dari string sumber menjadi string kosong.

Fungsi CalculateLevenshteinDistance bertanggung jawab untuk menghitung nilai dalam matriks ini. Algoritma dimulai dengan menginisialisasi matriks, mengisi baris dan kolom pertama dengan nilai incremental. Misalnya, untuk dua string source dan target, baris pertama matriks diisi dengan nilai 0, 1, 2, ..., target.Length dan kolom pertama diisi dengan nilai 0, 1, 2, ..., source.Length. Kemudian, algoritma mengiterasi melalui sisa matriks. Untuk setiap posisi (i, j), algoritma membandingkan karakter pada posisi ke-i dari sumber dan posisi ke-j dari target. Jika karakter-karakter ini cocok, biaya transformasi adalah biaya dari sel diagonal sebelumnya.

Sebagai contoh, jika kita menghitung jarak Levenshtein antara string "kitten" dan "sitting", algoritma akan mengisi matriks dengan nilai yang merepresentasikan biaya transformasi dari satu karakter ke karakter lainnya. Misalnya, untuk mengubah "kitten" menjadi "sitting", diperlukan operasi substitusi 'k' dengan 's', substitusi 'e' dengan 'i', dan penambahan 'g' di akhir, yang totalnya membutuhkan tiga operasi. Hasil akhir dari algoritma ini adalah nilai yang terletak di sel paling kanan bawah dari matriks, yang menunjukkan jumlah minimum operasi yang diperlukan untuk mengubah satu string menjadi string lainnya.

Nilai jarak Levenshtein yang dihasilkan menunjukkan seberapa mirip dua string tersebut, yang sangat berguna dalam berbagai aplikasi seperti pengenalan pola, koreksi kesalahan teks, dan pencocokan teks. Fungsi CalculateSimilarity menggunakan nilai jarak ini untuk menentukan persentase kesamaan antara dua string, dengan rumus 1.0 - (double)distance / maxLength, di mana distance adalah

jarak Levenshtein dan maxLength adalah panjang string yang lebih panjang dari kedua string yang dibandingkan.

d. Konversi BitmapImage ke Ascii

Proses mengubah BitmapImage menjadi representasi ASCII melibatkan beberapa langkah utama: konversi BitmapImage ke Bitmap, konversi gambar ke array grayscale, dan seleksi segmen piksel yang diubah menjadi string biner dan akhirnya ke ASCII. Langkah pertama adalah mengonversi objek BitmapImage, yang biasanya digunakan dalam konteks WPF, menjadi objek Bitmap yang lebih umum digunakan dalam operasi manipulasi gambar di .NET. Fungsi BitmapImageToBitmap menggunakan BitmapEncoder untuk menyimpan BitmapImage ke dalam stream memori (MemoryStream). Stream ini kemudian digunakan untuk membuat objek Bitmap. Proses ini memastikan bahwa gambar dalam format BitmapImage dapat dimanipulasi menggunakan kelas Bitmap.

Setelah gambar dikonversi menjadi objek Bitmap, langkah selanjutnya adalah mengubah setiap piksel dalam gambar menjadi nilai grayscale. Fungsi ImageToGrayscaleArray mengiterasi setiap piksel dalam gambar, mengaksesnya menggunakan metode GetPixel, dan menghitung nilai grayscale dengan rumus (pixelColor.R * 0.3) + (pixelColor.G * 0.59) + (pixelColor.B * 0.11). Rumus ini mempertimbangkan kontribusi setiap kanal warna (merah, hijau, biru) terhadap persepsi manusia terhadap kecerahan, memberikan bobot lebih besar pada warna hijau karena mata manusia lebih sensitif terhadapnya.

Langkah berikutnya adalah mengonversi array grayscale menjadi representasi biner. Fungsi ImageToAsciiFromUniqueSegment2 mengambil array grayscale dan mengubah setiap nilai grayscale menjadi biner berdasarkan ambang batas 128 (nilai di atas 128 menjadi 1, dan nilai di bawah atau sama dengan 128 menjadi 0). Seluruh gambar kemudian direpresentasikan sebagai string biner yang panjang. Algoritma ini kemudian mencari segmen unik dari string biner dengan panjang tertentu (misalnya, 64 bit) yang memiliki jumlah perubahan maksimal antara 0 dan 1. Segmen ini dipilih sebagai representasi biner yang paling bermakna dari gambar.

Setelah segmen biner dipilih, fungsi BinaryStringToAscii mengonversi string biner ini menjadi karakter ASCII. Proses ini melibatkan pemotongan string biner menjadi bagian 8-bit dan mengonversinya menjadi karakter ASCII. Hasil akhirnya adalah representasi ASCII dari gambar asli, yang dapat digunakan untuk berbagai tujuan seperti pencocokan pola atau pengenalan teks.

Selain itu, program ini menggunakan struktur data ResultData untuk menyimpan hasil pencarian dan pencocokan sidik jari. Objek ResultData menyimpan informasi biodata yang cocok, algoritma yang digunakan, persentase kecocokan, waktu eksekusi, dan jalur file sidik jari yang ditemukan. Struktur data ini memungkinkan penyimpanan dan pengambilan informasi hasil pencarian secara terstruktur, memastikan bahwa informasi dapat diakses dan ditampilkan dengan mudah.

Struktur data Biodata digunakan untuk menyimpan informasi pribadi seseorang, termasuk nama, NIK, tempat lahir, tanggal lahir, jenis kelamin, golongan darah, alamat, agama, status perkawinan, pekerjaan, dan kewarganegaraan. Informasi ini digunakan untuk menampilkan hasil pencocokan sidik jari yang lebih rinci kepada pengguna.

Program ini juga menggunakan array of string dan StringBuilder untuk menyimpan hasil konversi gambar ke representasi ASCII, yang kemudian digunakan dalam proses pencocokan. Konversi gambar ke ASCII dilakukan dengan membaca gambar piksel demi piksel, mengubahnya menjadi nilai grayscale, dan kemudian menjadi string biner yang direpresentasikan dalam bentuk ASCII. Implementasi ini dilakukan secara paralel untuk meningkatkan kinerja, dengan menggunakan Parallel.For dan opsi paralelisme untuk membatasi jumlah thread yang digunakan.

