

## Laporan Tugas Besar 3

# Pemanfaatan Pattern Matching untuk Membangun Sistem ATS Berbasis CV Digital

Disusun untuk memenuhi tugas mata kuliah IF2211 Strategi Algoritma pada Semester 2 (Genap) Tahun Akademik 2024/2025



Kelompok 22 (IngfoLoker)

Razi Rachman Widyadhana 13523004

Guntara Hambali 13523114

Reza Ahmad Syarif 13523119

**PROGRAM STUDI TEKNIK INFORMATIKA  
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA  
INSTITUT TEKNOLOGI BANDUNG  
JL. GANESA 10, BANDUNG 40132  
2025**

# Daftar Isi

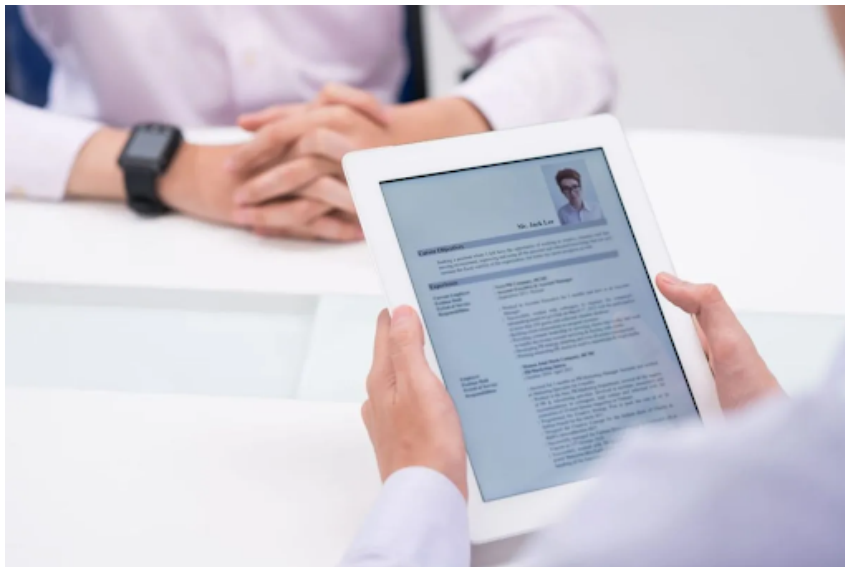
<b>Bab I: Deskripsi Masalah</b>	<b>1</b>
<b>Bab II: Algoritma KMP dan Booye-Moore dalam Pencocokan String</b>	<b>2</b>
2.1. Pencocokan String . . . . .	2
2.2. Algoritma <i>Knuth–Morris–Pratt</i> (KMP) . . . . .	2
2.3. Algoritma <i>Boyer-Moore</i> . . . . .	3
2.4. Algoritma <i>Aho–Corasick</i> . . . . .	5
2.5. Algoritma <i>Levenshtein Distance</i> . . . . .	6
2.6. Regular Expression ( <i>Regex</i> ) . . . . .	6
2.7. IngfoLoker ( <i>CV Analyzer App</i> ) . . . . .	8
<b>Bab III: Abstraksi Permasalahan</b>	<b>9</b>
3.1. File CV . . . . .	9
3.2. Kata Kunci . . . . .	9
3.3. Basis Data . . . . .	10
3.4. Pemetaan menjadi elemen-elemen KMP dan Boyer-Moore . . . . .	10
3.5. Algoritma Penyelesaian Masalah . . . . .	11
3.6. Arsitektur Aplikasi CV Analyzer . . . . .	12
3.6.1 Gambaran Umum Arsitektur . . . . .	12
3.6.2 <i>Frontend</i> . . . . .	12
3.6.3 Interaksi <i>Frontend-Controller</i> dan Basis Data . . . . .	12
3.6.4 Dockerization . . . . .	13
<b>Bab IV: Implementasi dan Pengujian</b>	<b>14</b>
4.1. Implementasi Program dengan PyQt6, Python, dan MySQL . . . . .	14
4.1.1 Algoritma Pencocokan <i>String</i> . . . . .	14
4.1.2 Algoritma <i>Encryption</i> . . . . .	22
4.2. Alur Program . . . . .	34
4.3. Eksperimen . . . . .	35
4.4. Analisis dan Pembahasan . . . . .	43
<b>Bab V: Penutup</b>	<b>45</b>
5.1. Kesimpulan . . . . .	45
5.2. Saran . . . . .	45
5.3. Refleksi . . . . .	45
<b>Lampiran</b>	<b>47</b>
<b>Referensi</b>	<b>48</b>

# Bab I

## Deskripsi Masalah

Di era digital ini, keamanan data dan akses menjadi semakin penting. Perkembangan proses rekrutmen tenaga kerja telah mengalami perubahan signifikan dengan memanfaatkan teknologi untuk meningkatkan efisiensi dan akurasi. Salah satu inovasi yang menjadi solusi utama adalah Applicant Tracking System (ATS), yang dirancang untuk mempermudah perusahaan dalam menyaring dan mencocokkan informasi kandidat dari berkas lamaran, khususnya Curriculum Vitae (CV). ATS memungkinkan perusahaan untuk mengelola ribuan dokumen lamaran secara otomatis dan memastikan kandidat yang relevan dapat ditemukan dengan cepat.

Meskipun demikian, salah satu tantangan besar dalam pengembangan sistem ATS adalah kemampuan untuk memproses dokumen CV dalam format PDF yang tidak selalu terstruktur. Dokumen seperti ini memerlukan metode canggih untuk mengekstrak informasi penting seperti identitas, pengalaman kerja, keahlian, dan riwayat pendidikan secara efisien. Pattern matching menjadi solusi ideal dalam menghadapi tantangan ini.



**Gambar 1:** CV ATS dalam Dunia Kerja (Sumber: [Antara News](#))

Pattern matching adalah teknik untuk menemukan dan mencocokkan pola tertentu dalam teks. Dalam konteks ini, algoritma Boyer-Moore dan Knuth-Morris-Pratt (KMP) sering digunakan karena keduanya menawarkan efisiensi tinggi untuk pencarian teks di dokumen besar. Algoritma ini memungkinkan sistem ATS untuk mengidentifikasi informasi penting dari CV pelamar dengan kecepatan dan akurasi yang optimal.

Di dalam Tugas Besar 3 ini, para mahasiswa IF2211 Strategi Algoritma 2024/2025 diminta untuk mengimplementasikan sistem yang dapat melakukan deteksi informasi pelamar berbasis dokumen CV digital. Metode yang akan digunakan untuk melakukan deteksi pola dalam CV adalah Boyer-Moore dan Knuth-Morris-Pratt. Selain itu, sistem ini akan dihubungkan dengan identitas kandidat melalui basis data sehingga harapannya terbentuk sebuah sistem yang dapat mengenali profil pelamar secara lengkap hanya dengan menggunakan CV digital.

## Bab II

# Algoritma KMP dan Booye-Moore dalam Pencocokan String

### 2.1. Pencocokan String

Pencocokan string adalah proses menemukan suatu pola teks di dalam teks yang lebih panjang. Misalkan  $T$  adalah string berukuran  $n$  karakter dan  $P$  adalah string pola berukuran  $m$  karakter ( $m < n$ ). Pencocokan string mencari semua indeks  $i$  sehingga  $T[i..i + m - 1] = P$ .

Algoritma pencocokan string terbagi menjadi dua kelompok besar:

#### 1. Algoritma Pencocokan String yang Tepat (Exact Matching)

Menemukan pola dengan kecocokan sempurna tanpa kesalahan. Contoh: Naïve, Knuth–Morris–Pratt (KMP), Rabin–Karp, Boyer–Moore, dan Aho–Corasick.

#### 2. Algoritma Pencocokan String Perkiraan (Fuzzy Matching)

Mengizinkan sejumlah kesalahan (penyisipan, penghapusan, substitusi, atau transposisi). Contoh: Levenshtein dan Damerau–Levenshtein.

Pada Tugas Besar 3 ini, akan digunakan KMP, Boyer–Moore, dan Aho–Corasick sebagai algoritma pencocokan string yang tepat, serta Levenshtein sebagai algoritma pencocokan string perkiraan, dalam proses *pattern matching* untuk membangun Sistem ATS (Applicant Tracking System) berbasis CV digital.

### 2.2. Algoritma *Knuth–Morris–Pratt* (KMP)

Algoritma *Knuth–Morris–Pratt* merupakan algoritma pencocokan string yang menjamin pencarian pola dengan kompleksitas waktu linear. Proses pencarian memanfaatkan pola itu sendiri melalui tabel *Border Function* sehingga perbandingan karakter yang berulang dapat dihindari. Saat terjadi *mismatch*, posisi pola digeser berdasarkan nilai Border Function tanpa memundurkan penunjuk pada teks sehingga pencarian tetap berjalan maju secara monoton.

Mekanisme internal algoritma terdiri atas dua fase, yaitu pra-proses pola dan fase pencarian. Pra-proses membangun tabel Border Function dalam waktu linear terhadap panjang pola, sedangkan fase pencarian bekerja linear terhadap panjang teks.

Secara algoritmik, tahapan KMP dapat diuraikan sebagai berikut:

#### 1. Pra-proses Pola (Konstruksi Border Function):

Inisialisasi `border[0] = 0`. Lakukan iterasi terhadap pola untuk menghitung nilai Border Function pada setiap indeks dengan membandingkan karakter pola terkini dengan karakter pada posisi `len`. Jika karakter sama, tingkatkan `len` dan simpan ke `border[i]`; jika berbeda dan `len ≠ 0`, mundurkan `len` ke `border[len-1]`; jika `len = 0`, simpan `border[i] = 0` dan lanjutkan iterasi.

$j$	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$k$	0	0	2	3	4	
$\text{border}[k]$	0	0	1	0	1	

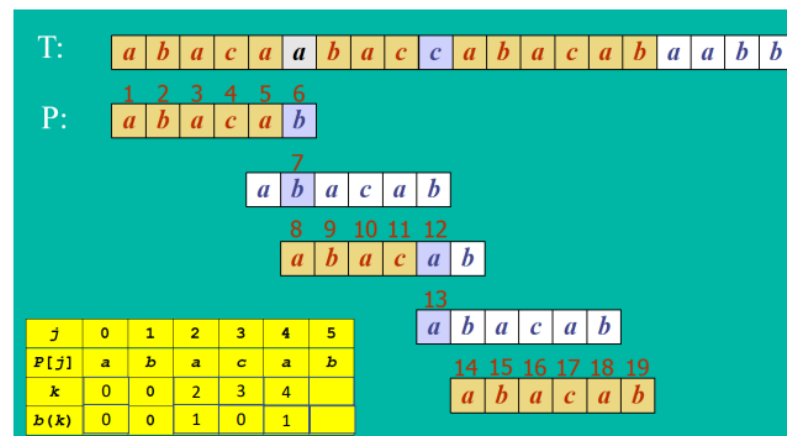
## 2. Pencarian di Teks:

Selama indeks teks  $i < n$ :

- Jika  $P[j] = T[i]$ , majukan kedua indeks  $i++$  dan  $j++$ .
- Jika  $j = m$ , pola ditemukan pada posisi  $i - j$ ; simpan posisi tersebut dan set  $j = \text{border}[j-1]$  untuk mencari kecocokan berikutnya.
- Jika terjadi *mismatch* dan  $j \neq 0$ , geser pola dengan  $j = \text{border}[j-1]$ .
- Jika terjadi *mismatch* dan  $j = 0$ , majukan indeks teks  $i++$ .

## 3. Basis:

Proses berakhir ketika indeks teks mencapai  $n$ , sehingga seluruh teks telah diperiksa.



**Gambar 2:** Ilustrasi pergerakan pola selama proses pencarian KMP

Algoritma ini banyak digunakan pada mesin pencari teks, deteksi plagiarisme, bioinformatika, dan sistem deteksi intrusi. Keunggulan utama terletak pada jaminan waktu eksekusi linear di kasus terburuk, sedangkan kelemahan berada pada konstanta waktu yang lebih besar dibanding algoritma Boyer–Moore ketika alfabet berukuran besar.

### 2.3. Algoritma *Boyer-Moore*

Algoritma *Boyer-Moore* melakukan pencocokan string dengan membandingkan karakter pola terhadap teks dari kanan ke kiri. Dua heuristik utama, yaitu **Bad Character** dan **Good Suffix** yang memungkinkan pola melompat lebih jauh setelah *mismatch*, sehingga sebagian besar karakter teks dilewati tanpa pemeriksaan.

Pada saat terjadi ketidakcocokan, pergeseran pola diputuskan melalui tiga skenario berikut:

### 1. Last Occurrence (Bad Character).

Bila karakter teks  $x$  yang menyebabkan *mismatch* muncul di pola, geser pola ke kanan sehingga kemunculan terakhir  $x$  pada pola sejajar dengan  $x$  di teks.

### 2. Shift Satu Posisi.

Jika  $x$  muncul di pola tetapi terletak di sebelah kanan posisi *mismatch*, pergeseran maksimum Bad Character bernilai negatif; pola cukup digeser satu karakter ke kanan.

### 3. Full Shift.

Apabila  $x$  tidak muncul sama sekali di pola, geser pola sehingga indeks pertama pola berada tepat di posisi teks setelah  $x$ .

Secara algoritmik, tahapan Boyer–Moore dapat diurai menjadi:

#### 1. Pra-proses Pola

Bangun *Last Occurrence Table*  $L(x)$  untuk setiap karakter alfabet dan tabel *Good Suffix* guna menentukan pergeseran minimal saat sufiks cocok sebelumnya.

Karakter ( $x$ )	a	b	c	d
$L(x)$	4	5	3	-1

#### 2. Pencarian di Teks

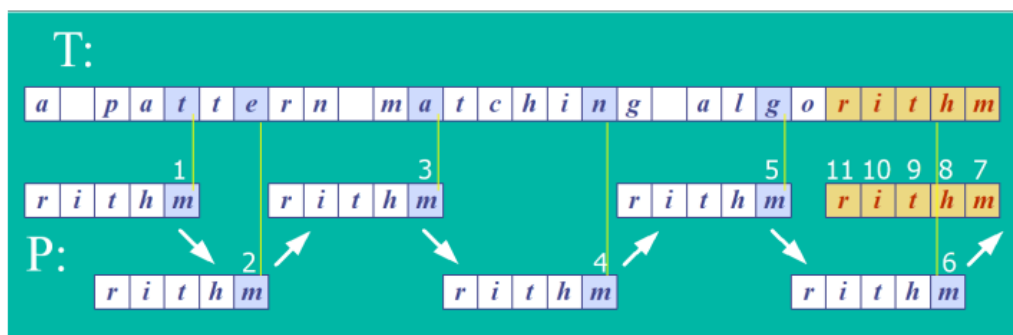
Bandingkan karakter pola dan teks dari kanan ke kiri. Jika semua karakter cocok, laporkan posisi kecocokan lalu geser pola menggunakan *Good Suffix*. Jika *mismatch* pada posisi  $j$ , hitung

$$\text{shift}_{\text{BC}} = j - L(T[i]), \quad \text{shift}_{\text{GS}} = \text{goodSuffix}[j],$$

kemudian geser pola sebesar  $\max(\text{shift}_{\text{BC}}, \text{shift}_{\text{GS}}, 1)$  dan ulangi proses.

#### 3. Basis

Pencarian selesai ketika indeks awal pola melampaui  $n - m$ .



**Gambar 3:** Ilustrasi pergerakan pola selama proses pencarian Boyer-Moore

Algoritma Boyer–Moore banyak dipakai pada utilitas pencarian teks (*grep*, *diff*), editor kode, kompresi data, dan analisis bioinformatika berkat kemampuannya melompati bagian teks yang panjang tanpa pemeriksaan karakter per karakter.

## 2.4. Algoritma Aho–Corasick

Algoritma *Aho–Corasick* menangani pencarian banyak pola secara serentak dengan membangun sebuah *finite-state automaton*. Struktur inti automaton mencakup fungsi **goto**, **failure**, dan **output** sehingga seluruh pola (kamus) dapat ditemukan dalam sekali lintasan teks tanpa *backtracking*.

