IF2211 Strategi Algoritma

Pemanfaatan Pattern Matching untuk Membangun Sistem ATS (Applicant Tracking System) Berbasis CV Digital

Laporan Tugas Besar 3

Disusun untuk memenuhi tugas mata kuliah IF2211 Strategi Algoritma pada Semester IV Tahun Akademik 2024/2025

Dibuat Oleh

Dita Maheswari	13523125
Ahmad Syafiq	13523135
Juan Sohuturon Arauna Siagian	18222086

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

2025

DAFTAR ISI

DAFTAR ISI 2
BAB 1 DESKRIPSI TUGAS 3
BAB 2 LANDASAN TEORI 4
2.1 Algoritma Knuth-Morris-Pratt 4
2.2 Algoritma Boyer-Moore 7
2.3 Algoritma Aho–Corasick 9
2.4 Penjelasan Singkat Applicant Tracking System 12
BAB 3 APLIKASI PEMECAHAN MASALAH 13
3.1 Langkah-Langkah Pemecahan Masalah 13
1. Pengolahan File PDF 13
2. Pencocokan Keyword 13
3. Penampilan Rangkuman 14
3.2 Proses Pemetaan Masalah Menjadi Elemen-Elemen Algoritma KMP, BM, dan AC 14
3.2.1 Pemetaan Elemen dan Penyelesaian Solusi dengan Algoritma KMP 14
3.2.2 Pemetaan Elemen dan Penyelesaian Solusi dengan Algoritma BM 15
3.2.3 Pemetaan Elemen dan Penyelesaian Solusi dengan Algoritma AC 16
3.3 Fitur Fungsional dan Arsitektur Aplikasi yang Dibangun 17
3.4 Contoh Ilustrasi Kasus 18
BAB 4 IMPLEMENTASI DAN PENGUJIAN 21
4.1 Spesifikasi Teknis Program (Struktur Data, Fungsi, dan Prosedur yang Dibangun) 21
4.1.1 Struktur Data 21
4.1.2 Fungsi dan Prosedur 22
4.2 Penjelasan Tata Cara Penggunaan Program 34
4.3 Hasil Pengujian 35
4.4 Analisis Hasil Pengujian 37
BAB 5 KESIMPULAN DAN SARAN 39
5.1 Kesimpulan 39
5.2 Saran 39
5.3 Refleksi 40
LAMPIRAN 41
A. Github 41
B. Video 41
C. Tabel Pemeriksaan 41
DAFTAR PUSTAKA 43

BAB 1 DESKRIPSI TUGAS

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.

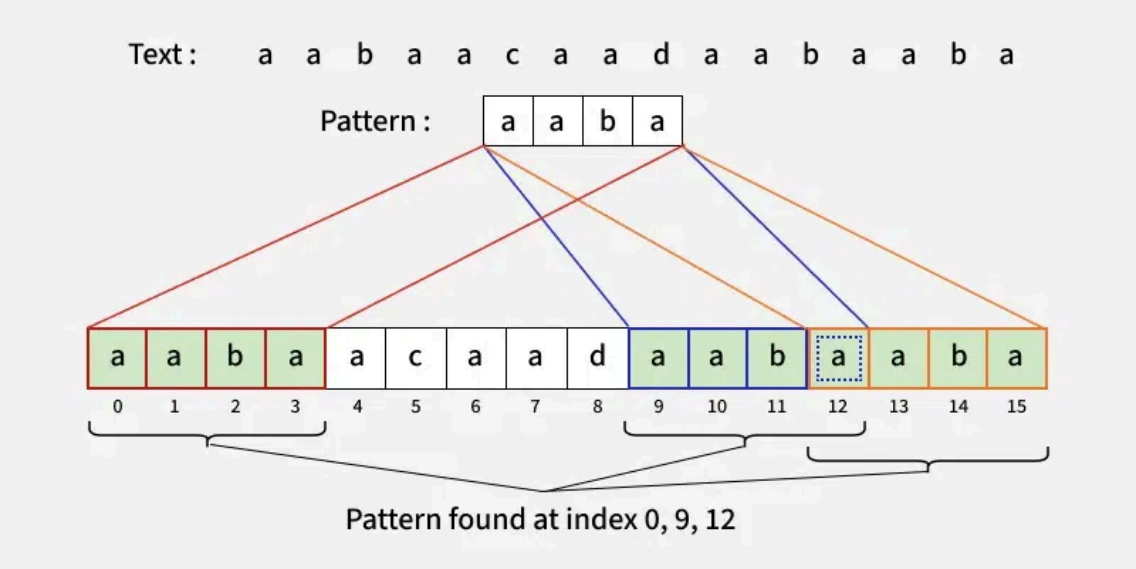
Pattern matching adalah teknik untuk menemukan dan mencocokkan pola tertentu dalam teks. Dalam konteks ini, algoritma Boyer-Moore dan Knuth-Morris-Pratt (KMP) sering digunakan karena keduanya menawarkan efisiensi tinggi untuk pencarian teks di dokumen besar. Algoritma ini memungkinkan sistem ATS untuk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