2. Fungsi dan Prosedur Program

Dalam keberjalan program KMP, BM dan LD yang telah dijelaskan sebelumnya, maka dibutuhkan fungsi dan prosedur untuk memenuhi kebutuhan tersebut. Maka dari itu, terdapat implementasi ketiga algoritma tersebut dalam bentuk fungsi yang akan dilampirkan sebagai berikut.

a. Fungsi KMP

```
public KMPSearcher(string pattern){
        this.pattern = pattern;
        M = pattern.Length;
        ComputeLPSArray();
    }
    public int KMPSearch(string text)
    {
        int N = text.Length;

        int j = 0;

        int i = 0;
        int result = -1;
        while (i < N)
        {
            if (pattern[j] == text[i])
            {
                j++;
                i++;
            }

            if (j == M)
            {
                result = i - j;
                Console.Write("Found pattern "
                              + "at index " + (i - j));
                j = lps[j - 1];
            }
            else if (i < N && pattern[j] != text[i])
            {
                if (j != 0)
                {
                    j = lps[j - 1];
                }
                else
                {
                    i++;
                }
            }
        }
        return result;
    }
```

```
private void ComputeLPSArray()
{
    lps = new int[M];
    int len = 0;
    int i = 1;
    lps[0] = 0;

    while (i < M)
    {
        if (pattern[i] == pattern[len])
        {
            len++;
            lps[i] = len;
            i++;
        }
        else
        {
            if (len != 0)
            {
                len = lps[len - 1];
            }
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

b. Fungsi BM

```
public int BMSearch(string text){

    int n = text.Length;
    int m = pattern.Length;

    int i = m - 1;
    int j = m - 1;

    while (i < n)
    {
        if (pattern[j] == text[i])
        {
            if (j == 0)
                return i;
            i--;
            j--;
        }
        else
        {
            i += m - Math.Min(j, 1 + lastOccurrence.GetValueOrDefault(text[i], -1));
            j = m - 1;
        }
    }
    return -1;
}
```

```
public List<int> BMSearchAll(string text)
    {
        List<int> occurrences = new List<int>();
        int n = text.Length;
        int m = pattern.Length;

        int i = m - 1;

        while (i < n)
        {
            int j = m - 1;
            while (j >= 0 && pattern[j] == text[i - (m - 1 - j)])
            {
                j--;
            }

            if (j < 0)
            {
                occurrences.Add(i - m + 1);
                i += m - lastOccurrence.GetValueOrDefault(text[i], -1);
            }
            else
            {
                i += Math.Max(1, j - lastOccurrence.GetValueOrDefault(text[i - (m - 1 - j)], -1));
            }
        }
        return occurrences;
    }
```

```
private Dictionary<char, int> GetLastOccurrenceTable(string pattern)
{
    var lastOccurrence = new Dictionary<char, int>();
    for (int i = 0; i < pattern.Length; i++)
    {
        lastOccurrence[pattern[i]] = i;
    }
    return lastOccurrence;
}
```

c. Fungsi LD

```
public static class LevenshteinDistance {
        // Calculate Levenshtein Distance
    public static int CalculateLevenshteinDistance(string source, string target) {
        if (String.IsNullOrEmpty(source)) {
            return String.IsNullOrEmpty(target) ? 0 : target.Length;
        }
        if (String.IsNullOrEmpty(target)) {
            return source.Length;
        }

        int sourceLength = source.Length;
        int targetLength = target.Length;
        int[,] distance = new int[sourceLength + 1, targetLength + 1];

        // Initialize matrix of distances
        for (int i = 0; i <= sourceLength; distance[i, 0] = i++) { }
        for (int j = 0; j <= targetLength; distance[0, j] = j++) { }

        for (int i = 1; i <= sourceLength; i++) {
            for (int j = 1; j <= targetLength; j++) {
                int cost = (target[j - 1] == source[i - 1]) ? 0 : 1;

                distance[i, j] = Math.Min(
                    Math.Min(distance[i - 1, j] + 1, distance[i, j - 1] + 1),
                    distance[i - 1, j - 1] + cost);
            }
        }
        return distance[sourceLength, targetLength];
    }

    // Calculate similarity percentage based on Levenshtein Distance
    public static double CalculateSimilarity(string source, string target) {
        int maxLength = Math.Max(source.Length, target.Length);
        if (maxLength == 0)
            return 1.0; // Both strings are empty
        int distance = CalculateLevenshteinDistance(source, target);
        return (1.0 - (double)distance / maxLength);
    }
```

d. Fungsi BitmapImageToAscii

```
public static string BitmapImageToAscii(BitmapImage bitmapImage)
        {
            Bitmap bitmap = BitmapImageToBitmap(bitmapImage);
            return ImageToAsciiFromUniqueSegment2(bitmap, 64);
        }
        private static Bitmap BitmapImageToBitmap(BitmapImage bitmapImage)
        {
            using (MemoryStream outStream = new MemoryStream())
            {
                BitmapEncoder enc = new BmpBitmapEncoder();
                enc.Frames.Add(BitmapFrame.Create(bitmapImage));
                enc.Save(outStream);
                Bitmap bitmap = new Bitmap(outStream);
                return new Bitmap(bitmap);
            }
        }

        public static string ImageToAsciiFromUniqueSegment2(Bitmap image, int segmentWidth = 32)
        {
            double[] grayscaleArray = ImageToGrayscaleArray(image);

            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < grayscaleArray.Length; i++)
            {
                int binary = grayscaleArray[i] > 128 ? 1 : 0;
                sb.Append(binary);
            }

            int changeCountMax = 0;
            int bestIndex = 0;
            for (int j = 0; j <= sb.Length - segmentWidth; j += 8)
            {
                int changeCount = CountChange(sb.ToString().Substring(j, segmentWidth));

                if (changeCount > changeCountMax)
                {
                    changeCountMax = changeCount;
                    bestIndex = j;
                }
            }

            string selectedBinary = sb.ToString().Substring(bestIndex, segmentWidth);

            // Ensure binary string length is a multiple of 8
            int remainder = selectedBinary.Length % 8;
            if (remainder != 0)
            {
                selectedBinary = selectedBinary.PadRight(selectedBinary.Length + (8 - remainder), '0');
            }

            return BinaryStringToAscii(selectedBinary);
        }
```

```
public static int CountChange(string binaryString)
        {
            int change = 0;
            char prev = binaryString[0];
            for (int i = 1; i < binaryString.Length; i++)
            {
                if (binaryString[i] != prev)
                {
                    change++;
                }
                prev = binaryString[i];
            }

            return change;
        }


        private static string BinaryStringToAscii(string binaryString)
        {
            StringBuilder ascii = new StringBuilder();

            for (int i = 0; i < binaryString.Length; i += 8)
            {
                string byteString = binaryString.Substring(i, 8);
                ascii.Append((char)Convert.ToInt32(byteString, 2));
            }

            return ascii.ToString();
        }

        private static double[] ImageToGrayscaleArray(Bitmap image)
        {
            double[] grayscaleArray = new double[image.Width * image.Height];
            int index = 0;

            for (int y = 0; y < image.Height; y++)
            {
                for (int x = 0; x < image.Width; x++)
                {
                    Color pixelColor = image.GetPixel(x, y);
                    grayscaleArray[index++] = (pixelColor.R * 0.3) + (pixelColor.G * 0.59) + (pixelColor.B * 0.11);
                }
            }

            return grayscaleArray;
        }
```

B. Tata Cara Penggunaan Program

Pastikan Docker telah terinstal di sistem Anda. Jika belum, lakukan instalasi Docker sesuai dengan petunjuk resmi dari situs web Docker. Pastikan juga sudah menginstall dotnet pada sistem Anda melalui situs resmi microsoft. Jika semua sudah terinstal lakukan hal berikut:

1. Silahkan lakukan *clone repository* ini dengan cara menjalankan perintah berikut pada terminal

```
git clone https://github.com/AlbertChoe/Tubes3_apaKek.git
```

2. Jalankan perintah berikut pada terminal untuk memasuki root directory program

```
cd ./Tubes3_apaKek
```

3. Pastikan Docker Desktop telah berjalan. Setelah pengguna berada pada root directory projek ini, jalankan perintah berikut pada terminal. Note: Pastikan port 3306 tidak terpakai karena akan dipakai untuk port mysql pada docker

```
docker compose up
```

4. Masuk ke Directory src dan tubes3apakek

```
cd src/tubes3apakek
```

5. Jalankan Program dengan dotnet run

   Perintah ini akan membangun dan menjalankan aplikasi .NET yang berada di dalam direktori tubes3apakek. Tunggu hingga proses selesai dan aplikasi siap digunakan. Program ini akan mulai berjalan dan Anda dapat mulai menggunakan fitur-fitur yang disediakan oleh program.