Tahapan algoritmik Aho–Corasick diuraikan sebagai berikut:

### 1. Konstruksi Trie (Fungsi *goto*):

Masukkan setiap pola ke dalam trie karakter per karakter. Nodus akar mewakili string kosong; setiap tepian dilabeli karakter. Jika seluruh karakter pola dimasukkan, tandai nodus akhir dengan indeks pola pada daftar *output*.

### 2. Konstruksi Fungsi *failure*:

Lakukan penjelajahan BFS pada trie. Arahkan simpul anak akar memiliki *failure link* ke akar. Untuk simpul lain, *failure link* menunjuk ke simpul terpanjang yang merupakan sufiks proper dari jalur saat ini dan sekaligus prefiks pada trie. Output pada simpul tujuan ditambahkan ke daftar *output* simpul asal untuk memastikan pola tumpang-tindih dilaporkan.

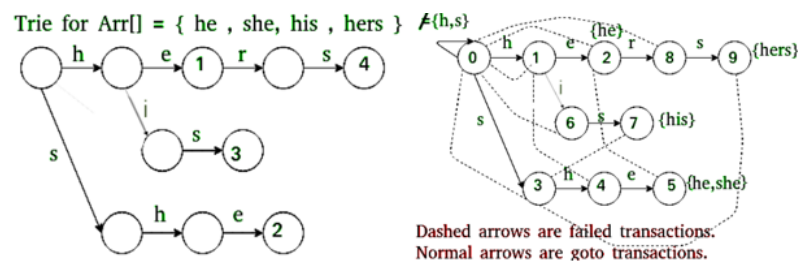
### 3. Pencarian di Teks:

Inisialisasi penunjuk keadaan pada akar. Untuk setiap karakter *c* pada teks:

- Selagi tidak ada tepi *c* dari keadaan saat ini *dan* bukan akar, ikuti *failure link*.
- Jika tepi *c* ada, berpindah ke keadaan tujuan; bila tidak, bertahan di akar.
- Keluarkan seluruh pola pada daftar *output* keadaan saat ini sebagai kemunculan pola berakhir di indeks karakter tersebut.

### 4. Basis:

Pemrosesan selesai setelah karakter terakhir teks dipindai, sehingga seluruh kemunculan pola dilaporkan.



**Gambar 4:** Contoh automaton Aho–Corasick beserta *failure link* dan daftar *output*

Algoritma Aho–Corasick banyak digunakan pada pemindaian kamus kata kunci (antivirus, IDS), pencarian entitas di teks, serta analisis bioinformatika karena mampu memproses jutaan karakter teks dengan sekali perjalanan automaton.

## 2.5. Algoritma *Levenshtein Distance*

Algoritma *Levenshtein Distance* (sering disebut *edit distance*) mengukur jarak antara dua string dengan menghitung jumlah minimum operasi **insert**, **delete**, dan **substitute** karakter yang diperlukan untuk mengubah satu string menjadi string lain. Nilai 0 menunjukkan kedua string identik, sedangkan nilai lebih besar mencerminkan tingkat perbedaan.

Secara algoritmik, tahapan perhitungan Levenshtein Distance dapat diurai menjadi:

### 1. Inisialisasi Matriks Dinamik:

Misalkan string  $A$  berukuran  $n$  dan string  $B$  berukuran  $m$ . Buat matriks  $(n + 1) \times (m + 1)$  bernama  $D$ . Set  $D[i][0] = i$  untuk setiap  $i$  (menghapus  $i$  karakter pertama  $A$ ) dan  $D[0][j] = j$  untuk setiap  $j$  (menyisip  $j$  karakter untuk membentuk  $B$ ).

### 2. Pengisian Matriks:

Untuk setiap  $i = 1 \dots n$  dan  $j = 1 \dots m$ , hitung

$$\text{cost} = \begin{cases} 0 & \text{jika } A[i] = B[j], \\ 1 & \text{jika } A[i] \neq B[j]. \end{cases}$$

Kemudian set

$$D[i][j] = \min \begin{cases} D[i-1][j] + 1 & (\text{hapus}), \\ D[i][j-1] + 1 & (\text{sisip}), \\ D[i-1][j-1] + \text{cost} & (\text{substitusi / cocok}). \end{cases}$$

### 3. Basis Hasil:

Nilai akhir  $D[n][m]$  merupakan jarak Levenshtein antara  $A$  dan  $B$ . Matriks dapat dioptimalkan ruangnya menjadi dua baris karena baris saat ini hanya bergantung pada baris sebelumnya.

Perhitungan jarak antara "kitten" dan "sitting" menghasilkan nilai akhir 3. Artinya, diperlukan tiga operasi—substitusi  $k \rightarrow s$ , substitusi  $e \rightarrow i$ , dan penyisipan  $g$ —untuk mentransformasikan "kitten" menjadi "sitting".

Levenshtein Distance banyak dimanfaatkan untuk koreksi ejaan pada mesin pencari atau editor teks, pencarian *fuzzy* yang menoleransi salah ketik pada basis data kata kunci, serta penilaian kesamaan jawaban teks pada kompetisi pemrograman atau platform pembelajaran daring.

## 2.6. Regular Expression (*Regex*)



Regular expression (*regex*) didefinisikan sebagai rangkaian simbol yang merepresentasikan pola pencarian dalam teks. Secara teori, regex berasal dari konsep bahasa formal—khususnya *regular language*—yang dapat dimodelkan dengan *finite automata*. Setiap pola diturunkan menjadi automaton deterministik/nondeterministik untuk memindai teks tanpa *backtracking* eksplisit.



Tanpa menampilkan sintaks detail, pola regex umumnya dibangun dari empat kategori elemen:

1. *Literal character* yang mewakili diri-nya sendiri.
2. *Metacharacter*—simbol abstrak untuk menyatakan kelas karakter, jangkar posisi, atau operator kuantifikasi.
3. *Quantifier* yang menentukan kardinalitas kemunculan bagian pola (mis. nol-atau-lebih, satu-atau-lebih, rentang tertentu).
4. *Grouping* dan *alternation* yang memungkinkan penyusunan sub-pola bersarang dan pilihan pola alternatif.

Prinsip pencocokan dimulai dari kiri ke kanan: engine membaca teks, menggerakkan state automaton berdasarkan karakter, dan mengeluarkan keberhasilan setiap kali mencapai state penerima dengan daftar pola yang valid.

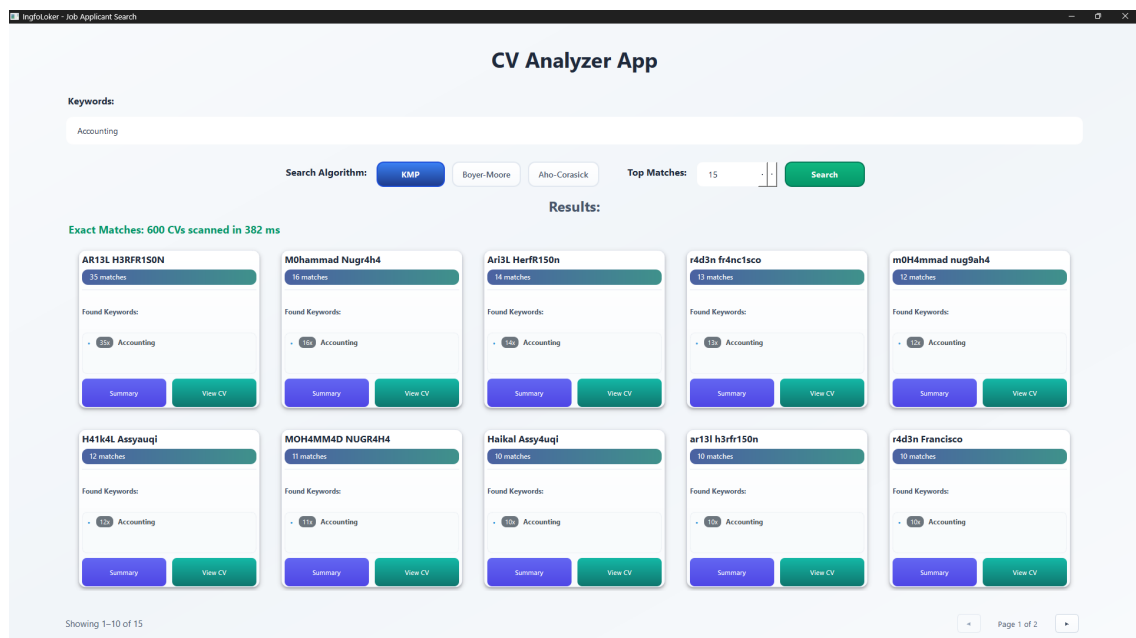
Regex book	Version History	Feedback	Blog
Options		Quick Reference	
.	Any character except newline.		 
\.	A period (and so on for \*, \ (, \\, etc.)		
^	The start of the string.		
\$	The end of the string.		
\d,\w,\s	A digit, word character [A-Za-z0-9_], or whitespace.		
\D,\W,\S	Anything except a digit, word character, or whitespace.		
[abc]	Character a, b, or c.		
[a-z]	a through z.		
[^abc]	Any character except a, b, or c.		
aa bb	Either aa or bb.		
?	Zero or one of the preceding element.		
*	Zero or more of the preceding element.		
+	One or more of the preceding element.		
{n}	Exactly <i>n</i> of the preceding element.		
{n, }	<i>n</i> or more of the preceding element.		
{m, n}	Between <i>m</i> and <i>n</i> of the preceding element.		
??,*?,+?, {n} ?, etc.	Same as above, but as few as possible.		
(expr)	Capture <i>expr</i> for use with \1, etc.		
(?:expr)	Non-capturing group.		
(?=expr)	Followed by <i>expr</i> .		
(?!expr)	Not followed by <i>expr</i> .		
<a href="#">Near-complete reference</a>			

**Gambar 5:** Ringkasan simbol dan kuantifikator regex

Regex dipakai luas di berbagai domain. Pada editor teks atau IDE, regex mempercepat fitur *find-and-replace* yang kompleks. Mesin pencari dan modul web memanfaatkan *fuzzy search* berbasis pola untuk menoleransi salah ketik. Dalam validasi data, pola digunakan memeriksa alamat surel, nomor telepon, atau format tanggal. Dalam NLP, regex membantu tokenisasi cepat ataupun pembuatan chatbot berbasis pola seperti ELIZA yang dikenalkan Weizenbaum (1966). Pada ranah keamanan, sistem deteksi intrusi (IDS) dan antivirus menggunakan regex untuk memindai tanda tangan berbahaya di aliran trafik.

## 2.7. IngfoLoker (*CV Analyzer App*)

IngfoLoker – CV Analyzer App adalah aplikasi desktop berbasis PyQt6 untuk membantu tim HR dalam mencari kandidat dari kumpulan CV (PDF) dengan cepat berdasarkan kata kunci. Pengguna dapat memasukkan daftar kata kunci, memilih algoritma pencarian eksak (KMP, Boyer-Moore, atau Aho-Corasick), dan menentukan jumlah hasil teratas yang diinginkan. Jika tidak ada hasil eksak, aplikasi secara otomatis melakukan pencarian fuzzy berbasis Levenshtein. Hasil pencarian ditampilkan dalam bentuk kartu bergaya dengan nama pelamar, jumlah kemunculan kata kunci, daftar kata kunci yang cocok beserta frekuensinya, serta tombol aksi “Summary” dan “View CV”. Fitur pagination memudahkan penelusuran apabila jumlah hasil melebihi satu halaman. Aplikasi ini juga menyediakan dialog ringkasan CV yang menampilkan informasi profil, ringkasan, keterampilan, riwayat pekerjaan, dan pendidikan, serta kemampuan membuka berkas PDF lewat browser.



**Gambar 6:** Cuplikan Tampilan Utama Pencarian dan Hasil pada IngfoLoker – CV Analyzer App

## Bab III

### Abstraksi Permasalahan

Seperti yang telah didefinisikan sebelumnya, algoritma Knuth-Morris-Pratt dan Boyer-Moore akan melakukan pencarian pola pada suatu teks. Oleh karena itu, fokus utama pada bagian ini adalah tahapan cara mengonversi string di cv menjadi teks/string yang dapat dilakukan pencocokan pola.

Dengan meninjau masing-masing komponen permasalahan terlebih dahulu, akan memudahkan proses abstraksi untuk penggunaan algoritma KMP dan Boyer-Moore dalam Pencarian Recipe ini.

#### 3.1. File CV

File CV adalah representasi data utama yang akan dianalisis dalam sistem ini. File CV umumnya berekstensi pdf dan berisi sekumpulan kata yang. Objek ini akan berfungsi sebagai "medan" pencarian, di mana semua informasi mentah seperti pengalaman kerja, keahlian, dan riwayat pendidikan tersimpan dalam format string. File ini nantinya akan dikonversi menjadi sebuah string tunggal.

Proses konversi menjadi string tunggal ini merupakan batasan desain yang bertujuan untuk mempermudah dan mengefisiensikan proses pencocokan pola menggunakan algoritma yang berbasis *string matching* (bukan *word matching*) Dengan kata lain, seluruh kompleksitas format visual PDF disederhanakan menjadi satu representasi data linear untuk dianalisis.

#### 3.2. Kata Kunci

Kata Kunci adalah objek yang merepresentasikan kriteria atau kualifikasi yang dicari oleh pengguna (misalnya, HR atau rekruter) di dalam Teks CV. Dalam algoritma, objek ini berfungsi sebagai "pola" (*pattern*) yang akan dicocokkan dengan teks CV yang sudah dikonversi. Pengguna dapat memasukkan satu atau lebih kata kunci sekaligus, yang dipisahkan dengan koma.

Interaksi antara objek Kata Kunci dan Teks CV diatur oleh dua aturan pencocokan:

- **Pencocokan Tepat (Exact Match):**

Aturan utama menyatakan bahwa setiap kata kunci harus dicari kecocokan persisnya menggunakan algoritma Knuth-Morris-Pratt (KMP) atau Boyer-Moore (BM). Kecocokan persis ini artinya semua karakter yang ada pada pola, hadir di teks sebagai substring (tidak berlaku sebaliknya).

Perlu dicatat bahwa pada tugas besar ini, pencocokannya adalah pada *string* bukan *word*. Contohnya seperti ini: misalkan sebuah teks "Saya menguasai reaction" dan sebuah pola "react".

Pada pencocokan kata tepat (*exact word matching*), maka tidak akan ditemukan pola pada teks. Ini terjadi karena teks dipandang sebagai kumpulan **kata** sehingga "react" **bukan** merupakan subkumpulan-kata (subwords) yang

berada pada teks. Sebaliknya, pada pencocokan string tepat (*exact string matching*), maka akan ditemukan pola pada teks.

Ini terjadi karena teks dipandang sebagai kumpulan **karakter** sehingga "react" adalah subkumpulan-karakter (substring) yang berada pada teks.

- **Pencocokan Mirip (Fuzzy Match):**

Jika aturan pertama gagal(jika sebuah kata kunci tidak ditemukan sama sekali melalui *exact match*) maka sistem wajib memberlakukan aturan kedua.

Aturan ini mengharuskan sistem untuk mencari kembali kata kunci tersebut berdasarkan tingkat kemiripan menggunakan algoritma Levenshtein Distance. Hal ini dilakukan untuk mengantisipasi kemungkinan adanya kesalahan pengetikan (*typo*).

Kedua aturan ini mendefinisikan bagaimana sebuah "Kata Kunci" dapat dianggap relevan terhadap sebuah "Teks CV".

### 3.3. Basis Data

Basis Data adalah objek yang berfungsi sebagai repositori informasi terstruktur mengenai pelamar dan aplikasi mereka. Sesuai spesifikasi, sistem ini menggunakan MySQL. Berbeda dengan Teks CV yang tidak terstruktur, objek Basis Data menyimpan informasi yang sudah terdefinisi dengan jelas. Penting untuk dicatat bahwa data mentah (*raw*) CV tidak disimpan di dalam basis data; hanya hasil pencarian yang diproses secara *in-memory*.