Dalam Tugas Besar 3 ini, mahasiswa 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 2 LANDASAN TEORI

2.1 Algoritma *Knuth-Morris-Pratt*

Algoritma KMP (*Knuth-Morris-Pratt*) adalah algoritma pencocokan string yang digunakan untuk mencari kemunculan suatu pola (pattern) dalam sebuah teks. Algoritma ini efisien karena memanfaatkan informasi tentang pola yang sudah diketahui untuk menghindari perbandingan yang tidak perlu saat pencocokan, sehingga memiliki kompleksitas waktu O(m+n). Algoritma KMP dirancang untuk mengatasi kelemahan algoritma brute-force yang membandingkan pola dengan teks secara satu per satu, yang bisa lambat jika ada banyak kemiripan antar karakter dalam pola. KMP menggunakan tabel "prefix function" atau "longest proper prefix suffix (LPS)" untuk menyimpan informasi tentang panjang awalan (prefix) terpanjang dari pola yang juga merupakan akhiran (suffix) dari bagian pola tersebut. Tabel LPS ini digunakan untuk menggeser pola ketika terjadi ketidakcocokan, sehingga proses pencocokan bisa dilanjutkan tanpa harus mengulang perbandingan dari awal.



Cara kerja algoritma KMP adalah sebagai berikut:

1. Membangun tabel LPS

Langkah pertama adalah membuat tabel LPS untuk pola yang akan dicari. Tabel ini menunjukkan panjang awalan terpanjang dari pola yang juga merupakan akhiran dari setiap prefix pola

2. Pencocokan string

   Algoritma KMP melakukan pencocokan string antara pola dan teks karakter demi karakter

3. Penanganan ketidakcocokan

   Jika terjadi ketidakcocokan, algoritma tidak langsung menggeser pola satu posisi ke kanan seperti pada brute-force. Sebaliknya, algoritma akan menggunakan tabel LPS untuk menggeser pola sejauh yang memungkinkan berdasarkan informasi prefix yang sudah diketahui

4. Pencarian berlanjut

   Proses pencocokan berlanjut hingga pola ditemukan di dalam teks atau sampai seluruh teks telah diperiksa

Algoritma KMP memiliki beberapa keuntungan, yaitu KMP memiliki kompleksitas waktu linear yang membuatnya lebih cepat dibandingkan algoritma brute-force (terutama untuk pola yang memiliki banyak kemiripan). KMP juga memanfaatkan informasi tentang pola yang sudah diketahui untuk menghindari perbandingan yang tidak perlu.