```
dotnet run
```

6. Alternatif database

   Selain menggunakan docker bisa juga menggunakan database mysql lokal yang ada pada sistem Anda. Agar database tersebut bisa terhubung dengan baik dengan aplikasi ini pastikan untuk melakukan load init.sql file yang terdapat pada folder init-sql ke dalam database lokal Anda (Gunakan mysqldump). Pastikan juga database yang digunakan bisa diakses dengan username root dan password 1234 serta berjalan pada port 3306. Jika tidak ingin menggunakan data dari kami bisa dengan cara membuat tabel sendiri dan menambahkan data sendiri. Namun sebelum itu pastikan struktur tabel yang dibuat tetaplah sama dengan struktur tabel yang kami pakai. Setelah menambahkan data sendiri data tersebut perlu dienkripsi agar bisa digunakan dengan baik oleh aplikasi kami. Untuk mengenkripsi database bisa dengan menjalankan file enkripsiAllData.py yang terdapat di root directory dari projek kami ini.

C. Interface dan fitur fitur program

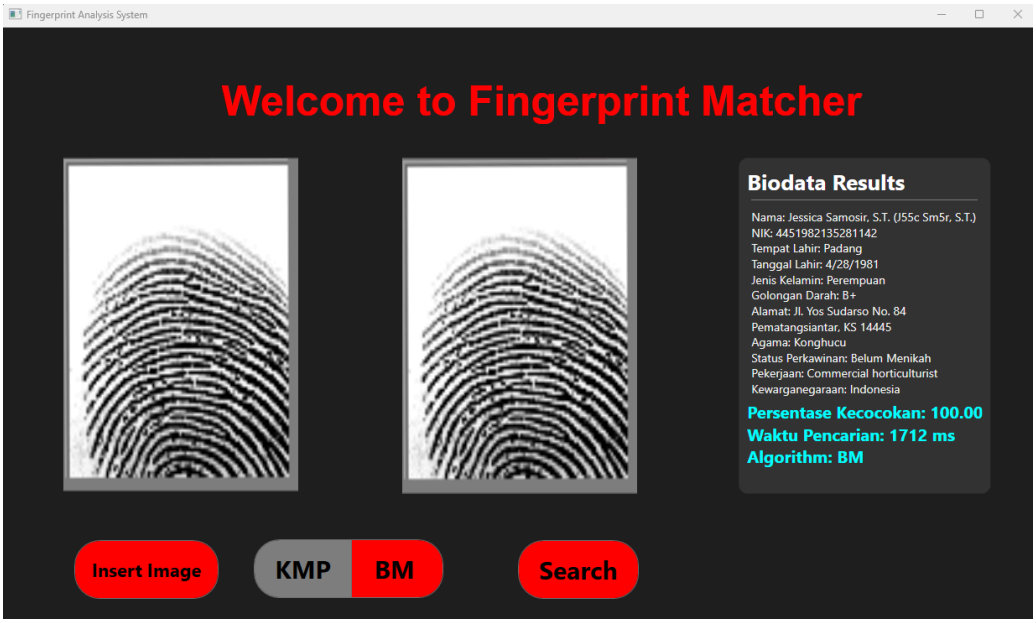
Setelah program berjalan, Anda akan melihat antarmuka pengguna (UI) yang telah dirancang untuk memudahkan interaksi dengan program. Antarmuka ini menyediakan beberapa fitur utama, termasuk:

- Unggah Gambar Sidik Jari: Tombol untuk mengunggah gambar sidik jari yang akan dicocokkan.
- Pilih Algoritma Pencarian: Pilihan untuk memilih algoritma pencarian yang akan digunakan, seperti KMP atau Boyer-Moore.
- Pencarian Sidik Jari: Tombol untuk memulai proses pencarian sidik jari dalam database.
- Hasil Pencarian: Bagian yang menampilkan hasil pencarian, termasuk informasi biodata yang cocok, persentase kecocokan, dan waktu eksekusi.
- Tampilan Gambar Sidik Jari: Tampilan gambar sidik jari yang diunggah dan gambar sidik jari yang ditemukan cocok dalam database.

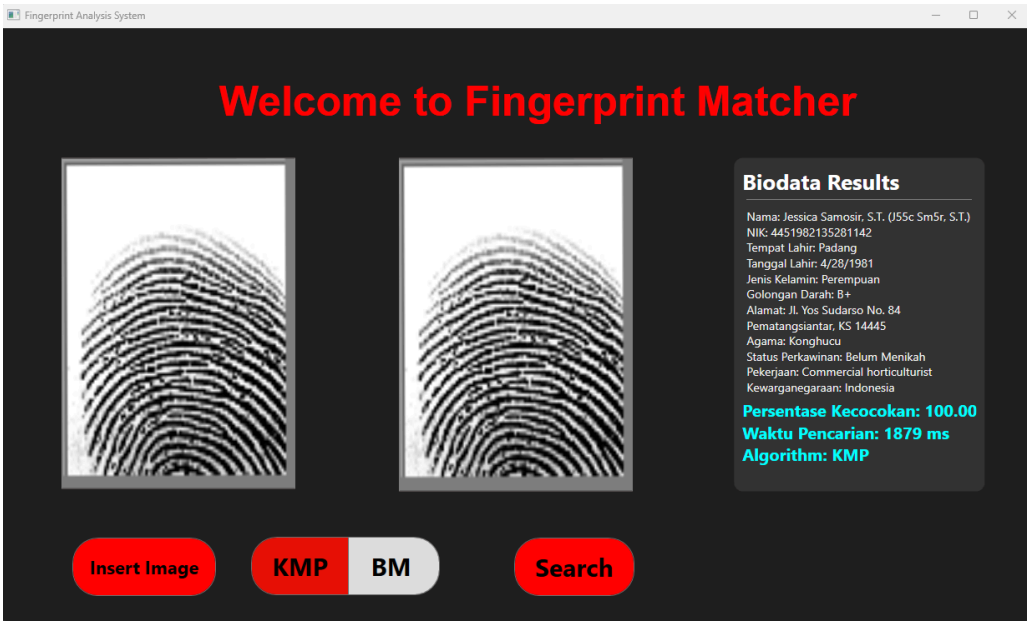
D. Hasil Pengujian

1. Test case 1

- Dengan BM

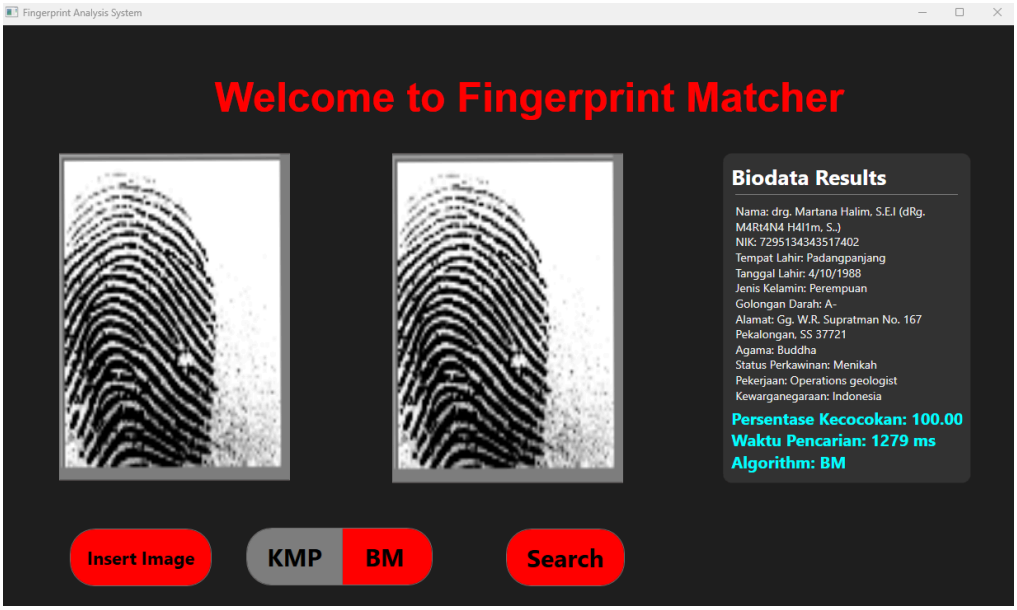


- Dengan KMP

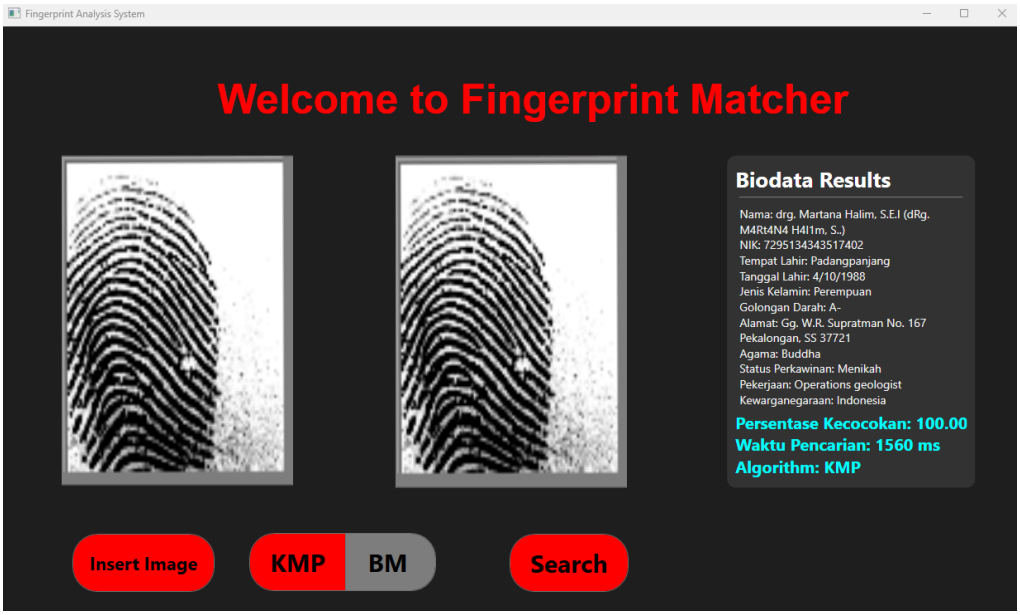


2. Test case 2

- Dengan BM

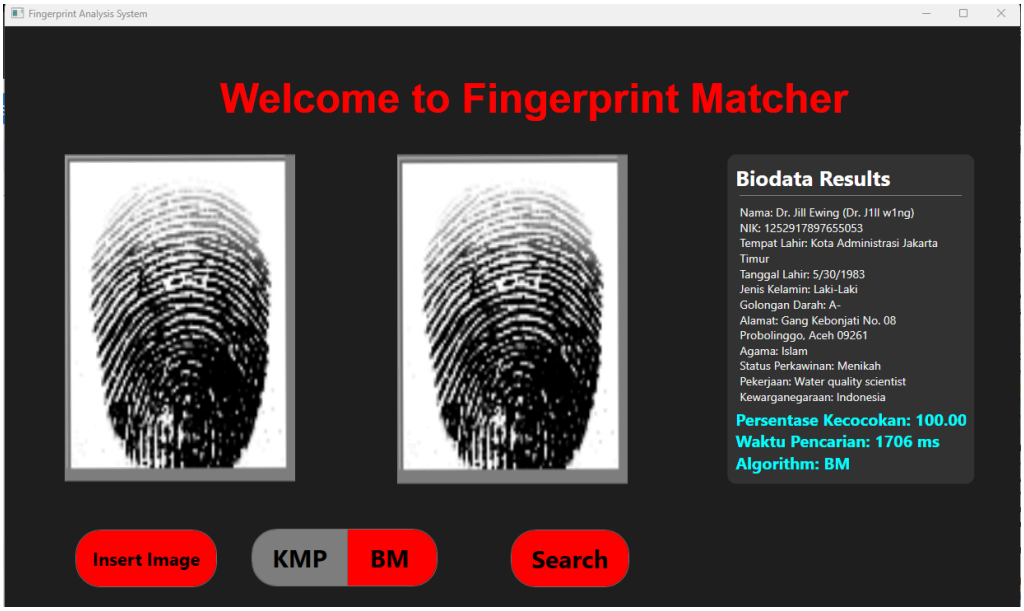


- Dengan KMP

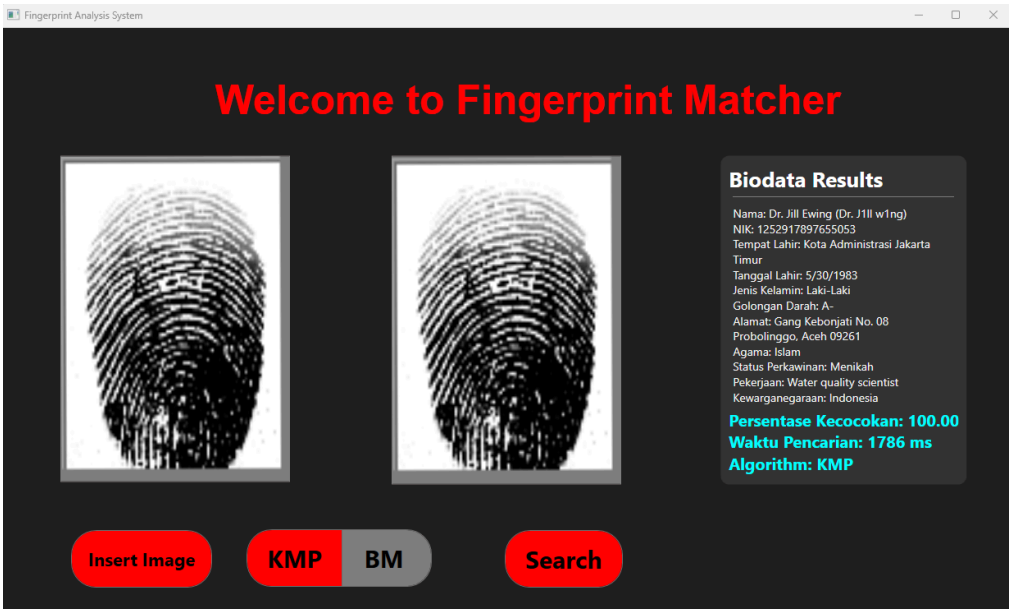


3. Test case 3

- Dengan BM

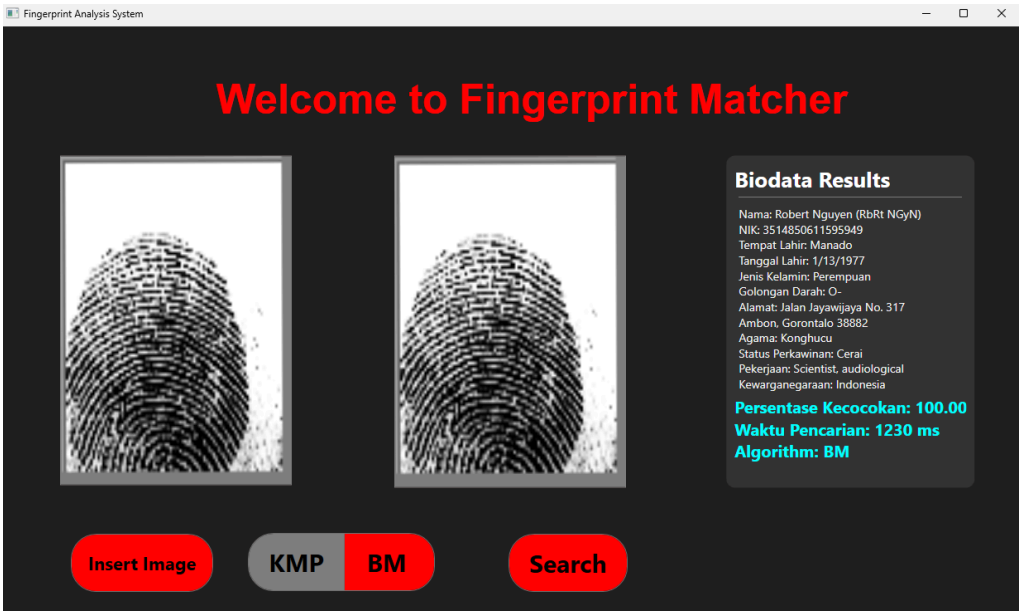


- Dengan KMP

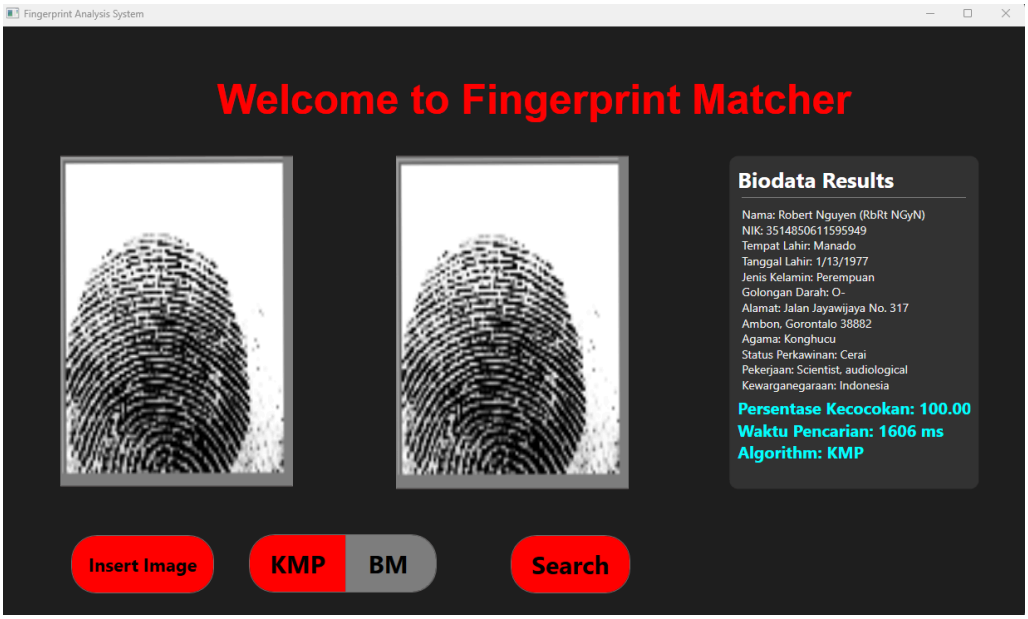


4. Test case 4

- Dengan BM

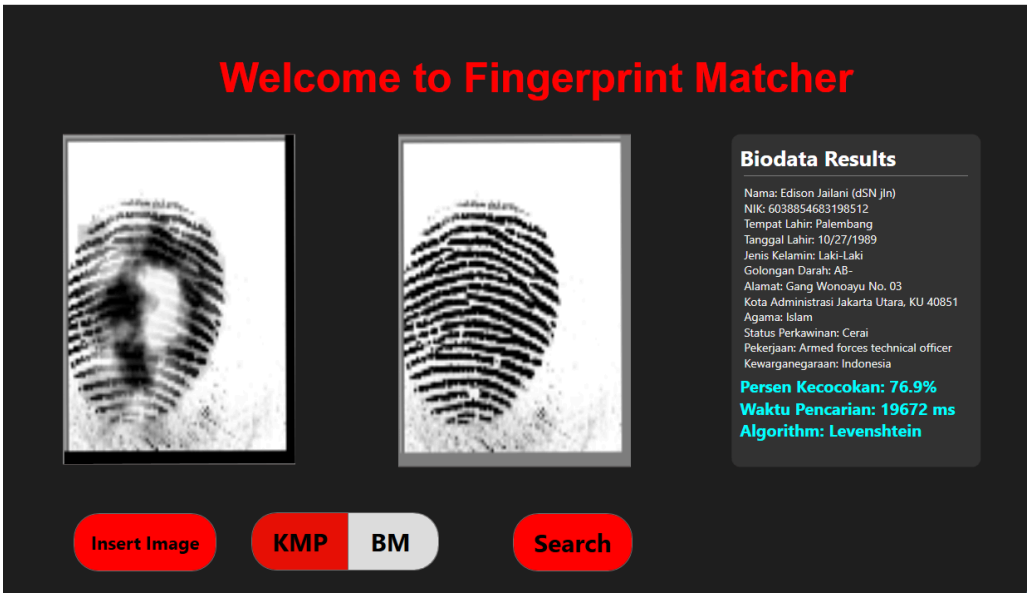


- Dengan KMP

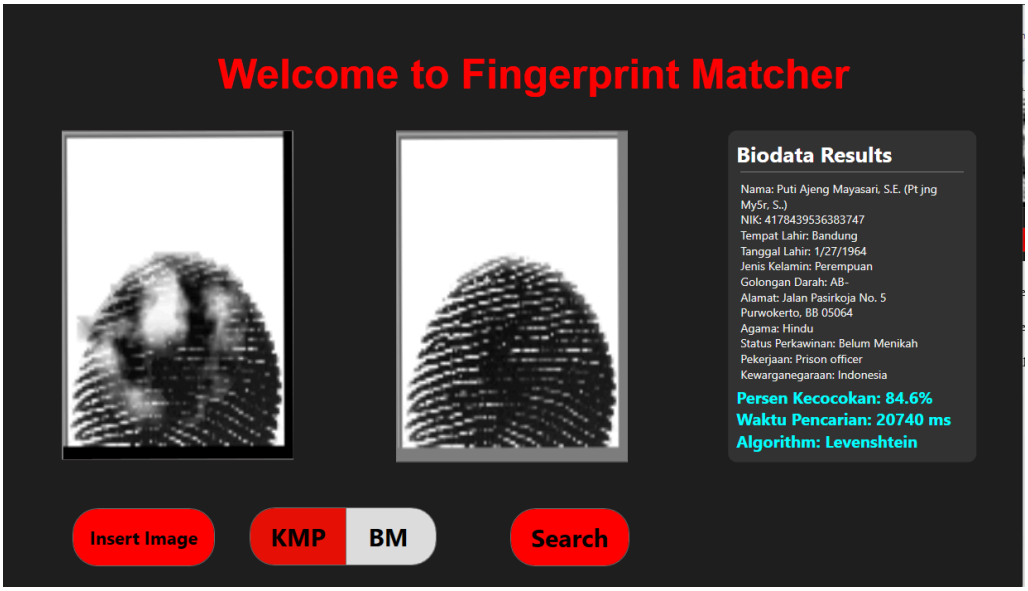


5. Test case 5

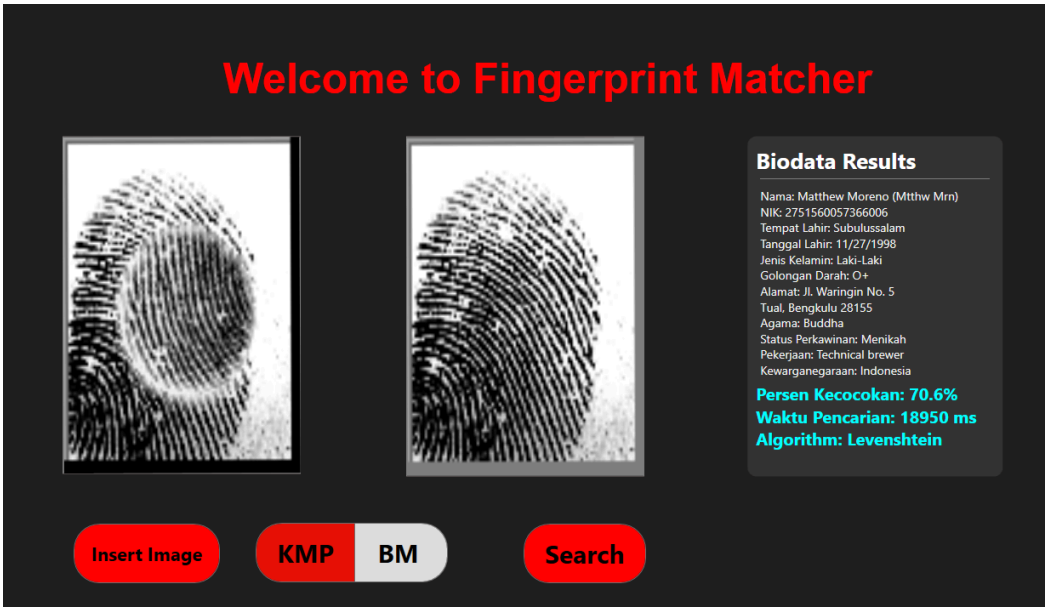
- Kasus menggunakan Levenshtein Distance



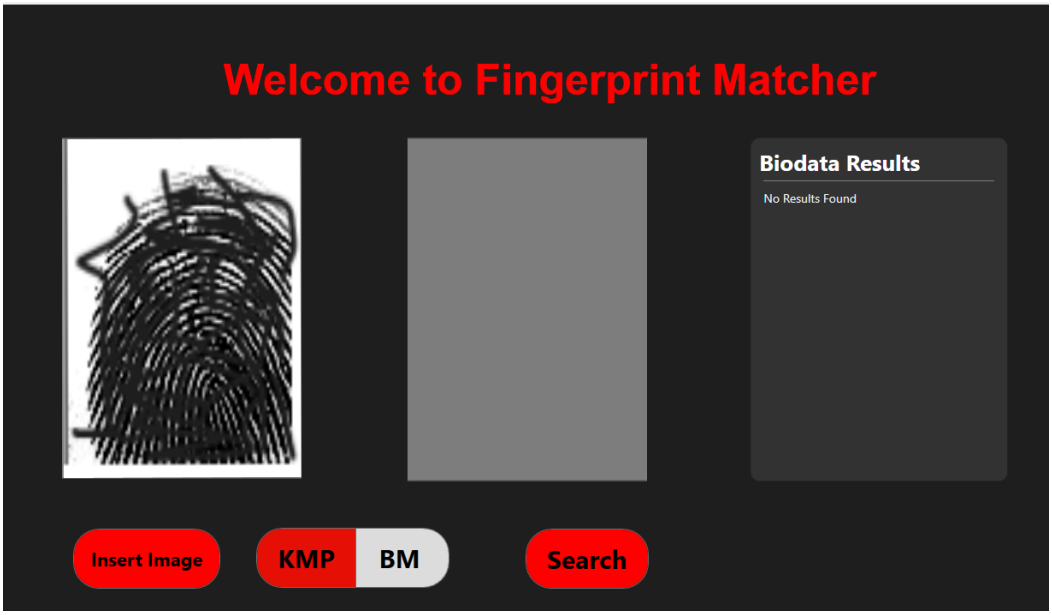
6. **Test case 6**