Struktur Basis Data ini memiliki batasan dan relasi yang jelas, terdiri dari dua tabel utama:

- **Tabel ApplicantProfile:** Menyimpan informasi pribadi pelamar seperti nama, tanggal lahir, alamat, dan nomor telepon.
- **Tabel ApplicationDetail:** Menyimpan detail spesifik lamaran, termasuk `application_role` dan `cv_path`. Atribut `cv_path` ini sangat krusial karena berfungsi sebagai penghubung (*pointer*) antara data terstruktur di basis data dengan lokasi file CV fisik di direktori `data/`.

Relasi antara kedua tabel ini didefinisikan sebagai *one-to-many*, di mana satu `ApplicantProfile` dapat memiliki banyak `ApplicationDetail`. Relasi ini memungkinkan seorang pelamar untuk melamar beberapa posisi berbeda dengan CV yang mungkin sudah disesuaikan.

### 3.4. Pemetaan menjadi elemen-elemen KMP dan Boyer-Moore

Setelah mendefinisikan objek-objek dasar seperti Teks CV dan Kata Kunci, langkah selanjutnya adalah memetakan objek-objek tersebut ke dalam elemen-elemen formal yang digunakan oleh algoritma pencocokan string, yaitu Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM). Kedua algoritma ini dirancang untuk mencari keberadaan sebuah "pola" dalam suatu "teks" dengan efisien, baik dalam hal waktu maupun memori.

### Pemetaan Elemen pada Sistem ATS:

- **Teks (Text):** Dalam konteks sistem ini, yang dimaksud dengan “Teks” adalah objek *Teks CV*. Ini adalah representasi string panjang hasil ekstraksi dari file CV berformat PDF. Satu CV akan direpresentasikan sebagai satu teks panjang yang tidak terstruktur.
- **Pola (Pattern):** Objek “Kata Kunci” yang dimasukkan oleh pengguna berperan sebagai “pola” yang akan dicocokkan. Pengguna dapat memasukkan beberapa kata kunci sekaligus (misalnya “React, Express”), yang masing-masing akan dianggap sebagai pola tersendiri yang dicocokkan satu per satu ke dalam Teks CV.

### Alur Pencocokan:

1. **Inisialisasi:** Sistem akan menginisialisasi algoritma pencocokan yang dipilih pengguna, baik KMP maupun BM.
2. **Pemindaian:** Setiap kata kunci akan dicocokkan dengan teks CV secara menyeluruh, dari awal hingga akhir, untuk mendeteksi semua kemunculan pola di dalam teks.
3. **Hasil:** Keluaran dari proses ini adalah frekuensi kemunculan masing-masing kata kunci dalam CV. Informasi ini akan digunakan untuk menilai relevansi dokumen tersebut dan menyusun peringkat hasil pencarian, agar CV yang paling sesuai ditampilkan terlebih dahulu.

**Contoh:** Jika pengguna memasukkan kata kunci “React, Express”, maka algoritma akan menjalankan proses pencarian dua kali: pertama untuk mencari semua kemunculan “React” dalam Teks CV, dan kedua untuk “Express”. Hasil pencocokan ini selanjutnya akan digunakan sebagai indikator utama relevansi CV terhadap kriteria pengguna.

### 3.5. Algoritma Penyelesaian Masalah

Secara garis besar dari abstraksi yang telah dilakukan, *keyword* akan dilakukan proses pencocokan dengan hasil ekstraksi CV *applicant* sehingga menemukan *exact string match* ataupun *fuzzy match* jika tidak ditemukan satupun *exact string match*.

Adapun tahapan algoritma pencocokan *string* yang diimplementasikan adalah sebagai berikut.

1. Inisialisai *function-function* ataupun pra-pemrosesan yang dibutuhkan untuk algoritma yang dipilih, seperti struktur data *trie* untuk algoritma *bonus Aho-Corasick*
2. Lakukan perbandingan karakter dengan algoritma yang dipilih
3. Jika terjadi ketidakcocokan, penanganan karakter *pattern* akan menyesuaikan algoritma yang dipilih. Untuk algoritma KMP dan BM akan melakukan pergeseran, sedangkan algoritma Aho-Corasick akan menandai *state pattern*
4. Jika terjadi kecocokan, maka *keyword* akan ditampung menjadi *map results* dengan *key* dari *map*, yaitu *keyword* akan menampung *occurrence*-nya sebagai *value*

5. Ulangi proses perbandingan (langkah 2), penanganan ketidakcocokan (langkah 3), dan pencatatan temuan (langkah 4) hingga seluruh `text` selesai ditelusuri.
6. Jika terdapat *keyword* yang tidak ditemukan *exact string match*, maka akan dilakukan *fuzzy matching* dengan algoritma Levenshtein yang akan menghitung similaritas *keyword* dengan tiap kata yang ada pada *text*.
7. Kembalikan total `count` dan indeks `found` sebagai hasil akhir.

### 3.6. Arsitektur Aplikasi CV Analyzer

Arsitektur sistem aplikasi CV Analyzer merupakan kerangka yang mengatur hubungan antar-komponen dalam aplikasi desktop, mulai dari interaksi pengguna hingga pengelolaan data CV. Pada bab ini, akan dijelaskan secara komprehensif arsitektur CV Analyzer yang dibangun menggunakan Python dengan PyQt6 pada sisi *frontend* dan MySQL sebagai basis data.

#### 3.6.1 Gambaran Umum Arsitektur

Secara keseluruhan, sistem dibagi menjadi tiga komponen utama: aplikasi desktop pengguna, server basis data MySQL, dan phpMyAdmin untuk manajemen basis data. Ketika pengguna membuka aplikasi, antarmuka grafis yang dibangun dengan PyQt6 akan ditampilkan. Aplikasi ini akan berinteraksi langsung dengan basis data MySQL untuk menyimpan dan mengambil data CV. MySQL dan phpMyAdmin di-*containerized* menggunakan Docker untuk memudahkan proses *deployment*.

#### 3.6.2 Frontend

Untuk sistem *frontend*, aplikasi CV Analyzer dibangun menggunakan Python dengan *library* PyQt6. PyQt6 dipilih karena menyediakan *binding* Python untuk *framework* GUI Qt yang *powerful* dan fleksibel, memungkinkan pembuatan antarmuka pengguna yang modern dan responsif. Meskipun ini adalah aplikasi desktop, PyQt6 memungkinkan integrasi yang mulus dengan fungsionalitas *backend* Python untuk pemrosesan CV. Penggunaan PyQt6 memudahkan pembuatan elemen-elemen UI seperti tombol, *input field*, tabel, dan grafik, serta mengelola interaksi pengguna dengan aplikasi.

#### 3.6.3 Interaksi Frontend-Controller dan Basis Data

Interaksi utama antara *frontend* (PyQt6) dan "*backend*" (*controller* dan *manager* pada Python) dengan melalui manipulasi data dan komunikasi dengan basis data MySQL. Ketika pengguna mengunggah atau memproses CV melalui antarmuka PyQt6, logika Python akan mem-*parsing* CV tersebut untuk memperoleh informasi *applicant*.

Komunikasi ini bersifat langsung dari aplikasi desktop ke server basis data MySQL, mencakup permintaan pembacaan data CV dari MySQL untuk ditampilkan di UI, penyimpanan data CV yang baru diproses ke dalam tabel MySQL, pembaruan informasi CV yang sudah ada di basis data, dan penghapusan data CV dari basis data.

#### 3.6.4 Dockerization

Untuk memudahkan *deployment* dan menjamin konsistensi lingkungan untuk basis data, MySQL dan phpMyAdmin di-*containerize* menggunakan Docker. Setiap komponen dibungkus dalam *image* Docker terpisah, memastikan dependensi dan konfigurasi spesifik dapat diisolasi. MySQL Container menyediakan lingkungan basis data yang konsisten dan terisolasi, memastikan bahwa aplikasi desktop dapat terhubung ke *instance* MySQL yang sudah dikonfigurasi dengan benar tanpa masalah beda versi.

Sementara itu, phpMyAdmin Container menyediakan antarmuka *web* yang mudah digunakan untuk mengelola basis data MySQL, seperti melihat tabel dan memberi *query* yang sangat berguna selama fase pengembangan dan *debugging*. Penggunaan `docker-compose` digunakan untuk mengorkestrasikan *image* Docker dari MySQL dan phpMyAdmin, memudahkan proses *setup* dan *management* kedua layanan tersebut.

Dengan pendekatan containerization, jadinya tidak perlu khawatir perbedaan versi atau dependensi pada mesin lokal. Aplikasi desktop tetap berjalan secara *native*, cukup terhubung ke *database database* database yang sudah ada sehingga *deployment* menjadi enak dan nyaman-nyaman.

## Bab IV

### Implementasi dan Pengujian

#### 4.1. Implementasi Program dengan PyQt6, Python, dan MySQL

Aplikasi IngfoLoker dikembangkan sepenuhnya menggunakan bahasa pemrograman Python dengan pustaka PyQt6 untuk menyusun antarmuka grafis (GUI). Koordinasi antara GUI, algoritma, dan penyimpanan data dikendalikan oleh `MainController` yang menghubungkan input pengguna dengan eksekusi algoritma dan pemanggilan basis data.

Data pelamar dan metadata CV disimpan dalam server MySQL dan dikelola melalui `phpMyAdmin` sebagai antarmuka web. Kedua layanan ini dijalankan di dalam kontainer Docker—`mysql:latest` untuk database dan `phpmyadmin` sebagai plugin administrasi—agar lingkungan pengembangan dan produksi selalu konsisten. Untuk koneksi ke database, digunakan pustaka `PyMySQL`, sedangkan `python-dotenv` memfasilitasi konfigurasi variabel lingkungan.

##### 4.1.1 Algoritma Pencocokan *String*

*File: kmp.py*

```
class KMP:
    def __init__(self, text: str, pattern: str):
        self.text = text
        self.pattern = pattern
        self.border_function = []

    def generate_prefixes(self, text):
        prefixes = []
        for i in range(1, len(text)):
            prefixes.append(text[:i])
        return prefixes

    def generate_suffixes(self, text):
        suffixes = []
        for i in range(len(text) - 1, 0, -1):
            suffixes.append(text[i:])
        return suffixes

    def search_match(self, pattern, idx) -> int:
        text = pattern[:idx]
        suffixes = self.generate_suffixes(text)
        prefixes = self.generate_prefixes(text)
        for i in range(len(suffixes) - 1, -1, -1):
            if suffixes[i] == prefixes[i]:
                return i + 1
        return 0

    def generate_border_function(self):
        self.border_function = []
        for i in range(1, len(self.pattern)):
            self.border_function.append(
                self.search_match(self.pattern, i)
            )
```



```

        ↪ self.border_function.append(self.search_match(self.pattern,
        ↪ i))
    self.border_function.append(0)

def kmp(self):
    if len(self.text) < len(self.pattern):
        return 0, -1

    i = 0
    count = 0
    found = -1

    while i < len(self.text):
        j = 0
        pergeseran = 0

        while j < len(self.pattern):
            pos = i + j
            if pos >= len(self.text) or self.text[pos] !=
            ↪ self.pattern[j]:
                ↪ self.border_function[j]
                j = self.border_function[j]
                if j != 0:
                    pergeseran = len(self.pattern) - j
                    break
            j += 1

        if pergeseran <= 0:
            pergeseran = 1

        if j == len(self.pattern):
            found = i
            count += 1
            pergeseran = len(self.pattern)

        i += pergeseran

    return count, found

def search(self):
    self.generate_border_function()
    return self.kmp()

```

Kelas KMP menyimpan dua atribut utama—`text` dan `pattern`—serta array `border_function` (LPS) yang akan diisi oleh metode `generate_border_function`. Dua metode pendukung, `generate_prefixes` dan `generate_suffixes`, menghasilkan daftar awalan dan sufiks dari setiap substring pola, yang kemudian digunakan oleh `search_match` untuk menentukan panjang awalan terpanjang yang juga merupakan sufiks pada setiap posisi.

Metode `generate_border_function` mengiterasi tiap indeks pola, memanggil `search_match` untuk menghitung nilai LPS (longest proper prefix which is also suffix), dan membangun array `border_function` yang memandu pergeseran saat mismatch. Dengan demikian, perhitungan

awal memastikan bahwa pada saat pencocokan, kita dapat melompati bagian pola yang telah teruji tanpa harus kembali ke awal.

Fungsi `kmp` melaksanakan algoritma pencocokan utama dalam dua lapis loop: loop luar bergerak di sepanjang teks (indeks  $i$ ), sedangkan loop dalam membandingkan karakter pola dengan teks dari posisi  $i$ , memanfaatkan `border_function` untuk menentukan pergeseran (variabel *pergeseran*) saat terjadi mismatch. Jika seluruh pola cocok  $j = \text{len}(\text{pattern})$ , penghitung kemunculan *count* bertambah dan posisi temuan pertama *found* disimpan, kemudian pola digeser utuh.

Metode `search` mengorkestrasi keseluruhan dengan pertama-tama membangun `border_function`, lalu menjalankan `kmp` untuk mengembalikan sepasang nilai: jumlah total kemunculan pola dalam teks dan indeks kemunculan pertama. Kompleksitas waktu keseluruhan adalah  $O(N + M)$ , dengan  $N$  panjang teks dan  $M$  panjang pola, sehingga cocok untuk pencarian efisien pada teks berukuran besar.

File: `bm.py`

```
class BoyerMoore:
    def __init__(self, text: str, pattern: str):
        self.text = text
        self.pattern = pattern
        self.last_occurence = {}

    def find_keys(self):
        keys = []
        for i in range(len(self.text)):
            if self.text[i] not in keys:
                keys.append(self.text[i])
        return keys

    def generate_last_occurence(self, keys):
        self.last_occurence = {}
        for key in keys:
            self.last_occurence[key] = -1
        for i in range(len(self.pattern)):
            if self.pattern[i] in keys:
                self.last_occurence[self.pattern[i]] = i

    def boyer_moore(self):
        if len(self.text) < len(self.pattern):
            return 0, -1

        i = 0
        count = 0
        found_at_pos = -1

        while i + len(self.pattern) <= len(self.text):
            j = len(self.pattern) - 1

            while j >= 0:
                pos = i + j
                if self.text[pos] != self.pattern[j]:
```

```

        to_be_aligned =
        ↪ self.last_occurence.get(self.text[pos], -1)
        if to_be_aligned == -1:
            pergeseran = len(self.pattern)
        elif to_be_aligned < j:
            pergeseran = j - to_be_aligned
        else:
            pergeseran = 1
        break
    j -= 1

    if j < 0:
        found_at_pos = i
        count += 1
        pergeseran = len(self.pattern)

    i += pergeseran

    return count, found_at_pos

def search(self):
    keys = self.find_keys()
    self.generate_last_occurence(keys)
    return self.boyer_moore()

```

Kelas BoyerMoore mengelola atribut `text` dan `pattern` serta kamus `last_occurence` yang dibangun oleh metode `generate_last_occurence`. Pertama, `find_keys` mengumpulkan karakter unik dari teks, kemudian `generate_last_occurence` menginisialisasi semua karakter ke  $-1$  dan mengisi posisi terakhir kemunculan setiap karakter pola, yang menjadi kunci pergeseran saat mismatch.

Pada metode `boyer_moore`, pencocokan dimulai dengan membandingkan karakter pola dari indeks  $j = \text{len}(\text{pattern}) - 1$  ke karakter teks di posisi  $i + j$ . Jika terjadi mismatch pada `text[pos]` dan `pattern[j]`, nilai `last_occurence[text[pos]]` ( $k$ ) digunakan untuk menghitung pergeseran  $s$ : jika  $k = -1$ ,  $s = \text{len}(\text{pattern})$ ; jika  $k < j$ ,  $s = j - k$ ; jika  $k \geq j$ ,  $s = 1$ . Pendekatan ini memungkinkan loncatan cepat melewati segmen teks yang tidak relevan.