Berikut adalah contoh implementasi algoritma KMP

```
Function computeLPS(pattern):

    m ← length(pattern)

    LPS ← array of size m, filled with 0

    length ← 0     // panjang prefix yang juga suffix

    i ← 1


    while i < m:

        if pattern[i] = pattern[length]:

            length ← length + 1
```

```
                LPS[i] ← length
                i ← i + 1
            else:
                if length ≠ 0:
                    length ← LPS[length - 1]
                else:
                    LPS[i] ← 0
                    i ← i + 1
        return LPS


    Function KMP_Search(text, pattern):
        n ← length(text)
        m ← length(pattern)
        LPS ← computeLPS(pattern)
        i ← 0   // index untuk text
        j ← 0   // index untuk pattern

        while i < n:
            if pattern[j] = text[i]:
                i ← i + 1
                j ← j + 1

            if j = m:
                Output: "Pola ditemukan pada index", i - j
                j ← LPS[j - 1]

            else if i < n and pattern[j] ≠ text[i]:
                if j ≠ 0:
                    j ← LPS[j - 1]
                else:
                    i ← i + 1
```

2.2 Algoritma *Boyer-Moore*

Algoritma Boyer-Moore adalah salah satu algoritma pencocokan string yang efisien dalam mencari suatu pola (pattern) dalam teks. Algoritma ini bekerja dengan cara membandingkan karakter dari kanan ke kiri, yang memungkinkan algoritma untuk melompati beberapa karakter dalam teks dan mempercepat proses pencarian. Algoritma ini dianggap sebagai salah satu algoritma pencocokan string paling efisien untuk aplikasi umum. Algoritma Boyer-Moore merupakan algoritma yang lebih cepat prosesnya dibandingkan algoritma string matching lainnya karena algoritma ini dapat melompati beberapa karakter dalam teks, sehingga mengurangi jumlah perbandingan yang diperlukan. Cara kerja algoritma Boyer-Moore adalah sebagai berikut:

1. Pembuatan tabel

   Algoritma BM melakukan praproses pada pattern yang dicari untuk membuat dua tabel, yaitu tabel bad character dan good suffix. Tabel bad character adalah tabel yang berisi informasi terakhir yang cocok dalam pola. Jika karakter dalam teks tidak cocok dengan karakter dalam pola, tabel ini digunakan untuk menentukan seberapa jauh pola dapat digeser. Sedangkan, tabel good suffix adalah tabel yang berisi informasi tentang kecocokan sebagian suffix dalam pola. Jika terjadi kecocokan sebagian, tabel ini membantu menentukan seberapa jauh pola dapat digeser berdasarkan kecocokan suffix tersebut

2. Pencocokan

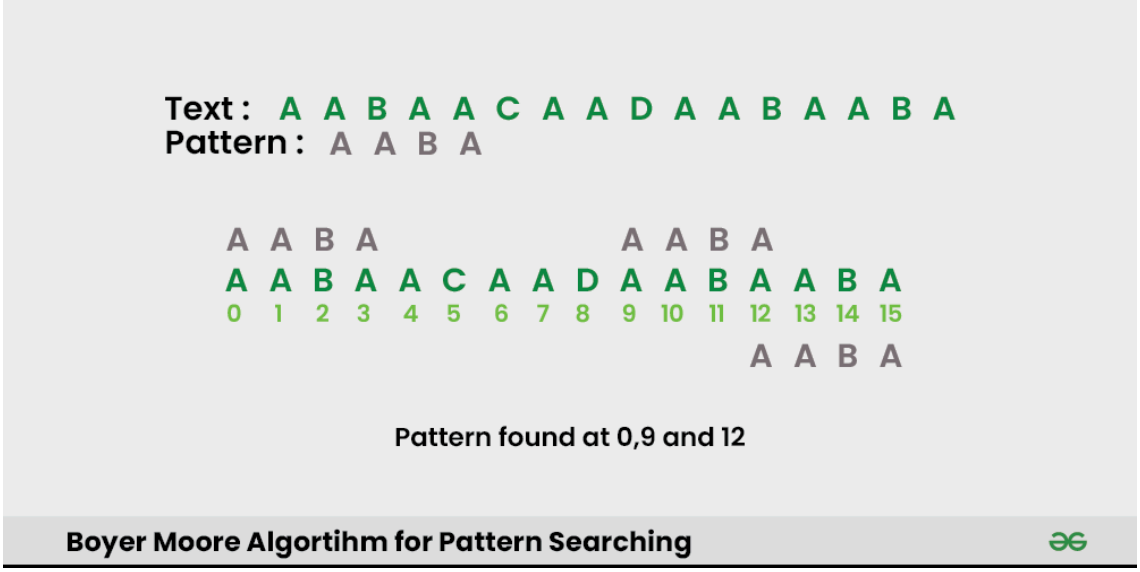
   Algoritma BM memulai pencocokan dari karakter terakhir pada pola dan bergerak ke kiri