    - Kasus menggunakan Levenshtein Distance



7. **Test case 7**
    - Kasus menggunakan Levenshtein Distance



8. **Test case 8**
    - Kasus Tidak ada yang cocok



D. Analisis

Berdasarkan test case yang kami lakukan bisa dikatakan bahwa algoritma KMP dan BM yang kami buat telah berhasil mencari gambar sidik jari dari database yang sesuai untuk dengan gambar sidik jari yang diberikan. Ketika kedua algoritma tersebut berhasil menemukan solusi maka secara otomatis kami akan membuat persentase kemiripannya menjadi 100% dengan asumsi bahwa sebuah pola unik pada suatu sidik jari hanya dimiliki oleh satu orang saja. Oleh sebab itu ketika sebuah pola unik ditemukan pada suatu sidik jari dan kemudian pola unik tersebut digunakan untuk mencari gambar sidik jari yang sama pada database maka bisa dipastikan bahwa hanya ada satu pemilik dengan sidik jari yang sesuai sehingga bisa diasumsikan bahwa persentasenya pasti 100%.

Berdasarkan test case yang telah dilakukan bisa diketahui bahwa algoritma BM memang lebih cepat dibandingkan dengan algoritma KMP. Di semua test case yang kami lakukan bisa dilihat bahwa algoritma BM selalu lebih unggul dibandingkan dengan algoritma KMP. Hal tersebut bisa terjadi karena pada algoritma BM terdapat fungsi heuristik yang memungkinkan algoritma ini untuk melakukan lompatan yang besar ketika terdapat ketidaksamaan sebuah karakter. Berbeda dengan algoritma KMP yang hanya memanfaatkan tabel suffix untuk melakukan lompatan yang mungkin sebenarnya lompatan tersebut tidak terlalu besar. Namun

seperti yang bisa dilihat juga pada test case, perbedaan waktu diantara kedua algoritma tersebut sebenarnya tidaklah terlalu signifikan. Hal tersebut bisa terjadi karena variansi karakter yang dihasilkan dari gambar sidik jari kuranglah bervariasi sehingga mengakibatkan lompatan yang bisa dilakukan oleh algorithm BM tidak terlalu jauh berbeda dibandingkan dengan lompatan pada algoritma KMP. Di beberapa kondisi tertentu mungkin saja algoritma KMP bisa menjadi lebih cepat dibandingkan dengan algoritma BM karena secara teori seharusnya algoritma KMP bisa lebih diuntungkan dibandingkan dengan algoritma BM ketika variansi karakternya sedikit. Namun tetap saja berdasarkan test case yang telah kami lakukan bisa ditarik kesimpulan bahwa secara keseluruhan algoritma BM merupakan algoritma pattern matching yang terbaik dalam kasus pencocokan sidik jari ini.