Ketika seluruh pola cocok ( $j < 0$ ), penghitung kemunculan (*count*) bertambah dan posisi pertama ditemukan (*found\_at\_pos* =  $i$ ), lalu pola digeser utuh ( $s = \text{len}(\text{pattern})$ ). Loop utama kemudian menambahkan  $s$  ke  $i$  dan melanjutkan hingga ujung teks.

Metode `search` mengorkestrasi eksekusi dengan memanggil `find_keys` dan `generate_last_occurence` untuk menyiapkan tabel skip, lalu menjalankan `boyer_moore` untuk mengembalikan jumlah kemunculan pola dan indeks kemunculan pertama. Kompleksitas rata-rata algoritma bersifat sub-linear, yaitu sekitar  $O(n/m)$ , namun dalam kasus terburuk menjadi  $O(n \times m)$ , dengan  $n = \text{len}(\text{text})$  dan  $m = \text{len}(\text{pattern})$ .

```
from collections import defaultdict
import string

class AhoCorasick:
    def __init__(self, keywords):
        self.words = [word.lower() for word in keywords]
        self.alphabet = string.ascii_lowercase + string.digits +
            ↪ string.punctuation
        self.alpha_size = len(self.alphabet)
        self.char_to_index = {ch: i for i, ch in
            ↪ enumerate(self.alphabet)}
        self.max_states = sum(len(w) for w in self.words) + 1
        self.goto = [[-1] * self.alpha_size for _ in
            ↪ range(self.max_states)]
        self.out = [0] * self.max_states
        self.fail = [-1] * self.max_states
        self.states_count = self.__build_matching_machine()

    def __build_matching_machine(self):
        states = 1
        for idx, word in enumerate(self.words):
            s = 0
            for ch in word:
                i = self.char_to_index.get(ch)
                if i is None:
                    continue
                if self.goto[s][i] == -1:
                    self.goto[s][i] = states
                    states += 1
                s = self.goto[s][i]
            self.out[s] |= (1 << idx)

        for i in range(self.alpha_size):
            if self.goto[0][i] == -1:
                self.goto[0][i] = 0

        queue = []
        for i in range(self.alpha_size):
            nxt = self.goto[0][i]
            if nxt != 0:
                self.fail[nxt] = 0
                queue.append(nxt)

        while queue:
            r = queue.pop(0)
            for i in range(self.alpha_size):
                nxt = self.goto[r][i]
                if nxt != -1:
                    queue.append(nxt)
                    f = self.fail[r]
                    while self.goto[f][i] == -1:
                        f = self.fail[f]
                    self.fail[nxt] = self.goto[f][i]
                    self.out[nxt] |= self.out[self.fail[nxt]]
        return states
```

```

def __find_next_state(self, s, ch):
    i = self.char_to_index.get(ch)
    if i is None:
        return s
    while self.goto[s][i] == -1:
        s = self.fail[s]
    return self.goto[s][i]

def search(self, text):
    s = 0
    hits = defaultdict(int)
    last_end = -1

    for i, raw_ch in enumerate(text):
        ch = raw_ch.lower()
        s = self.__find_next_state(s, ch)
        if self.out[s] != 0:
            for idx in range(len(self.words)):
                if self.out[s] & (1 << idx):
                    w = self.words[idx]
                    start = i - len(w) + 1
                    if start > last_end:
                        hits[w] += 1
                        last_end = i

    return hits

```

Kelas `AhoCorasick` menggabungkan trie dan automaton ber-fail-link untuk pencarian banyak pola secara efisien. Konstruktor menyimpan pola dalam huruf kecil, membangun alfabet (huruf, digit, tanda baca), dan menyiapkan struktur `goto`, `out` (bitmask pola akhir), serta `fail` (tautan fallback). Mesin dibangun lewat `__build_matching_machine`, yang pertama-tama memasukkan setiap karakter pola ke dalam trie dan menandai status akhir pola, kemudian melakukan BFS untuk menetapkan tautan `fail` sehingga tiap status mewarisi kecocokan pola dari fallback-nya.

Fungsi `__find_next_state` menangani transisi karakter demi karakter: jika transisi langsung tidak ada (`-1`), ia mengikuti `fail` links hingga menemukan status valid, atau kembali ke status awal. Ini memastikan pencarian tetap linear terhadap panjang teks meskipun terjadi mismatch.

Metode `search` memindai teks masukan satu per satu, memperbarui status automaton dan memeriksa `out[s]`. Saat `out[s]` tidak nol, pola terdeteksi—hitungannya disimpan di `hits`—dan `last_end` menjaga agar kecocokan tidak tumpang tindih. Hasil akhir adalah peta pola ke jumlah kemunculannya, yang diperoleh dalam satu lintasan  $O(N + \Sigma|\text{pattern}|)$  terhadap teks.

File: levenshtein.py

```
class Levenshtein:
    def __init__(self, text: str, pattern: str):
        self.text = text
        self.pattern = pattern
        self.mat = []

    def lev(self, i: int, j: int):
        self.mat[i][j] = min(
            self.mat[i - 1][j] + 1,
            self.mat[i][j - 1] + 1,
            self.mat[i - 1][j - 1] + (1 if self.pattern[j - 1] !=
            ↪ self.text[i - 1] else 0)
        )

    def compute_lev_distance(self, a: str, b: str) -> int:
        self.text = b
        self.pattern = a
        self.mat = [[0 for _ in range(len(a) + 1)] for _ in
        ↪ range(len(b) + 1)]

        for col in range(len(a) + 1):
            self.mat[0][col] = col
        for row in range(len(b) + 1):
            self.mat[row][0] = row

        for row in range(1, len(b) + 1):
            for col in range(1, len(a) + 1):
                self.lev(row, col)

        return self.mat[len(b)][len(a)]

    def compute_similarity(self, threshold: float = 80.0):
        pattern_words = self.pattern.split()
        text_words = self.text.split()
        window_size = len(pattern_words)
        matches = []

        for i in range(len(text_words) - window_size + 1):
            window_str = ' '.join(text_words[i:i + window_size])
            distance = self.compute_lev_distance(self.pattern,
            ↪ window_str)
            max_len = max(len(self.pattern), len(window_str))
            similarity = (1 - distance / max_len) * 100

            if similarity >= threshold:
                char_start = len(' '.join(text_words[:i]))
                if i > 0:
                    char_start += 1
                matches.append((char_start, window_str,
                ↪ round(similarity, 2)))

        return matches

    def search_fuzzy_matches(self, threshold: float = 80.0):
        match_details = self.compute_similarity(threshold)
```

```
(base) PS D:\ITB\Semester 4\Stima\Tubes3_IngfoLoker> uv run -m src.algo.levenshtein
Best threshold: 61% (cross-validated F1=0.893)
```

Gambar 7: Threshold untuk *fuzzy matching*

```
total_count = len(match_details)

matched_strings_dict = {}
for _, window_str, _ in match_details:
    matched_strings_dict[window_str] =
        ↪ matched_strings_dict.get(window_str, 0) + 1
return total_count > 0, total_count, matched_strings_dict
```

Kelas `Levenshtein` menerapkan algoritma jarak edit menggunakan pemrograman dinamis. Matriks `mat` berukuran  $(|b| + 1) \times (|a| + 1)$  dibangun dalam `compute_lev_distance`, dengan inisialisasi baris dan kolom pertama sesuai urutan indeks. Setiap sel dihitung oleh metode `lev` yang memilih operasi termurah—hapus, sisip, atau ganti—sehingga `mat[i][j]` menyimpan jarak Levenshtein antara prefix pertama  $i$  karakter dari teks dan prefix  $j$  karakter dari pola. Hasil akhir jarak adalah `mat[len(b)][len(a)]`.

Metode `compute_similarity` membagi teks dan pola menjadi kata, mengambil jendela kata sebanyak panjang pola, lalu menghitung jarak edit tiap jendela. Persentase kesamaan dihitung sebagai  $(1 - \text{distance} / \max(|\text{pattern}|, |\text{window}|)) \times 100$ . Jika melewati ambang *threshold*, posisi karakter awal, substring yang cocok, dan nilai kesamaan dibubuhi ke daftar `matches`.

Untuk perhitungan threshold, kami terinspirasi dari *hyperparameter tuning* pada model machine learning. Kami membuat dataset secara manual dengan melabeling nilai keyword hasil dan keyword prediksinya. Selanjutnya data dibagi menjadi beberapa bagian (folds) untuk validasi silang agar tidak terlalu *overfit*. Selanjutnya, sistem menguji setiap nilai threshold dari 50 hingga 100. Untuk setiap threshold, dilakukan prediksi kecocokan antara teks dan pola menggunakan algoritma Levenshtein, lalu dibandingkan dengan label kebenaran yang telah ditentukan sebelumnya. Hasil prediksi ini digunakan untuk menghitung nilai F1-score secara manual, yang mencerminkan keseimbangan antara presisi dan recall. Setelah semua threshold diuji, nilai threshold yang menghasilkan rata-rata F1-score tertinggi dari seluruh fold dianggap sebagai threshold terbaik. Berikut merupakan hasil threshold yang didapat:

### 4.1.2 Algoritma *Encryption*

File: FF3.py

```
import logging
import math
import string

from typing import Dict, Union, Optional
from src.crypto.AES import AESCipher

logger = logging.getLogger(__name__)

class FF3Cipher:

    # FF3 Algorithm Constants
    NUM_ROUNDS = 8
    BLOCK_SIZE = 16
    TWEAK_LEN = 8
    TWEAK_LEN_NEW = 7
    HALF_TWEAK_LEN = TWEAK_LEN // 2

    # Domain and Radix Limits
    DOMAIN_MIN = 1_000_000
    RADIX_MAX = 256

    # Base62 Alphabet
    BASE62 = string.digits + string.ascii_lowercase +
        ↪ string.ascii_uppercase
    BASE62_LEN = len(BASE62)

    # Database field type configurations
    FIELD_CONFIGS = {
        'name': {
            'radix': 62,      # Letters only (a-z, A-Z) = 26 + 26 =
            ↪ 52
            'min_len': 1,    # Allow short names like "J"
            'max_len': 50,   # VARCHAR(50) limit
            'alphabet': string.ascii_lowercase +
            ↪ string.ascii_uppercase + string.digits
        },
        'phone': {
            'radix': 15,     # Digits + phone symbols: 0-9, +, -,
            ↪ space, ( )
            'min_len': 6,    # Security minimum
            'max_len': 20,   # VARCHAR(20) limit
            'alphabet': string.digits + '+- ()' # International
            ↪ phone format
        },
        'date': {
            'radix': 11,     # Digits + dash for date format
            ↪ (YYYY-MM-DD)
            'min_len': 8,    # Minimum date length
            'max_len': 10,   # VARCHAR(10) limit
            'alphabet': string.digits + '-' # Digits and dash
        },
        'address': {
```



```

        'radix': 72,
        'min_len': 1,
        'max_len': 255,
        'alphabet': string.digits + string.ascii_lowercase +
        ↪ string.ascii_uppercase + " .,-'#/()&"
    }
}

def __init__(self, key: str, tweak: str, radix: int = 10) ->
    ↪ None:
    keybytes = bytes.fromhex(key)
    self.key = key
    self.tweak = tweak
    self.default_radix = radix

    if radix <= self.BASE62_LEN:
        self.default_alphabet = self.BASE62[0:radix]
    else:
        self.default_alphabet = None

    self.default_min_len = math.ceil(math.log(self.DOMAIN_MIN)
    ↪ / math.log(radix))
    self.default_max_len = 2 * math.floor(96/math.log2(radix))

    klen = len(keybytes)

    if klen not in (16, 24, 32):
        raise ValueError(f'key length is {klen} but must be
        ↪ 128, 192, or 256 bits')

    if (radix < 2) or (radix > self.RADIX_MAX):
        raise ValueError(f"radix must be between 2 and
        ↪ {self.RADIX_MAX}, inclusive")

    if (self.default_min_len < 2) or (self.default_max_len <
    ↪ self.default_min_len):
        raise ValueError("Default minLen or maxLen invalid,
        ↪ adjust your radix")

    self.aesCipher = AESCipher(self._reverse_string(keybytes),
    ↪ AESCipher.MODE_ECB)

def _reverse_string(self, txt: Union[str, bytes]) -> Union[str,
    ↪ bytes]:
    return txt[::-1]

def _get_field_config(self, field_type: str) -> Dict:
    if field_type not in self.FIELD_CONFIGS:
        raise ValueError(f"Unknown field_type: {field_type}. "
        ↪ f"Available:
        ↪ {list(self.FIELD_CONFIGS.keys())}")

    return self.FIELD_CONFIGS[field_type]

def _validate_field_input(self, plaintext: str, field_type:
    ↪ str) -> None:
    config = self._get_field_config(field_type)

```

```

if len(plaintext) < config['min_len'] or len(plaintext) >
↳ config['max_len']:
    raise ValueError(f"Input length {len(plaintext)} is not
↳ within "
                                f"min {config['min_len']} and max
                                ↳ {config['max_len']} "
                                f"bounds for field_type '{field_type}''")

if config['alphabet']:
    invalid_chars = set(plaintext) -
↳ set(config['alphabet'])
    if invalid_chars:
        raise ValueError(f"Invalid characters
↳ {invalid_chars} for field_type '{field_type}'".
↳ "
                                f"Allowed: {config['alphabet']}")

def _calculate_p(self, i: int, alphabet: str, W: bytes, B: str)
↳ -> bytearray:
    # P is always 16 bytes
    P = bytearray(self.BLOCK_SIZE)

    # Calculate P by XORing W, i into the first 4 bytes of P
    P[0] = W[0]
    P[1] = W[1]
    P[2] = W[2]
    P[3] = W[3] ^ int(i)

    # The remaining 12 bytes of P are for rev(B) with padding
    val = self._decode_int_r(B, alphabet)
    try:
        BBytes = val.to_bytes(12, "big")
    except OverflowError:

        # Use modulo arithmetic to fit large values in 12 bytes
        max_12_byte_int = (1 << 96) - 1 # 2^96 - 1
        val = val % (max_12_byte_int + 1)
        BBytes = val.to_bytes(12, "big")

    logger.debug(f"B: {B} val: {val} BBytes: {BBytes.hex()}")

    P[self.BLOCK_SIZE - len(BBytes):] = BBytes
    logger.debug(f"[round: {i}] P: {P.hex()} W: {W.hex()} ")

    return P

def _calculate_tweak64_ff3_1(self, tweak56: bytes) ->
↳ bytearray:
    tweak64 = bytearray(8)
    tweak64[0] = tweak56[0]
    tweak64[1] = tweak56[1]
    tweak64[2] = tweak56[2]
    tweak64[3] = (tweak56[3] & 0xF0)
    tweak64[4] = tweak56[4]
    tweak64[5] = tweak56[5]
    tweak64[6] = tweak56[6]

```

```

tweak64[7] = ((tweak56[3] & 0x0F) << 4)

return tweak64

def _encode_int_r(self, n: int, alphabet: str, length: int = 0)
↳ -> str:
    base = len(alphabet)
    if (base > self.RADIX_MAX):
        raise ValueError(f"Base {base} is outside range of
↳ supported radix "
                        f"2..{self.RADIX_MAX}")

    x = ''
    while n >= base:
        n, b = divmod(n, base)
        x += alphabet[b]
    x += alphabet[n]

    if len(x) < length:
        x = x.ljust(length, alphabet[0])

    return x

def _decode_int_r(self, astring: str, alphabet: str) -> int:
    strlen = len(astring)
    base = len(alphabet)

    num = 0
    idx = 0

    try:
        for char in reversed(astring):
            power = (strlen - (idx + 1))
            num += alphabet.index(char) * (base ** power)
            idx += 1

    except ValueError:
        raise ValueError(f'char {char} not found in alphabet
↳ {alphabet}')

    return num

def encrypt(self, plaintext: str, field_type: Optional[str] =
↳ None) -> str:
    if field_type:
        config = self._get_field_config(field_type)
        self._validate_field_input(plaintext, field_type)
        alphabet = config['alphabet'] if config['alphabet']
↳ else self.BASE62[0:config['radix']]
        return self._encrypt_with_config(plaintext, self.tweak,
↳ config['radix'], alphabet)

    else:
        return self._encrypt_with_config(plaintext, self.tweak,
↳ self.default_radix, self.default_alphabet)