3. Pergeseran

   Setelah terjadi ketidakcocokan, algoritma menggunakan tabel bad character dan tabel good suffix untuk menentukan seberapa jauh pola dapat digeser. Pola digeser sejauh pergeseran terbesar yang ditentukan oleh kedua tabel tersebut

4. Pengulangan

   Proses pencocokan dan pergeseran diulang sampai pola ditemukan atau seluruh teks telah dicari



Berikut adalah contoh implementasi algoritma Boyer-Moore

```
Function buildBadCharTable(pattern):
    badChar ← array of size 256, all filled with -1
    for i from 0 to length(pattern) - 1:
        badChar[ASCII(pattern[i])] ← i
    return badChar


Function BM_Search(text, pattern):
    n ← length(text)
    m ← length(pattern)
    badChar ← buildBadCharTable(pattern)
    s ← 0     // shift of the pattern


    while s ≤ n - m:
        j ← m - 1
```

```
            while j ≥ 0 and pattern[j] = text[s + j]:
                j ← j - 1

            if j < 0:
                Output: "Pola ditemukan pada index", s
                  s ← s + (if s + m < n then m - badChar[ASCII(text[s + m])] else
1)
            else:
                shift ← max(1, j - badChar[ASCII(text[s + j])])
                s ← s + shift
```

2.3 Algoritma *Aho–Corasick*

Algoritma Aho-Corasick adalah algoritma pencocokan string yang memungkinkan pencarian beberapa pola (string) dalam sebuah teks secara efisien. Algoritma ini dikembangkan oleh Alfred V. Aho dan Margaret J. Corasick pada tahun 1975. Algoritma ini menggunakan struktur data trie dan tautan untuk mencari kemunculan pola-pola dalam teks, sehingga dapat menemukan semua kemunculan pola secara bersamaan dalam satu kali pembacaan teks. Kompleksitas waktu algoritma Aho-Corasick adalah O(n + m + z), di mana n adalah panjang teks, m adalah total panjang semua pola, dan z adalah jumlah total kecocokan pola dalam teks. Ini membuatnya sangat efisien untuk pencarian beberapa pola dalam teks yang besar. Cara kerja algoritma Aho-Corasick adalah sebagai berikut:

1. Membangun trie (pohon awalan)

   Algoritma Aho-Corasick membangun struktur data trie (pohon awalan) dari semua pattern yang ingin dicari. Setiap node pada trie mewakili awalan pada sebuah pattern.