Pola yang kami gunakan dalam algoritma KMP dan BM kami pilih dari 64 pixel (sekuensial) terbaik dari gambar sidik jari yang dimasukkan. Enam puluh empat pixel terbaik itu adalah 64 pixel dengan jumlah minutiae yang paling banyak. Kami menghitung jumlah minutiae dengan cara menghitung banyaknya transisi dari warna hitam ke putih atau dari putih ke hitam yang paling banyak. Proses ini kami lakukan pada saat gambar sudah diubah ke binary sehingga jumlah minutiae itu bisa dihitung dengan cara menghitung banyak transisi dari angka 1 ke 0 atau pun sebaliknya. Sebelumnya kami juga sudah mencoba berbagai cara lain seperti dengan menggunakan entropi dan juga variansi namun hasilnya kurang memuaskan dan kami juga kurang bisa memvisualisasikan perhitungan matematis dari kedua cara tersebut. Disini kami memilih 64 pixel karena kami rasa ini adalah ukuran untuk pattern yang ideal untuk ukuran gambar sekitar 96 x 103 pixel. Dimana 64 pixel tidak terlalu panjang yang mungkin bisa memperlama proses pencarian dan juga tidak terlalu pendek dimana data pola yang unik masih bisa diekstrak dengan baik.

Dari semua test case yang kami lakukan bisa dilihat bahwa kami telah berhasil menunjukkan data dari pemilik sidik jari yang bersangkutan dengan baik. Pada hasil, kami juga menunjukkan nama asli dari pemilik sidik jari beserta dengan nama alay yang disimpan di database. Untuk dapat menemukan nama alay yang bersesuaian dengan nama asli kami memanfaatkan regex untuk melakukannya. Dari nama asli yang kami dapat dari data gambar sidik jari, kami kemudian akan membangun sebuah regex berdasarkan nama asli tersebut untuk mencari data dengan nama alay yang bersesuaian. Ketika ditemukan data dengan nama alay yang sesuai maka kami akan langsung mengambil data tersebut dan menampilkannya

pada GUI kami. Regex yang kami buat telah kami sesuaikan dengan ciri-ciri dari bahasa alay yang dijelaskan pada spek sehingga kami bisa melakukan pencarian dengan baik dan maksimal.

Selain menguji algoritma KMP dan BM kami juga melakukan pengujian untuk algoritma alternatif yaitu Levenshtein Distance. Ketika program kami diberikan gambar sidik jari yang sudah lumayan rusak maka pada umumnya algoritma KMP dan BM tidak akan bisa menemukan solusi. Oleh sebab itu, algoritma Levenshtein Distance akan digunakan untuk melakukan pencarian solusi. Algoritma ini berbeda dengan kedua algoritma sebelumnya yang memanfaatkan keberadaan pola unik pada sidik jari. Algoritma ini lebih memilih untuk melakukan perbandingan secara menyeluruh di antara dua buah gambar sidik jari kemudian memberikan sebuah poin untuk menghitung kemiripannya. Dikarenakan algoritma ini melakukan