```

```

def decrypt(self, ciphertext: str, field_type: Optional[str] =
↳ None) -> str:
    if field_type:
        config = self._get_field_config(field_type)
        alphabet = config['alphabet'] if config['alphabet']
↳ else self.BASE62[0:config['radix']]

        return self._decrypt_with_config(ciphertext,
↳ self.tweak, config['radix'], alphabet,
                                                config['min_len'],
                                                ↳ config['max_len'])

    else:
        return self._decrypt_with_config(ciphertext,
↳ self.tweak, self.default_radix,
                                                self.default_alphabet,
↳ self.default_min_len,
↳ self.default_max_len)

def _encrypt_with_config(self, plaintext: str, tweak: str,
↳ radix: int, alphabet: str) -> str:
    tweakBytes = bytes.fromhex(tweak)
    n = len(plaintext)

    if len(tweakBytes) not in [self.TWEAK_LEN,
↳ self.TWEAK_LEN_NEW]:
        raise ValueError(f"tweak length {len(tweakBytes)}
↳ invalid: tweak must be 56"
                                f" or 64 bits")

    # Calculate split point
    u = math.ceil(n / 2)
    v = n - u

    # Split the message
    A = plaintext[:u]
    B = plaintext[u:]

    if len(tweakBytes) == self.TWEAK_LEN_NEW:
        # FF3-1
        tweakBytes = self._calculate_tweak64_ff3_1(tweakBytes)

    Tl = tweakBytes[:self.HALF_TWEAK_LEN]
    Tr = tweakBytes[self.HALF_TWEAK_LEN:]
    logger.debug(f"Tweak: {tweak},
↳ tweakBytes:{tweakBytes.hex()}")

    # Pre-calculate the modulus since it's only one of 2 values
    modU = radix ** u
    modV = radix ** v
    logger.debug(f"modU: {modU} modV: {modV}")

    # Main Feistel Round, 8 times
    for i in range(self.NUM_ROUNDS):
        logger.debug(f"----- Round {i}")
        # Determine alternating Feistel round side
        if i % 2 == 0:
            m = u

```

```

        W = Tr
    else:
        m = v
        W = Tl

    # P is fixed-length 16 bytes
    P = self._calculate_p(i, alphabet, W, B)
    revP = self._reverse_string(P)

    S = self.aesCipher.encrypt(bytes(revP))
    S = self._reverse_string(S)
    logger.debug(f"S: {S.hex()}")

    y = int.from_bytes(S, byteorder='big')

    # Calculate c
    c = self._decode_int_r(A, alphabet)
    c = c + y

    if i % 2 == 0:
        c = c % modU
    else:
        c = c % modV

    logger.debug(f"m: {m} A: {A} c: {c} y: {y}")
    C = self._encode_int_r(c, alphabet, int(m))

    # Final steps
    A = B
    B = C

    logger.debug(f"A: {A} B: {B}")

    return A + B

def _decrypt_with_config(self, ciphertext: str, tweak: str,
    ↪ radix: int, alphabet: str, min_len: int, max_len: int) ->
    ↪ str:
    tweakBytes = bytes.fromhex(tweak)
    n = len(ciphertext)

    if (n < min_len) or (n > max_len):
        raise ValueError(f"message length {n} is not within min
    ↪ {min_len} and "
    ↪ f"max {max_len} bounds")

    if len(tweakBytes) not in [self.TWEAK_LEN,
    ↪ self.TWEAK_LEN_NEW]:
        raise ValueError(f"tweak length {len(tweakBytes)}
    ↪ invalid: tweak must be 8 "
    ↪ f"bytes, or 64 bits")

    # Calculate split point
    u = math.ceil(n/2)
    v = n - u

    # Split the message

```

```

A = ciphertext[:u]
B = ciphertext[u:]

if len(tweakBytes) == self.TWEAK_LEN_NEW:
    # FF3-1
    tweakBytes = self._calculate_tweak64_ff3_1(tweakBytes)

Tl = tweakBytes[:self.HALF_TWEAK_LEN]
Tr = tweakBytes[self.HALF_TWEAK_LEN:]
logger.debug(f"Tweak: {tweak},
↳ tweakBytes:{tweakBytes.hex()}")

# Pre-calculate the modulus since it's only one of 2 values
modU = radix ** u
modV = radix ** v
logger.debug(f"modU: {modU} modV: {modV}")

# Main Feistel Round, 8 times
for i in reversed(range(self.NUM_ROUNDS)):
    logger.debug(f"----- Round {i}")
    # Determine alternating Feistel round side
    if i % 2 == 0:
        m = u
        W = Tr
    else:
        m = v
        W = Tl

    # P is fixed-length 16 bytes
    P = self._calculate_p(i, alphabet, W, A)
    revP = self._reverse_string(P)

    S = self.aesCipher.encrypt(bytes(revP))
    S = self._reverse_string(S)
    logger.debug("S: ", S.hex())

    y = int.from_bytes(S, byteorder='big')

    # Calculate c
    c = self._decode_int_r(B, alphabet)
    c = c - y

    if i % 2 == 0:
        c = c % modU
    else:
        c = c % modV

    logger.debug(f"m: {m} B: {B} c: {c} y: {y}")
    C = self._encode_int_r(c, alphabet, int(m))

    # Final steps
    B = A
    A = C

    logger.debug(f"A: {A} B: {B}")

return A + B

```

```

@staticmethod
def withCustomAlphabet(key: str, tweak: str, alphabet: str) ->
    ↪ 'FF3Cipher':
    c = FF3Cipher(key, tweak, len(alphabet))
    c.default_alphabet = alphabet
    return c

```

Kelas `FF3Cipher` mengemas semua konstanta dan konfigurasi untuk algoritma FF3-1 (Format-Preserving Encryption) — termasuk jumlah ronde  $N = 8$ , panjang tweak, domain, dan alfabet default. Atribut `FIELD_CONFIGS` menyediakan parameter khusus per tipe data (nama, telepon, tanggal, alamat) seperti radix, batas panjang, dan alfabet yang diperbolehkan.

Konstruktor menerima kunci dalam format hex dan tweak, memvalidasi panjang kunci (16, 24, 32 byte) serta nilai radix ( $2 \leq \text{radix} \leq 256$ ). Ia menghitung `default_min_len` dan `default_max_len` berdasarkan `DOMAIN_MIN`, serta menyiapkan instansi `AESCipher` (mode ECB) dengan kunci yang dibalik (`_reverse_string`) guna mendukung FF3-1.

Metode `encrypt` (dan sebaliknya `decrypt`) membagi plaintext menjadi dua bagian  $A$  dan  $B$  dengan panjang  $u = \lceil n/2 \rceil$  dan  $v = n - u$ . Tweak dipecah menjadi dua setengah  $T_L$  dan  $T_R$ , dengan penanganan khusus jika panjang tweak baru (7 byte) untuk FF3-1.

Setiap ronde Feistel ke- $i$  membangun blok  $P$  16 byte melalui `_calculate_p`: empat byte pertama hasil XOR tweak dan indeks ronde, serta 12 byte sisanya adalah representasi big-endian dari nilai integer  $B$  yang di-decode dari alfabet. Blok  $P$  dibalik, dienkripsi dengan AES-ECB, lalu hasilnya ( $S$ ) dibalik kembali untuk memperoleh integer  $y$ .

Nilai antara  $c = \text{decode}(A) + y$  (atau  $c = \text{decode}(B) - y$  pada dekripsi) kemudian di-modulo dengan  $r^u$  atau  $r^v$  sesuai paritas ronde ( $r = \text{radix}$ ), dan di-encode kembali menjadi string panjang  $m$  via `_encode_int_r`. Akhirnya,  $A$  dan  $B$  ditukar untuk ronde berikutnya.

Setelah  $N$  ronde, ciphertext adalah concatenation dari  $A||B$ . Kompleksitas enkripsi per pesan panjang  $n$  dan  $N$  ronde adalah  $O(N \times n)$ , dengan setiap ronde melakukan satu enkripsi blok AES dan beberapa operasi aritmetika modular, cocok untuk aplikasi real-time pada data format-preserving.

File: AES.py

```
from typing import List

class AESCipher:

    RCON = [
        0x01, 0x02, 0x04, 0x08, 0x10,
        0x20, 0x40, 0x80, 0x1b, 0x36,
        0x6c, 0xd8, 0xab, 0x4d, 0x9a
    ]

    MODE_ECB = 1

    def __init__(self, key: bytes, mode: int) -> None:

        if mode != self.MODE_ECB:
            raise ValueError("Only ECB mode is implemented")

        if len(key) not in [16, 24, 32]:
            raise ValueError("Key must be 16, 24, or 32 bytes")

        self.key = key
        self.mode = mode

    def _sub_bytes(self, state: List[int]) -> None:
        for i in range(16):
            state[i] = self.SBOX[state[i]]

    def _shift_rows(self, state: List[int]) -> None:
        # Row 1: shift left by 1
        temp = state[1]
        state[1] = state[5]
        state[5] = state[9]
        state[9] = state[13]
        state[13] = temp

        # Row 2: shift left by 2
        temp1, temp2 = state[2], state[6]
        state[2] = state[10]
        state[6] = state[14]
        state[10] = temp1
        state[14] = temp2

        # Row 3: shift left by 3
        temp = state[15]
        state[15] = state[11]
        state[11] = state[7]
        state[7] = state[3]
        state[3] = temp

    def _gmul(self, a: int, b: int) -> int:
        p = 0

        for i in range(8):
            if b & 1:
                p ^= a
```



```

        hi_bit_set = a & 0x80
        a <<= 1
        if hi_bit_set:
            a ^= 0x1b
        b >>= 1

    return p & 0xff

def _mix_columns(self, state: List[int]) -> None:
    mix = [
        [0x02, 0x03, 0x01, 0x01],
        [0x01, 0x02, 0x03, 0x01],
        [0x01, 0x01, 0x02, 0x03],
        [0x03, 0x01, 0x01, 0x02]
    ]

    for col in range(4):
        col_start = col * 4
        column = [state[col_start], state[col_start+1],
                  state[col_start+2], state[col_start+3]]

        for row in range(4):
            result = 0
            for i in range(4):
                result ^= self._gmul(mix[row][i], column[i])
            state[col_start + row] = result

def _add_round_key(self, state: List[int], round_key:
    List[int]) -> None:
    for i in range(16):
        state[i] ^= round_key[i]

def _rot_word(self, word: List[int]) -> List[int]:
    return word[1:] + word[:1]

def _sub_word(self, word: List[int]) -> List[int]:
    return [self.SBOX[b] for b in word]

def _key_expansion(self, key: bytes) -> List[List[int]]:
    key_len = len(key)

    if key_len == 16:      # AES-128
        rounds = 10
        nk = 4
    elif key_len == 24:    # AES-192
        rounds = 12
        nk = 6
    elif key_len == 32:    # AES-256
        rounds = 14
        nk = 8
    else:
        raise ValueError("Invalid key length. Must be 16, 24,
                           or 32 bytes.")

    w = []

    # First nk words are the original key

```

```

        for i in range(nk):
            w.append([key[4*i], key[4*i+1], key[4*i+2],
                      ↪ key[4*i+3]])

        # Generate remaining words
        for i in range(nk, 4 * (rounds + 1)):
            temp = w[i-1][:]

            if i % nk == 0:
                temp = self._sub_word(self._rot_word(temp))
                temp[0] ^= self.RCON[i//nk - 1]
            elif nk > 6 and i % nk == 4:
                temp = self._sub_word(temp)

            w.append([w[i-nk][j] ^ temp[j] for j in range(4)])

        # Convert to round keys
        round_keys = []
        for round_num in range(rounds + 1):
            round_key = []
            for i in range(4):
                round_key.extend(w[round_num * 4 + i])
            round_keys.append(round_key)

        return round_keys

def _encrypt_block(self, plaintext_block: bytes) -> bytes:

    if len(plaintext_block) != 16:
        raise ValueError("Block must be exactly 16 bytes")

    state = list(plaintext_block)
    round_keys = self._key_expansion(self.key)
    rounds = len(round_keys) - 1

    # Initial round key addition
    self._add_round_key(state, round_keys[0])

    # Main rounds
    for round_num in range(1, rounds):
        self._sub_bytes(state)
        self._shift_rows(state)
        self._mix_columns(state)
        self._add_round_key(state, round_keys[round_num])

    # Final round (no MixColumns)
    self._sub_bytes(state)
    self._shift_rows(state)
    self._add_round_key(state, round_keys[rounds])

    return bytes(state)

def encrypt(self, plaintext: bytes) -> bytes:

    if len(plaintext) % 16 != 0:
        raise ValueError("Plaintext length must be multiple of
        ↪ 16 bytes for ECB mode")

```

```

ciphertext = b''

for i in range(0, len(plaintext), 16):
    block = plaintext[i:i+16]
    encrypted_block = self._encrypt_block(block)
    ciphertext += encrypted_block

return ciphertext

```

Kelas `AESCipher` menginisialisasi parameter utama—`key` (16, 24, atau 32 byte) dan mode ECB—serta konstanta `RCON` untuk *round constants*. Konstruktor memastikan hanya mode ECB yang diizinkan dan panjang kunci yang valid.

Metode `_sub_bytes` dan `_shift_rows` melaksanakan dua tahap substitusi dan perpindahan baris pada *state* 16 byte, sedangkan `_mix_columns` memanfaatkan perkalian di  $GF(2^8)$  melalui `_gmul` untuk mencampur kolom dengan matriks konstan. Langkah `_add_round_key` melakukan XOR antara *state* dan *round key*.

Ekspansi kunci pada `_key_expansion` menghasilkan `round_keys` sebanyak  $r + 1$  buah, di mana  $r$  tergantung pada panjang kunci (10, 12, atau 14). Proses ini memanggil `_rot_word`, `_sub_word`, dan menambahkan `RCON` untuk setiap word kelipatan  $n_k$ , sehingga membangun tabel kunci penuh yang mengamankan setiap ronde enkripsi.

Fungsi `_encrypt_block` mengenkripsi satu blok 16 byte dengan urutan:

1. Tambah kunci awal (`add_round_key`)
2.  $r - 1$  ronde utama: `sub_bytes`, `shift_rows`, `mix_columns`, `add_round_key`
3. Ronde akhir tanpa `mix_columns`: `sub_bytes`, `shift_rows`, `add_round_key`

Metode `encrypt` membagi *plaintext* menjadi blok-blok 16 byte, memanggil `_encrypt_block` untuk setiap blok, dan menggabungkan hasilnya menjadi *ciphertext*.

Dengan demikian, algoritma AES-ECB ini memiliki kompleksitas  $O(B \times r)$  untuk  $B$  blok dan  $r$  ronde, menawarkan keamanan dan kecepatan enkripsi yang terstandarisasi.

## 4.2. Alur Program

Aplikasi *CV Analyzer App* menyediakan fitur-fitur utama berikut:

- **Pilihan Algoritma Pencocokan String:**  
KMP, Boyer–Moore, atau Aho–Corasick sebagai algoritma opsi pengguna.
- **Top Matches:**  
Batasi jumlah hasil teratas yang ditampilkan pada kolom *Top Matches*. Beralih ke *Fuzzy Matching* jika tidak ada hasil *exact* setelah pencocokan.
- **Result Cards:**  
Setiap hasil direpresentasikan dalam kartu berisi nama *applicant*, jumlah kemunculan kata kunci, serta daftar frekuensi kata kunci.
- **Summary & View CV:**  
Tombol *Summary* menampilkan ringkasan temuan, dan *View CV* membuka file CV asli.