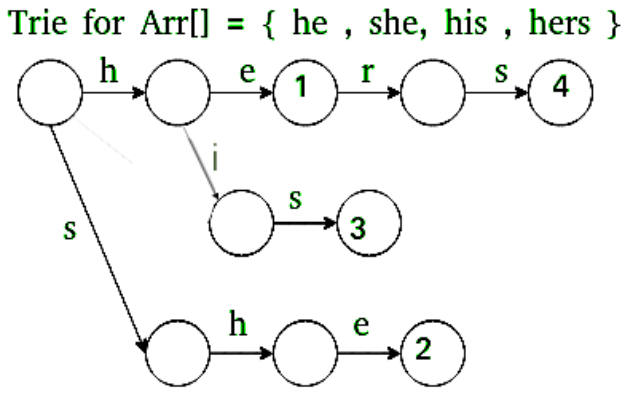
2. Menghitung tautan suffix

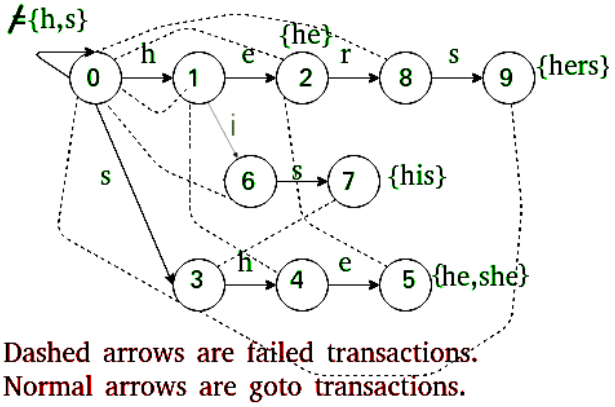
   Setelah trie dibangun, algoritma menghitung tautan suffix untuk setiap node. Tautan suffix mengarah ke node lain dalam trie yang mewakili suffix terpanjang dari string yang sesuai dengan node saat ini

3. Pencarian dalam teks

   Algoritma ini kemudian membaca teks satu per satu karakter. Pada setiap karakter, algoritma mengikuti tautan pada trie. Jika menemukan kecocokan pola,

algoritma akan melaporkan kecocokan tersebut. Jika tidak menemukan kecocokan, algoritma akan mengikuti tautan sufiks untuk mencoba kecocokan dengan pola lain.





Berikut adalah contoh implementasi algoritma Aho-Corasick:

```
Function buildTrie(patterns):

    root ← new TrieNode()


    for each pattern in patterns:

        node ← root

        for char in pattern:

            if char not in node.next:

                node.next[char] ← new TrieNode()
```

```
                node ← node.next[char]
            node.output.append(pattern)

        return root

    Function buildFailureLinks(root):
        queue ← empty queue

        for each char, node in root.next:
            node.fail ← root
            queue.enqueue(node)

        while not queue.empty():
            current ← queue.dequeue()
            for each char, nextNode in current.next:
                failNode ← current.fail
                while failNode ≠ null and char not in failNode.next:
                    failNode ← failNode.fail

                if failNode = null:
                    nextNode.fail ← root
                else:
                    nextNode.fail ← failNode.next[char]
                    nextNode.output.extend(nextNode.fail.output)

                queue.enqueue(nextNode)

    Function search(text, root):
        node ← root

        for i from 0 to length(text) - 1:
            char ← text[i]
```

```
            while node ≠ root and char not in node.next:
                node ← node.fail

            if char in node.next:
                node ← node.next[char]
            else:
                node ← root

            if node.output ≠ empty:
                for each pattern in node.output:
                        Output: "Pattern", pattern, "found at index", i -
length(pattern) + 1
```

2.4 Penjelasan Singkat Applicant Tracking System

Aplikasi yang dikembangkan dalam tugas besar ini merupakan sebuah sistem ATS berbasis CV digital. Sistem ini dirancang untuk membantu perekrut dalam mendeteksi kecocokan antara kata kunci tertentu (misalnya keahlian atau teknologi) dengan isi CV pelamar. Aplikasi ini menerapkan algoritma pencocokan string seperti Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM) untuk pencocokan exact match, serta Levenshtein Distance untuk fuzzy matching ketika terjadi kesalahan pengetikan.