perbandingan secara menyeluruh tentu saja waktu eksekusi dari algoritma ini menjadi lebih lama dibandingkan dengan dua algoritma sebelumnya. Hal tersebut juga bisa dilihat secara langsung dari test case yang kami lakukan dimana waktu dari algoritma Levenshtein Distance terpaut cukup jauh dibandingkan dua algoritma sebelumnya. Algoritma ini menghitung kesamaan atau perbedaan dua buah teks dengan cara menghitung jumlah penyisipan, penghapusan, dan penggantian yang mungkin dilakukan untuk membuat dua buah teks tersebut menjadi sama. Algoritma ini sangat cocok untuk mencari perbedaan dari dua buah teks karena algoritma ini bisa mempertimbangkan semua kemungkinan yang mungkin terjadi. Seperti yang bisa dilihat pada test case diatas algoritma ini juga merupakan algoritma alternatif terbaik yang bisa digunakan untuk kasus tugas besar ini karena hasil yang diberikan dari algoritma ini cukuplah memuaskan dan bisa dipercaya. Dikarenakan algoritma ini juga menggunakan sistem poin kami juga jadi bisa menghitung persentase kemiripan dengan lebih akurat seperti yang tertampil dari hasil test case diatas. Berdasarkan percobaan dan analisis yang telah kami lakukan persentase diatas 50% masihlah bisa dipercaya hasilnya oleh sebab itu batas persentase yang kami gunakan adalah 50% dan ketika persentase di bawah 50% maka kami berasumsi bahwa data sidik jari tidak ditemukan seperti yang terlihat pada test case terakhir.

BAB 5
KESIMPULAN, SARAN, TANGGAPAN, DAN REFLEKSI

A. Kesimpulan

Melalui tugas besar ini kami telah berhasil membuat sebuah aplikasi desktop dengan menggunakan bahasa C# yang berfungsi untuk melakukan pencarian gambar sidik jari yang sesuai. Dalam tugas besar ini kami juga telah berhasil memahami kegunaan dari algoritma pattern matching dan regex dalam penerapan kasus di dunia nyata. Melalui analisis yang sudah kami lakukan bisa diambil kesimpulan bahwa algoritma BM memang lebih cepat dan lebih efisien dibandingkan dengan algoritma KMP. Dalam algoritma BM terdapat fungsi heuristik yang memungkinnya melakukan lompatan yang besar yang. Dalam tugas ini kami juga memanfaatkan multi threading untuk mempercepat proses pencarian dan juga mempercepat proses pengubahan sebuah gambar menjadi ASCII. Bagian pada gambar yang akan kami gunakan sebagai pattern untuk algoritma KMP dan BM adalah bagian dari gambar yang paling unik. Kami menilai keunikan tersebut dengan memilih 64 pixel pada gambar yang mempunyai jumlah minutiae yang paling banyak. Selain algoritma KMP dan BM kami juga menggunakan algoritma