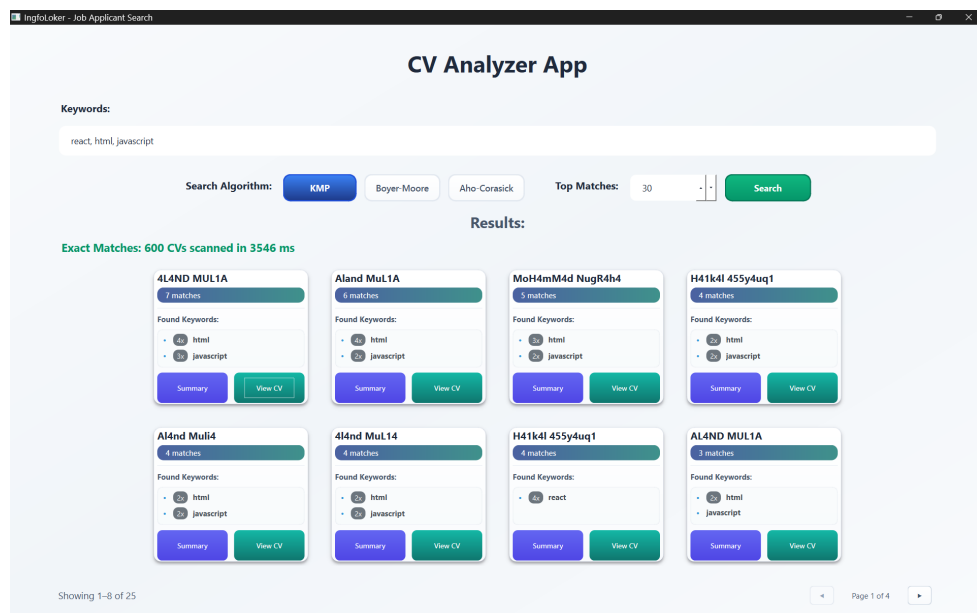
Berikut adalah tata cara penggunaan program:

1. Masukkan kata kunci pencarian pada kolom *Keywords* di bagian atas halaman (misalnya “Accounting”).
2. Pilih algoritma pencocokan pada opsi *Search Algorithm*.
3. Tentukan jumlah hasil teratas yang diinginkan pada kolom *Top Matches*.
4. Klik tombol *Search* untuk memulai proses.
5. Hasil pencarian akan muncul di area *Results* dengan ringkasan performa dan kartu-kartu hasil.
6. Gunakan tombol navigasi di bawah untuk berpindah halaman jika diperlukan.

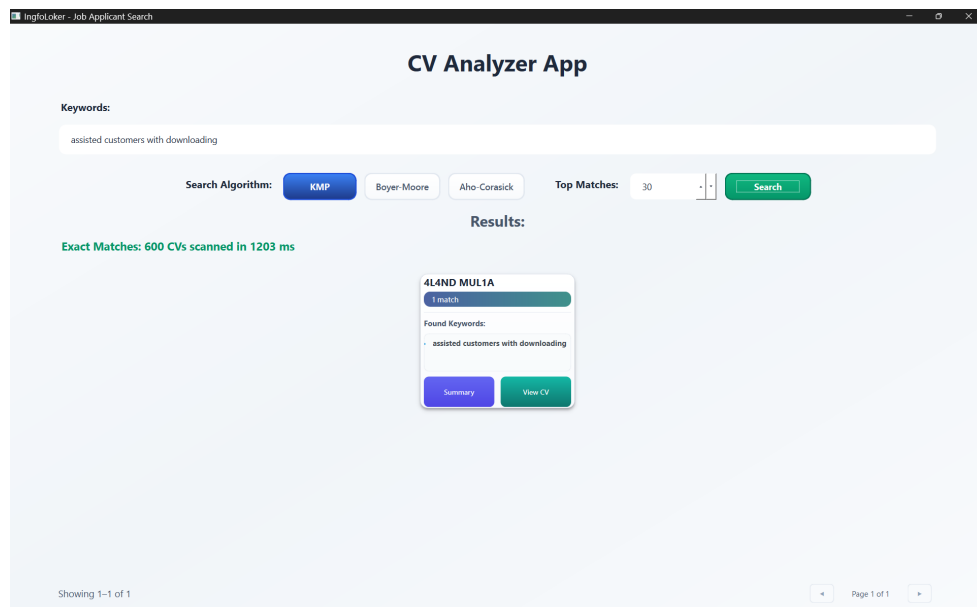
### 4.3. Eksperimen

Berikut hasil eksperimen dari berbagai tes yang diujikan, mulai *test-case* umum hingga *stress-case* untuk menilai performa program. Untuk setiap algoritma, akan ada empat test case.