Setiap CV dalam format PDF diekstraksi menjadi teks panjang, kemudian dicocokkan dengan daftar keyword yang diberikan pengguna. Aplikasi menampilkan hasil pencarian dalam bentuk daftar CV paling relevan, lengkap dengan informasi penting dari pelamar seperti nama, kontak, keahlian, pengalaman kerja, dan riwayat pendidikan. Data pelamar juga disimpan dalam basis data MySQL untuk mendukung integrasi sistem dan penyimpanan data secara terstruktur. Pengguna dapat:

1. Memasukkan keyword pencarian.
2. Memilih algoritma pencarian (KMP atau BM atau AC).
3. Menentukan jumlah hasil yang ingin ditampilkan.
4. Melihat hasil pencarian berupa CV yang paling relevan, ringkasan informasi pelamar, serta akses ke file CV asli.

BAB 3 APLIKASI PEMECAHAN MASALAH

3.1 Langkah-Langkah Pemecahan Masalah

Langkah-langkah pemecahan masalah dalam pembangunan sistem Applicant Tracking System (ATS) dimulai dengan mendefinisikan tujuan utama, yaitu mencocokkan kata kunci yang dimasukkan oleh pengguna terhadap isi dokumen CV digital untuk menemukan kandidat yang paling relevan. Langkah pertama yang dilakukan adalah melakukan pra-pemrosesan terhadap seluruh dokumen CV dalam format PDF. Dokumen-dokumen ini dikonversi menjadi bentuk string panjang agar dapat diproses secara in-memory, sehingga pencocokan pola dapat dilakukan secara efisien tanpa perlu membaca file berulang kali. Setelah itu, sistem menerima input dari pengguna berupa daftar kata kunci, pilihan algoritma pencarian (Knuth-Morris-Pratt, Boyer-Moore, atau Aho-Corasick), dan jumlah maksimum hasil pencarian (misalnya, top 10 CV teratas).

Pemecahan masalah dilakukan dengan membagi-bagi permasalahan menjadi permasalahan-permasalahan yang lebih kecil. Pada program ini, diperlukan langkah untuk mengubah data-data cv dalam format file pdf menjadi string dan menyimpannya secara terstruktur supaya mudah diproses. Diperlukan juga langkah pencarian string dengan best match terhadap keyword input(s) dengan pemrosesan string berdasarkan algoritma yang telah dipilih, yaitu KMP, BM, atau AC. Selain itu, juga diperlukan langkah untuk menentukan relevansi atau skor kecocokan dan menampilkan hasil dengan urutan relevansi terbaik. Hasil yang ditampilkan dapat diperluas untuk menampilkan rangkuman ataupun keseluruhan PDF sehingga diperlukan langkah untuk menampilkan informasi penting dari string. Secara rincinya, berikut langkah-langkah pemecahan masalah pada program ini.

1. **Pengolahan File PDF**
   Dalam permasalahan ini, pustaka pdfminer digunakan untuk membaca teks dari file pdf menjadi string. String tersebut disimpan sebagai value pada sebuah dictionary dengan key berupa path ke file tersebut.

2. **Pencocokan Keyword**
   Algoritma pencocokan yang digunakan adalah Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), dan Aho-Corasick (AC). Pencocokan string dilakukan dengan menelusuri seluruh value dictionary, di mana setiap value merupakan hasil ekstraksi teks dari file PDF CV. Untuk setiap value pada dictionary, sistem melakukan pencarian exact matching terhadap setiap keyword menggunakan algoritma yang dipilih. Jika ditemukan, jumlah keyword dihitung sebagai skor relevansi. Jika tidak ditemukan kecocokan exact untuk suatu keyword, sistem akan melakukan pencarian fuzzy matching menggunakan algoritma Levenshtein Distance untuk mengukur kemiripan kata. Semua hasil pencarian diurutkan

berdasarkan skor kecocokan tertinggi. Sistem juga mencatat dan menampilkan waktu eksekusi pencarian exact matching dan fuzzy matching untuk memberikan gambaran performa algoritma yang digunakan.