alternatif yaitu Levenshtein Distance untuk menghitung kemiripan dua buah gambar ketika algoritma KMP dan BM gagal. Pilihan penggunaan algoritma Levenshtein Distance ini kami rasa juga sudah sangat tepat karena sangat sesuai dengan kebutuhan yang diperlukan dan bisa memberikan hasil yang memang kami inginkan. Regex yang kami buat pada tugas ini juga sudah sesuai dan sudah berhasil mendeteksi nama-nama orang yang menggunakan bahasa alay dengan baik. Pada tugas ini kami juga menggunakan algoritma Blowfish dan juga algoritma Vignere Cipher untuk melakukan enkripsi pada data di database untuk meningkatkan keamanannya. Pilihan untuk menggunakan algoritma Blowfish dan juga algoritma Vignere Cipher ini merupakan pilihan yang tepat karena enkripsi dari kedua algoritma ini sudah cukup baik dan mudah untuk diterapkan. Kedua algoritma tersebut juga sudah berjalan dengan sangat baik di dalam program yang telah kami buat ini. Secara keseluruhan program yang kami buat ini sudah

sangat berhasil dan sudah memenuhi semua spesifikasi yang diminta secara maksimal tanpa adanya kekurangan yang signifikan.

B. Saran

Cara kami melakukan pemilihan bagian dari gambar yang terbaik untuk dijadikan pattern pada algoritma KMP dan BM masih cukup sederhana dan seharusnya masih bisa dikembangkan lagi menjadi cara pemilihan yang lebih baik lagi. Pada tugas ini bagian dari gambar yang kami pilih juga hanya berukuran 64 pixel mungkin ukuran ini masih bisa disesuaikan lagi untuk kasus yang lebih umum dengan pertimbangan yang lebih baik lagi. Algoritma enkripsi yang kami gunakan juga masih sangat sederhana dan punya banyak kelemahan, oleh sebab itu bagian tersebut masih bisa dikembangkan lagi. Alternatif algoritma pencocokan lain yaitu Levenshtein distance mempunyai waktu yang cukup lama dibandingkan dengan algoritma KMP dan BM oleh sebab itu mungkin bisa dicari cara optimalisasinya agar lebih cepat atau bisa juga dicari alternatif algoritma lain yang lebih baik. Algoritma KMP dan BM yang kami pilih adalah algoritma KMP dan BM sederhana yang sudah diajari pada materi di kelas. Pada kenyataanya algoritma KMP dan BM mempunyai banyak sekali alternatif yang mungkin bisa dicoba untuk menghasilkan hasil yang lebih baik lagi. Aplikasi kami juga masih bisa dikembangkan agar bisa terhubung langsung dengan scanner dibandingkan harus melalui upload foto sidik jari.

C. Tanggapan

Tugas ini sangatlah menarik dan sangat membantu kami untuk lebih memahami tentang pattern matching dan juga regex. Melalui tugas ini kami telah belajar banyak hal baru dan kami bisa belajar banyak hal mengenai penerapan pattern matching dalam contoh kasus di dunia nyata.