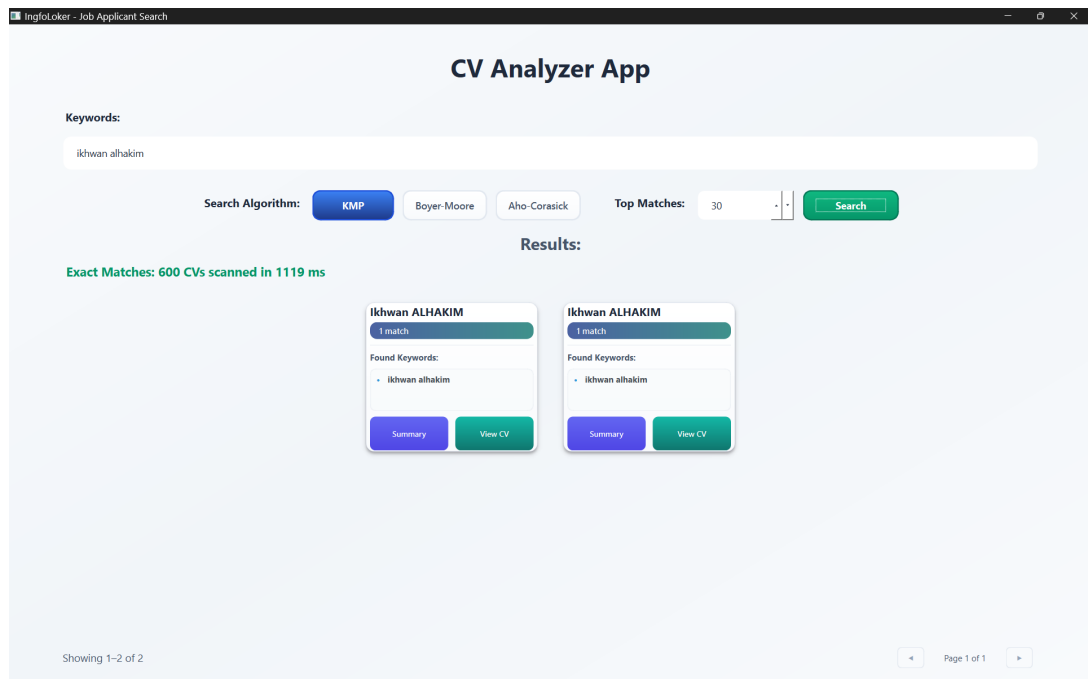
#### a. Knuth-Morris-Pratt



Gambar 8: Percobaan Algoritma KMP dengan tiga keywords

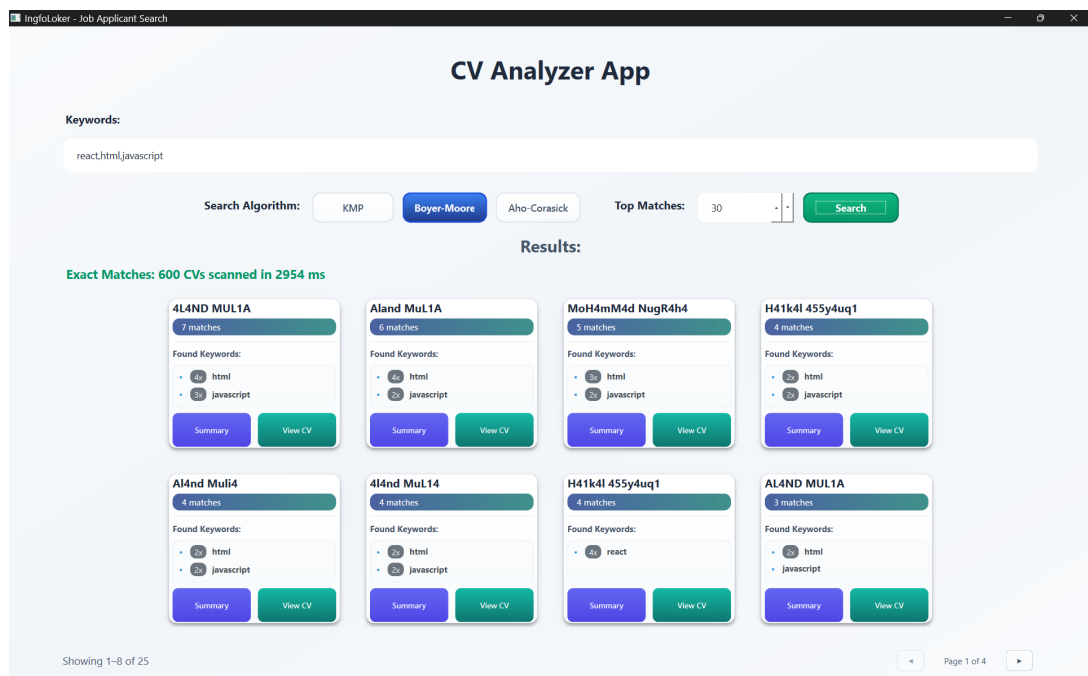


Gambar 9: Percobaan Algoritma KMP dengan keyword panjang

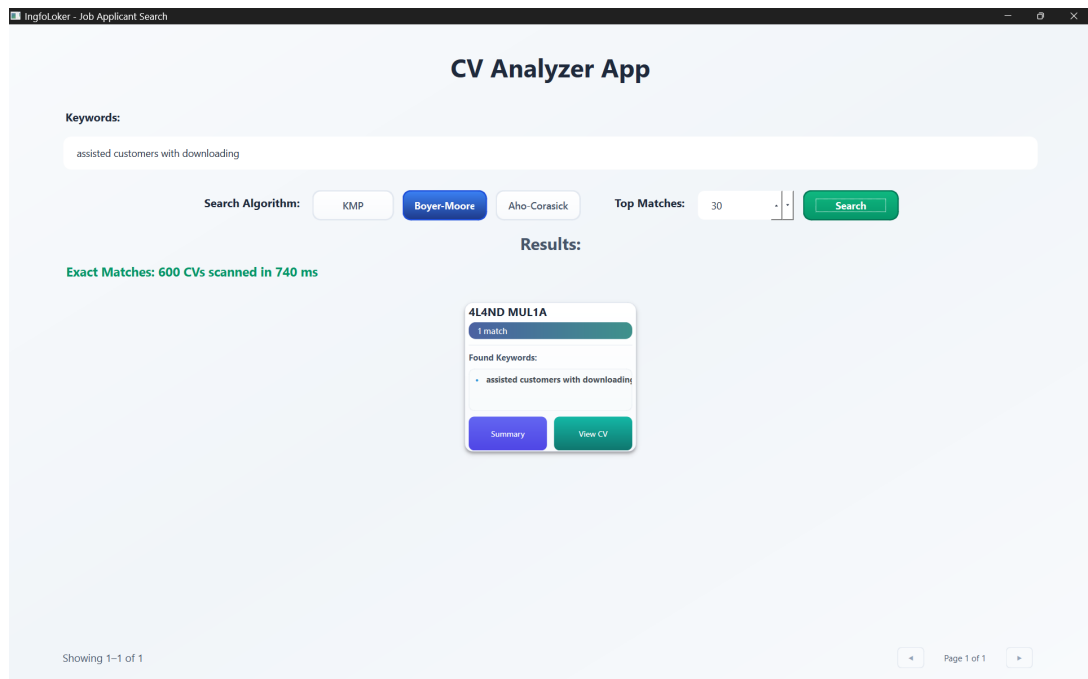


**Gambar 10:** Percobaan Algoritma KMP dengan keyword nama *applicant*

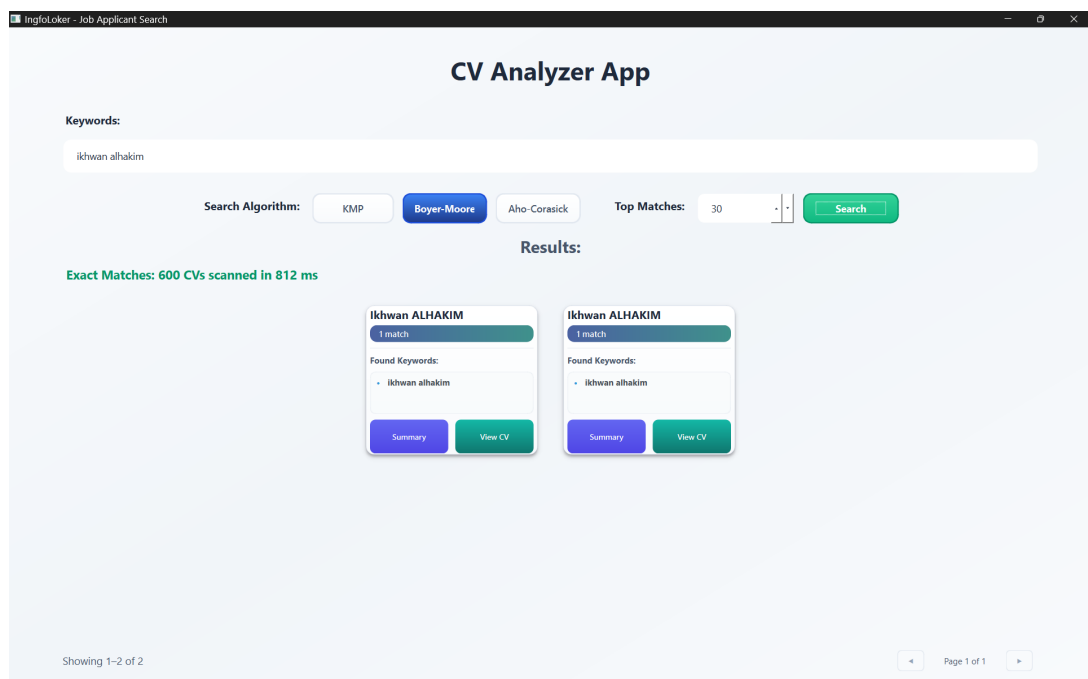
## b. Boyer-Moore



**Gambar 11:** Percobaan Algoritma Boyer-Moore dengan tiga keywords

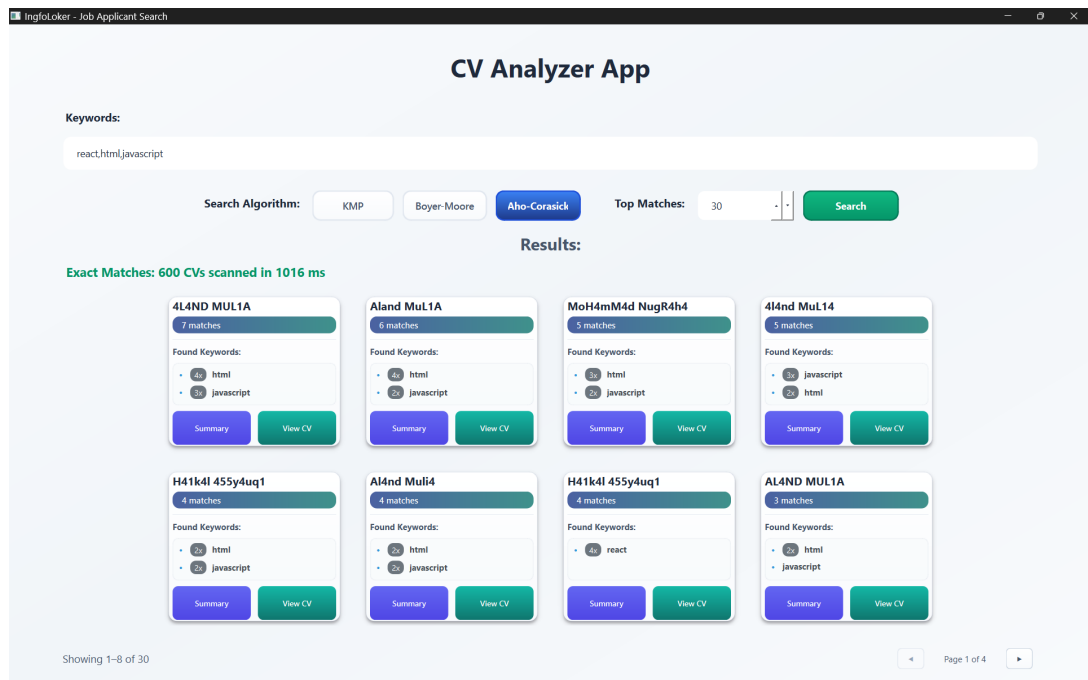


**Gambar 12:** Percobaan Algoritma Boyer-Moore dengan keyword panjang

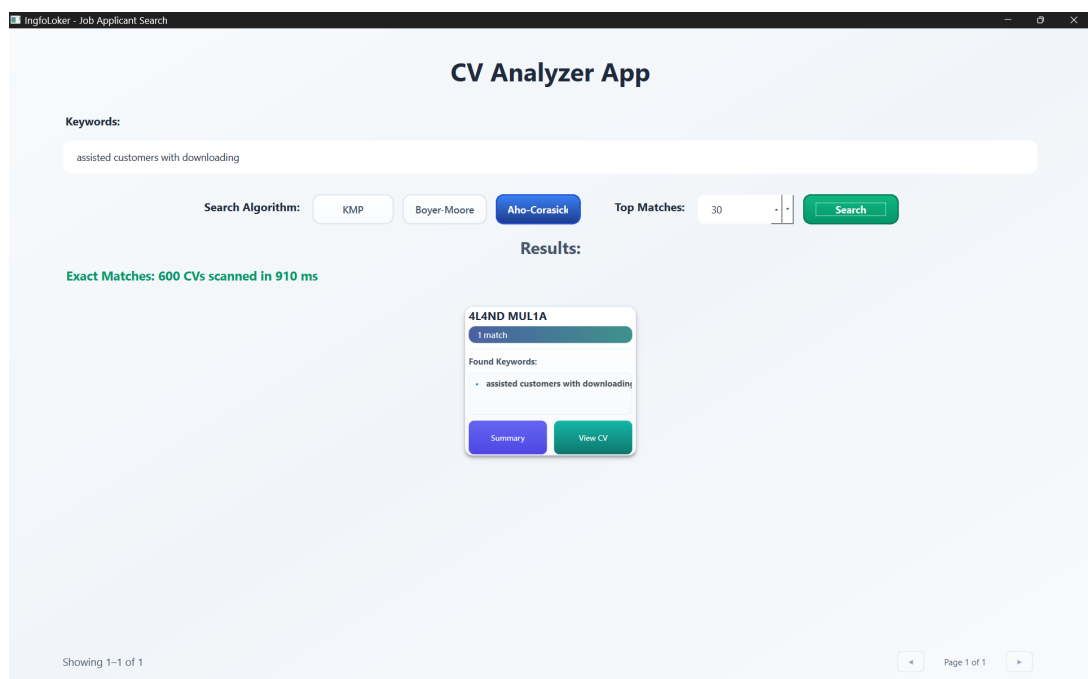


**Gambar 13:** Percobaan Algoritma Boyer-Moore dengan keyword nama *applicant*

### c. Aho-Corasick

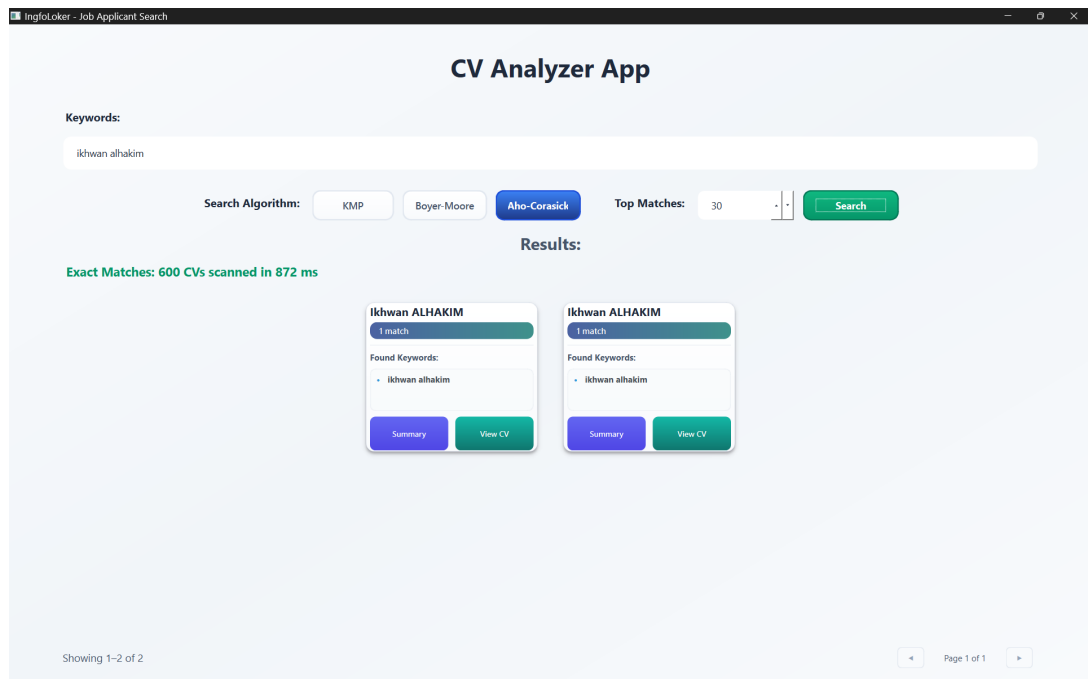


Gambar 14: Percobaan Algoritma Aho-Corasick dengan tiga keywords



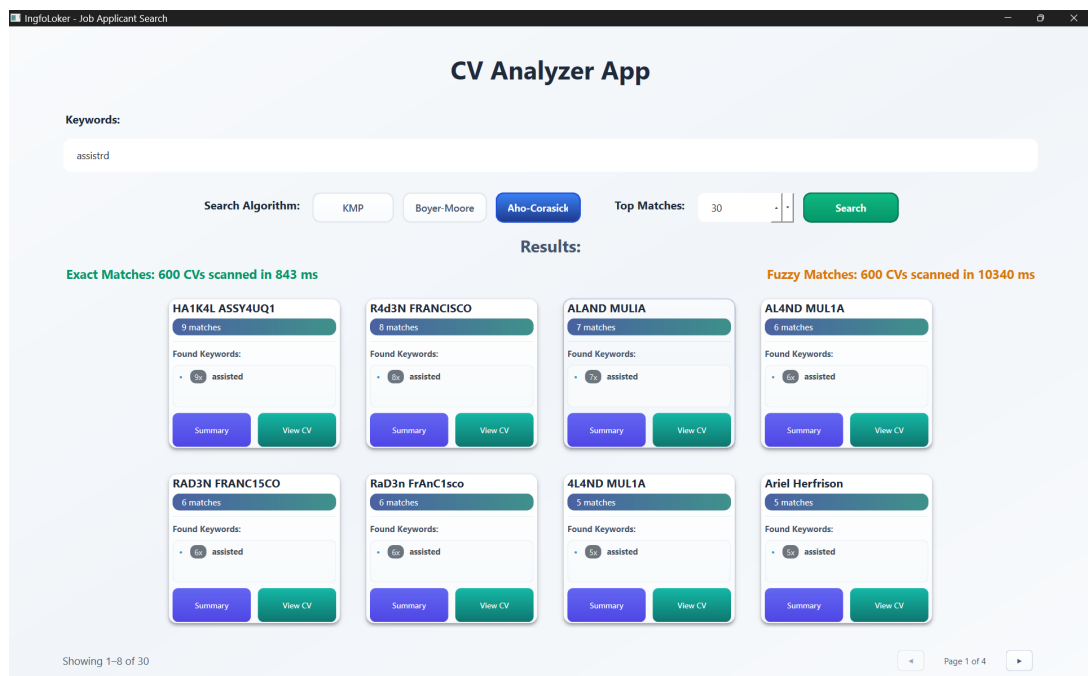
Gambar 15: Percobaan Algoritma Aho-Corasick dengan keyword panjang



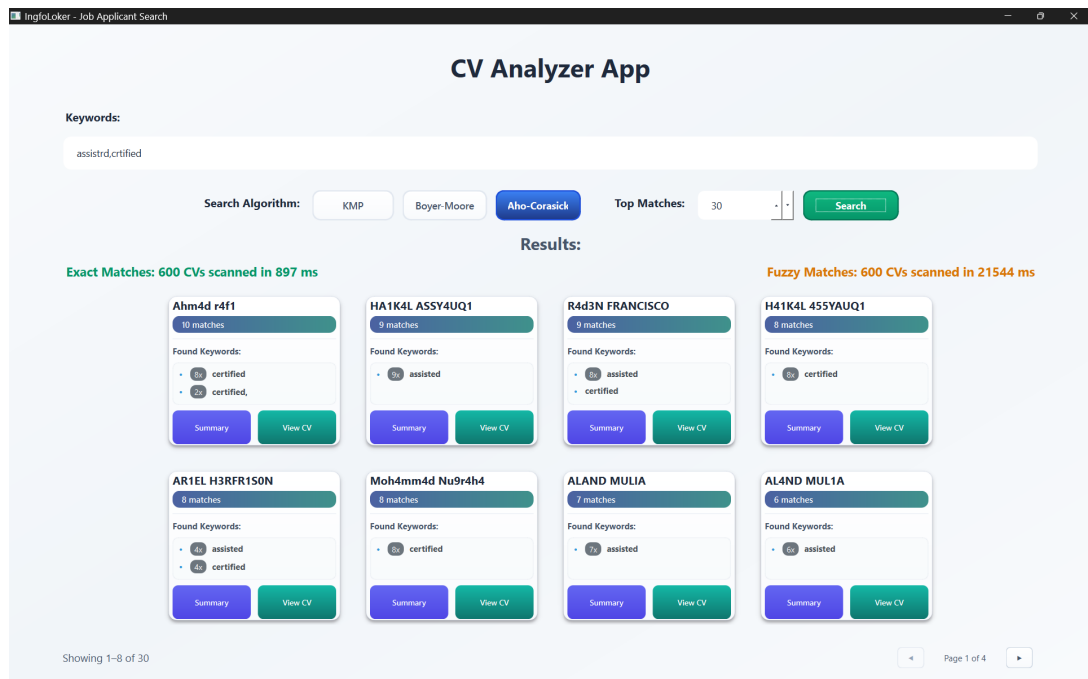


**Gambar 16:** Percobaan Algoritma Aho-Corasick dengan keyword nama

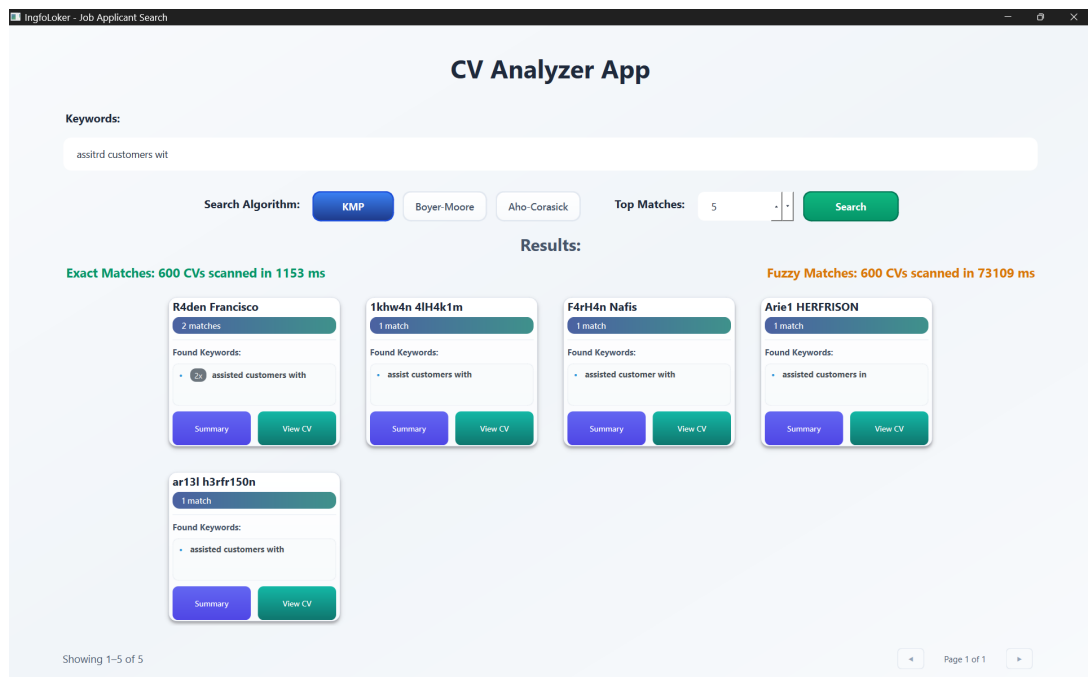
#### d. Levenshtein Distance



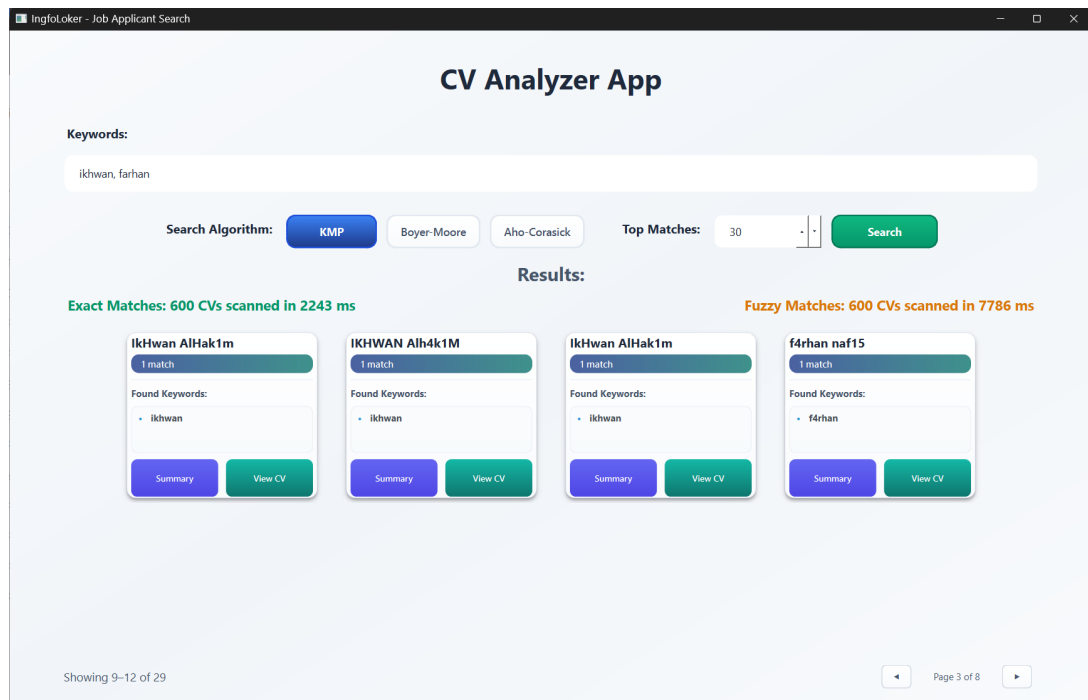
**Gambar 17:** Percobaan Algoritma Levenshtein dengan satu keyword