3. **Penampilan Rangkuman**
   Ketika pengguna memilih salah satu hasil, program mengambil string teks CV dari dictionary yang telah diproses sebelumnya (hasil ekstraksi PDF). Program menggunakan regular expression (regex) untuk mengekstrak informasi penting dari string CV, seperti ringkasan pelamar (summary/overview), keahlian (skills), pengalaman kerja kerja (job experience), dan riwayat pendidikan (education). Selain rangkuman, pengguna juga diberikan opsi untuk membuka dan melihat keseluruhan isi CV dalam format PDF jika diperlukan.

Langkah-langkah ini memastikan bahwa sistem dapat secara efisien mendapatkan CV yang memiliki kriteria yang diinginkan.

3.2 Proses Pemetaan Masalah Menjadi Elemen-Elemen Algoritma KMP, BM, dan AC

3.2.1 Pemetaan Elemen dan Penyelesaian Solusi dengan Algoritma KMP

Algoritma KMP digunakan untuk mencari keberadaan sebuah “word” dalam “text” dengan efisien. Ini bekerja dengan menghindari pengulangan pengecekan karakter yang sudah sesuai. KMP melakukan ini dengan memanfaatkan informasi yang sudah diketahui dari pencocokan sebelumnya melalui longest prefix yang juga merupakan suffix (LPS) untuk menghindari perbandingan yang tidak perlu terjadi secara mismatch.

a. Pemetaan Elemen

   KMP memproses pattern untuk menciptakan sebuah array yang dikenal sebagai LPS (Longest Prefix Suffix). Array LPS ini menyimpan panjang prefix terpanjang dari pattern yang juga merupakan suffix pada setiap posisi. Dengan informasi ini, KMP dapat mengetahui seberapa jauh pattern dapat digeser tanpa harus membandingkan karakter yang sudah pasti cocok.

b. Penyelesaian Solusi

   1. Iterasi Melalui Teks: Algoritma melakukan iterasi pada setiap karakter dalam teks utama (text) dan membandingkannya dengan karakter pada pattern (word) yang dicari.
   2. Pengecekan Kecocokan: Jika karakter pada text dan pattern cocok, indeks pada keduanya akan bertambah. Jika seluruh karakter pattern telah cocok, maka ditemukan satu kemunculan pattern di text.
   3. Handling Mismatch: .Jika terjadi mismatch (karakter tidak cocok), KMP tidak kembali ke awal pattern, melainkan menggunakan nilai pada array

LPS untuk menentukan posisi baru pada pattern yang harus dibandingkan. Ini menghindari perbandingan ulang karakter yang sudah pasti cocok.

4. Kinerja Algoritma: KMP memiliki kompleksitas waktu O(n + m), di mana n adalah panjang text dan m adalah panjang pattern. Hal ini karena setiap karakter pada text dan pattern hanya dibandingkan sekali, sehingga sangat efisien untuk pencarian string dalam teks yang besar.

3.2.2 Pemetaan Elemen dan Penyelesaian Solusi dengan Algoritma BM

Algoritma Boyer-Moore (BM) adalah salah satu algoritma pencarian string yang sangat efisien, terutama untuk pattern yang relatif panjang dan alfabet yang besar. BM bekerja dengan melakukan pencocokan dari kanan ke kiri pada pattern dan menggunakan dua buah tabel utama untuk mempercepat proses pencarian, yaitu bad character rule dan good suffix rule.