D. Refleksi

Selama proses penulisan laporan ini, kami mendapat banyak pelajaran berharga yang tidak hanya meningkatkan kemampuan teknis kami, tetapi juga kemampuan analisis dan pemecahan masalah. Keterlibatan dalam pengembangan aplikasi ini memberikan pengalaman langsung dalam mengaplikasikan teori yang telah dipelajari di kelas ke dalam praktik nyata.

Salah satu tantangan terbesar yang kami hadapi adalah integrasi dari berbagai algoritma ke dalam satu sistem yang koheren. Proses ini memerlukan pemahaman mendalam tentang masing-masing algoritma serta kreativitas dalam mengatasi masalah yang muncul. Melalui tantangan ini, kami belajar pentingnya kerjasama tim dan komunikasi yang efektif, terutama dalam situasi di mana keputusan cepat dan efisien diperlukan.

Refleksi ini juga tidak lengkap tanpa mengakui betapa pentingnya kegigihan dan kesabaran. Ada momen-momen ketika beberapa aspek dari proyek tampak tidak mungkin diatasi. Namun, dengan dorongan tim dan tekad yang kuat, kami berhasil mengatasi hambatan tersebut.

Terakhir, pengalaman ini telah membuka pandangan kami terhadap potensi penerapan teknologi informasi dalam berbagai bidang lain dan bagaimana itu bisa berdampak pada efisiensi dan efektivitas proses kerja. Saya berharap pengetahuan dan keterampilan yang saya peroleh dari pengalaman ini dapat saya terapkan di masa depan, baik dalam studi lanjutan maupun karir profesional kami.

Kami berterima kasih kepada semua pihak yang telah mendukung kami selama proses ini, termasuk dosen pembimbing, asisten laboratorium, dan rekan-rekan tim yang telah memberikan masukan dan saran yang konstruktif. Laporan ini bukan hanya bukti dari apa yang telah kami capai, tetapi juga simbol dari apa yang dapat kami raih melalui kerja keras dan kolaborasi.

LAMPIRAN

- **Link Repository Github**

  https://github.com/AlbertChoe/Tubes3_apaKek

- **Link Video**

  https://youtu.be/SE-oPfaQFls

DAFTAR PUSTAKA

R. Munir, "Pencocokan String," 2021. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf . [Accessed: May. 29, 2024].

R. Munir, "String Matching dengan Regex," 2019. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/String-Matching-dengan-Regex-2019.pdf . [Accessed: May. 29, 2024].

"Fingerprint Identification using Bozorth and Boyer-Moore Algorithm." [Online]. Available: https://iopscience.iop.org/article/10.1088/1757-899X/662/2/022040 . [Accessed: May. 26 , 2024].