**Gambar 18:** Percobaan Algoritma Levenshtein dengan dua keyword

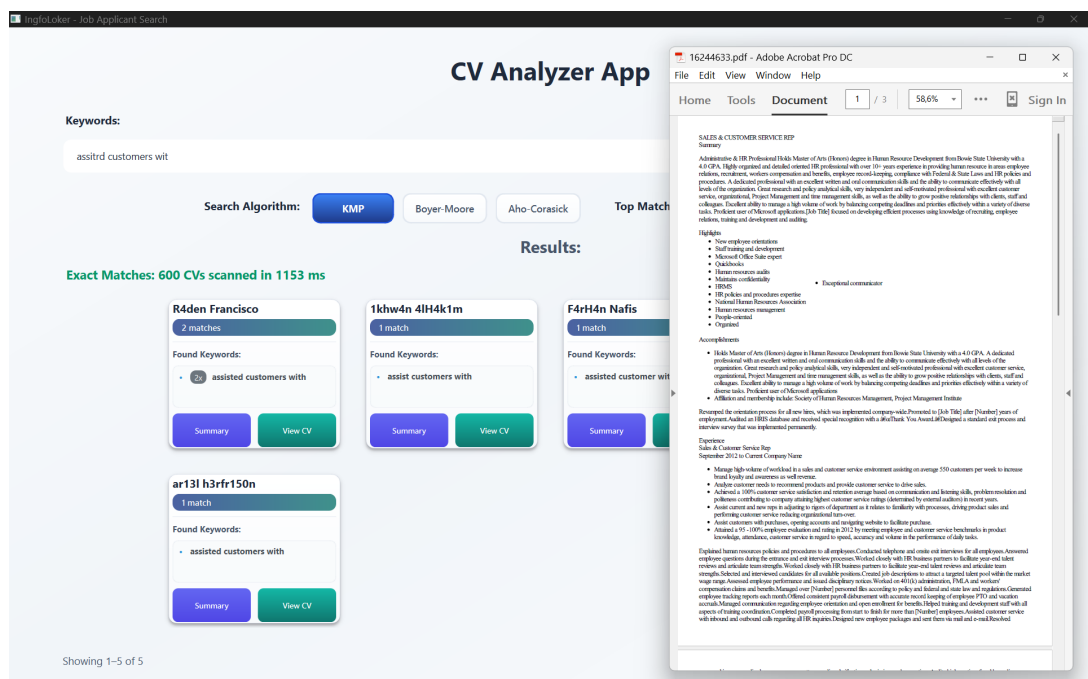


**Gambar 19:** Percobaan Algoritma Levenshtein dengan panjang Keyword lebih dari satu



**Gambar 20:** Percobaan Algoritma Levenshtein dengan satu keyword tidak ditemukan *exact match*-nya

#### f. View CV



**Gambar 21:** Percobaan View CV

## e. RegEx (Summary)

CV Summary

Personal Information

1khw4n 4IH4k1m

Birthdate: 2003-06-06

Address: Jl. Kenari No. 23, Parliman

Phone: 081223456789

Summary

Administrative & HR Professional Holds Master of Arts (Honors) degree in Human Resource Development from Bowie State University with a 4.0 GPA. Highly organized and detailed oriented HR professional with over 10+ years experience in providing human resource in areas employee relations, recruitment, workers compensation and benefits, employee record-keeping, compliance with Federal & State Laws and HR policies and procedures. A dedicated professional with an excellent written and oral communication skills and the ability to communicate effectively with all levels of the organization. Great research and policy analytical skills, very independent and self-motivated professional with excellent customer service, organizational, Project Management and time management skills, as well as the ability to grow positive relationships with clients, staff and colleagues. Excellent ability to manage a high volume of work by balancing competing deadlines and priorities effectively within a variety of diverse tasks. Proficient user of Microsoft applications.[Job Title] focused on developing efficient processes using knowledge of recruiting, employee relations, training and development and auditing.

Technical Skills

New employee orientations

Staff training and development

Microsoft Office Suite expert

Quickbooks

Human resources audits

Maintains confidentiality

HRMS

National Human Resources Association

Human resources management

People-oriented

Organized

Exceptional communicator

Professional Experience

September 2012 to Current

Manage high-volume of workload in a sales and customer service environment assisting on average 550 customers per week to increase

Office Manager

January 2011 to August 2012

Created social media initiatives for new employee search strategies.

August 2007 to September 2010

Administration & Organization Provided administrative and business support for the firm, CEO and executive team members.

Education

Master of Arts in Human Resource Development BOWIE STATE UNIVERSITY

Arts Human Resource

Master of Arts in Human Resource Development

BOWIE UNIVERSITY Bowie

Project Management Bowie State University Certificate in Project Management

2011

Bachelor's in Human Resources Management CENTRAL UNIVERSITY COLLEGE City

Human Resources Management

Bachelor's degree in Human Resources Management July

CENTRAL UNIVERSITY COLLEGE

2003

Gambar 22: Percobaan Algoritma RegEx untuk Summary

42

IF2211 - Strategi Algoritma

#### 4.4. Analisis dan Pembahasan

Secara teori, algoritma KMP memiliki kompleksitas waktu sebesar  $O(n + m)$ , di mana  $n$  menyatakan panjang teks (dalam konteks ini adalah jumlah karakter dari satu CV), dan  $m$  merupakan panjang dari pola pencarian (gabungan kata kunci). Kompleksitas ini menjadikan KMP bersifat linier terhadap total ukuran input. Proses *preprocessing* berupa pembuatan tabel **lps** (longest prefix suffix) membutuhkan waktu  $O(m)$ , namun selama proses pencarian, algoritma tidak pernah kembali mundur di dalam teks. Ketika diujikan pada pencarian tiga kata kunci dalam 600 CV, KMP menunjukkan performa stabil dan cepat dengan waktu eksekusi sekitar 3546 milidetik, memperkuat validitas efisiensinya secara teoritis, meskipun sedikit lebih lambat dibanding algoritma Boyer–Moore dalam kasus yang sama.

Algoritma Boyer–Moore dikenal sebagai salah satu algoritma pencocokan paling cepat secara praktis, terutama karena dua teknik utama yang digunakan, yaitu *bad character rule* dan *good suffix rule*, yang memungkinkan lompatan lebih jauh saat terjadi ketidakcocokan. Rata-rata kompleksitas waktu untuk satu pola adalah mendekati  $O(n/m)$ , sehingga sangat efisien jika pola cukup pendek dan teks cukup panjang. Dalam penerapannya pada pencarian tiga kata kunci dalam sekumpulan besar CV, algoritma ini menunjukkan performa tercepat dengan waktu sekitar 2954 milidetik, yang menunjukkan kemampuannya dalam mengeksplorasi struktur pola dan teks untuk menghasilkan lompatan-lompatan optimal.

Berbeda dari KMP dan Boyer–Moore yang dirancang untuk pencarian satu pola, algoritma Aho–Corasick dioptimalkan untuk pencarian banyak pola secara bersamaan (multi-pattern matching). Dengan membangun automaton dari semua pola yang ingin dicari, proses pencarian dapat dilakukan hanya dalam satu lintasan terhadap teks. Automaton ini dibentuk dalam waktu  $O(\Sigma)$ , dengan  $\Sigma$  adalah total panjang semua pola, dan pencocokan berlangsung dalam  $O(n + z)$ , di mana  $z$  adalah jumlah total kecocokan. Ketika diuji pada pencarian dua kata kunci di dalam 600 CV, Aho–Corasick mampu menyelesaikan pencocokan dengan waktu sekitar 897 milidetik, menjadikannya algoritma paling efisien dalam skenario banyak pola sekaligus.

Levenshtein Distance merupakan algoritma yang digunakan dalam pencarian tak eksak (fuzzy matching), yaitu dengan mengukur seberapa besar perbedaan antara dua string dalam bentuk jumlah minimum operasi edit (penyisipan, penghapusan, atau substitusi). Pendekatan ini dilakukan melalui pemrograman dinamis dengan kompleksitas waktu  $O(nm)$ , yang berarti bersifat kuadratik terhadap panjang teks dan pola. Ketika diterapkan pada dua kata kunci di atas korpus 600 CV, waktu pemrosesan mencapai sekitar 21 544 milidetik, menunjukkan bahwa metode ini jauh lebih lambat dibanding metode exact matching. Keterlambatan ini selaras dengan kompleksitasnya yang lebih tinggi, namun tetap relevan jika sistem perlu mengakomodasi variasi atau kesalahan penulisan dalam kata kunci.

Jika dibandingkan berdasarkan teori kompleksitas rata-rata, maka urutan efisiensi dari yang tercepat hingga terlambat adalah: Boyer–Moore (sublinear), Aho–Corasick dan KMP (linier), lalu Levenshtein Distance (kuadratik). Hasil eksperimen yang dilakukan memperkuat prediksi tersebut. Boyer–Moore menunjukkan keunggulan dalam pencarian eksak multi-pola pendek, Aho–Corasick tampil unggul dalam pencarian banyak pola sekaligus, KMP konsisten dalam performa linier, sementara Levenshtein Distance menjadi metode paling mahal secara waktu namun tetap diperlukan dalam skenario pencarian tak eksak.

## Bab V

### Penutup

#### 5.1. Kesimpulan

Dalam proyek Tugas Besar 3 IF2211 Strategi Algoritma ini, algoritma KMP, Boyer-Moore, dan Aho-Corasick dapat digunakan dalam proses pencarian *CV* pada Sistem ATS (Applicant Tracking System) Berbasis *CV* Digital. Kami berhasil untuk mengimplementasikan algoritma KMP, Boyer-Moore, dan Aho-Corasick dalam aplikasi pencarian *CV* kami. Selain menghasilkan *CV* yang sesuai dengan keyword yang dicari, program kami juga dapat menghasilkan **Summary** dan menampilkan PDF pelamar kerja.

#### 5.2. Saran

Pelaksanaan Tugas Kecil 3 IF2211 Strategi Algoritma di Semester II (Genap) Tahun 2024/2025 merupakan pengalaman yang sangat berharga bagi kami. Dari pengalaman ini, kami ingin berbagi beberapa saran kepada pembaca yang mungkin akan menghadapi tugas serupa di masa depan:

##### **rzi**

Tantangan terbesar sebenarnya dalam tucil dan tubes di IF hanyalah prokas. Sungguh, dunia begitu damai jika kelar tucil dan tubes jauh hari sebelum deadline. Terima kasih banyak farr, semoga yang disemogakan secepatnya tersemogakan.

##### **igun**

Sebaiknya memberikan seedingga jangan di hari H deadline

##### **rejaah**

Alhamdullillah kelar

Semoga saran-saran ini membantu pembaca dalam menyiapkan diri untuk menangani tugas serupa di masa depan.

#### 5.3. Refleksi

Ruang perbaikan dan pengembangan dalam mengerjakan tugas dapat difokuskan pada beberapa aspek yang dapat ditingkatkan. Pertama-tama, pengaturan waktu menjadi kunci utama. Kami menyadari bahwa perencanaan waktu yang lebih baik dapat meningkatkan efisiensi pengerjaan. Menerapkan strategi manajemen waktu, seperti membuat jadwal yang terstruktur dan menetapkan tenggat waktu internal untuk setiap tahap pekerjaan, dapat membantu menghindari tekanan waktu yang tidak perlu.

Selain itu, ketelitian membaca spesifikasi dari awal menjadi aspek yang perlu diperhatikan. Memahami secara menyeluruh tentang apa yang diminta dalam tugas, termasuk detail-detail kecil, dapat mengurangi risiko kesalahan dan memastikan pekerjaan berjalan sesuai dengan harapan. Menempatkan perhatian ekstra pada spesifikasi tugas dapat meminimalkan revisi dan penyesuaian yang mungkin diperlukan.

Komunikasi tim yang baik juga dapat dioptimalkan. Membuat saluran komunikasi yang jelas dan terbuka di antara anggota tim dapat menghindari kebingungan dan memastikan bahwa semua anggota memiliki pemahaman yang sama tentang tujuan dan tanggung jawab masing-masing. Diskusi reguler dan pembahasan mengenai kemajuan proyek dapat meningkatkan keterlibatan semua anggota tim.

Dalam konteks pengembangan web, penambahan fitur-fitur yang mendukung dapat memberikan nilai tambah. Memperhatikan kebutuhan pengguna dan mengidentifikasi area yang dapat diperbaiki atau ditingkatkan dalam hal fungsionalitas dapat meningkatkan kualitas tugas. Pemikiran kreatif dalam mengimplementasikan fitur-fitur baru yang relevan dengan tujuan proyek dapat membuat proyek lebih bermanfaat dan menarik.

Dalam konteks optimalisasi algoritma, dapat melakukan eksplorasi lebih jauh lagi mengenai efisiensi kinerja program. Dengan memahami cara agar kinerja program lebih efisien, program akan menjadi lebih cepat dan lebih baik lagi. Hal tersebut sangat diperlukan agar program kami bisa menjadi program yang layak untuk dipublikasikan.

Terakhir, evaluasi diri secara berkala dapat menjadi langkah penting. Menerima umpan balik, baik dari anggota tim maupun asisten, dan memanfaatkannya sebagai dasar untuk perbaikan lebih lanjut adalah cara efektif untuk terus berkembang. Selalu terbuka terhadap saran dan berkomitmen untuk belajar dari setiap pengalaman dapat membantu memperbaiki kinerja secara berkelanjutan.



## Lampiran

*Repository* dan Video program pada Tugas Besar 3 ini dapat diakses melalui tautan berikut:

- **GitHub:**

[https://github.com/zirachw/Tubes3\\_IngfoLoker](https://github.com/zirachw/Tubes3_IngfoLoker)

- **Video:**

[https://youtu.be/N80C-Dn\\_ksM](https://youtu.be/N80C-Dn_ksM)

Adapun berikut tabel spesifikasi dari Tugas Besar 3 ini:

**Tabel 1:** TABEL SPESIFIKASI TUGAS BESAR 3

No	Poin	Ya	Tidak
1	Aplikasi dapat dijalankan.	✓	
2	Aplikasi menggunakan basis data berbasis SQL dan berjalan dengan lancar.	✓	
3	Aplikasi dapat mengekstrak informasi penting menggunakan Regular Expression (Regex).	✓	
4	Algoritma <i>Knuth-Morris-Pratt</i> (KMP) dan <i>Boyer-Moore</i> (BM) dapat menemukan kata kunci dengan benar.	✓	
5	Algoritma Levenshtein Distance dapat mengukur kemiripan kata kunci dengan benar.	✓	
6	Aplikasi dapat menampilkan <i>summary CV applicant</i> .	✓	
7	Aplikasi dapat menampilkan CV <i>applicant</i> secara keseluruhan.	✓	
8	Membuat laporan sesuai dengan spesifikasi.	✓	
9	Membuat bonus enkripsi data profil <i>applicant</i> .	✓	
10	Membuat bonus algoritma Aho-Corasick.	✓	
11	Membuat bonus video dan diunggah pada Youtube.	✓	

## Referensi

- Rinaldi Munir. (2025). "Pencocokan string (string matching) dengan algoritma brute force, KMP, Boyer-Moore." Diakses pada 13 Juni 2025. [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-\(2025\).pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf)
- Rinaldi Munir. (2025). "Pencocokan string dengan regular expression (regex)." Diakses pada 13 Juni 2025. [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/24-String-Matching-dengan-Regex-\(2025\).pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/24-String-Matching-dengan-Regex-(2025).pdf)
- Geeks For Geeks. (2025). "Pattern Searching." Diakses pada 13 Juni 2025. <https://www.geeksforgeeks.org/pattern-searching/>
- Geeks For Geeks. (2025). "KMP Algorithm for Pattern Searching." Diakses pada 13 Juni 2025. <https://www.geeksforgeeks.org/dsa/kmp-algorithm-for-pattern-searching/>
- Siddharth. (2021). "The Boyer Moore String Search Algorithm" Diakses pada 14 Juni 2025. <https://medium.com/@siddharth.21/the-boyer-moore-string-search-algorithm-674906cab162>
- pymupdf. (2025). "PyMuPDF Library" Diakses pada 14 Juni 2025. <https://github.com/pymupdf/PyMuPDF>
- Andrew. (2024). "Setting Up MYSQL in a docker container (+ phpmyadmin)" Diakses pada 14 Juni 2025. <https://youtu.be/BuAELkBFMh8?si=kPgvcFHLLuY1F8wW>