a. Pemetaan Elemen
   - Bad Character Table: Tabel ini menyimpan informasi tentang posisi terakhir kemunculan setiap karakter dalam pattern. Jika terjadi mismatch, tabel ini digunakan untuk menentukan seberapa jauh pattern dapat digeser sehingga karakter yang tidak cocok pada text disejajarkan dengan kemunculan terakhir karakter tersebut pada pattern (atau dilewati jika tidak ada).
   - Good Suffix Table: Tabel ini menyimpan informasi tentang substring (suffix) dari pattern yang sudah cocok sebelum terjadi mismatch. Jika terjadi mismatch, tabel ini digunakan untuk menentukan seberapa jauh pattern dapat digeser agar substring yang sudah cocok disejajarkan dengan kemunculan berikutnya pada pattern, atau dengan prefix pattern jika substring tersebut tidak muncul lagi.

b. Penyelesaian Solusi
   1. Iterasi Melalui Teks: Algoritma memulai pencocokan dari kanan ke kiri pada pattern, dan membandingkan dengan substring pada text yang sedang diperiksa.
   2. Pengecekan Kecocokan: Jika karakter pada pattern dan text cocok, pencocokan dilanjutkan ke karakter sebelumnya pada pattern. Jika seluruh karakter pattern cocok, maka ditemukan satu kemunculan pattern di text.
   3. Handling Mismatch: .Jika terjadi mismatch, BM menggunakan nilai minimum dari hasil pergeseran berdasarkan bad character rule dan good suffix rule untuk menentukan seberapa jauh pattern dapat digeser ke kanan. Hal ini memungkinkan BM untuk melewati lebih banyak karakter text dibandingkan algoritma pencarian lain.
   4. Kinerja Algoritma: Boyer-Moore sangat efisien pada praktiknya, terutama untuk pattern yang panjang dan alfabet yang besar. Kompleksitas waktu

rata-rata BM adalah O(n/m), namun dalam kasus terburuk tetap O(n + m), di mana n adalah panjang text dan m adalah panjang pattern.

3.2.3 Pemetaan Elemen dan Penyelesaian Solusi dengan Algoritma AC

Algoritma Aho-Corasick (AC) adalah algoritma pencarian string yang sangat efisien untuk mencari banyak pattern (kata kunci) sekaligus dalam satu kali pemindaian teks. AC membangun struktur automata (finite state machine) dari seluruh pattern yang ingin dicari, sehingga dapat melakukan pencarian semua pattern secara paralel dalam satu lintasan teks.

a. Pemetaan Elemen
   - Trie (Prefix Tree): AC memetakan seluruh pattern ke dalam sebuah struktur trie, di mana setiap node mewakili karakter dari pattern. Setiap jalur dari root ke node tertentu membentuk sebuah pattern yang ingin dicari.
   - Failure Link: Untuk setiap node pada trie, dibuat sebuah failure link yang menunjuk ke node lain yang merupakan fallback jika terjadi mismatch. Failure link ini memungkinkan automata untuk melompat ke posisi yang tepat tanpa harus kembali ke root, sehingga pencarian tetap efisien.
   - Output Link: Setiap node yang merupakan akhir dari sebuah pattern akan menyimpan informasi pattern apa saja yang ditemukan pada posisi tersebut. Ini memungkinkan pencarian banyak pattern sekaligus.

b. Penyelesaian Solusi
   1. Pembangunan Automata: Seluruh pattern dimasukkan ke dalam trie, kemudian failure link dan output link dihitung untuk setiap node. Proses ini dilakukan satu kali sebelum pencarian dimulai.
   2. Iterasi Melalui Teks: Algoritma melakukan iterasi pada setiap karakter dalam teks. Pada setiap langkah, automata berpindah ke node berikutnya sesuai karakter yang dibaca, atau mengikuti failure link jika terjadi mismatch.
   3. Pengecekan Kecocokan: Jika automata berada pada node yang memiliki output (pattern ditemukan), maka pattern tersebut dicatat sebagai hasil pencarian pada posisi saat ini.
   4. Kinerja Algoritma: Kompleksitas waktu pencarian dengan AC adalah O(n + m + z), di mana n adalah panjang teks, m adalah total panjang seluruh pattern, dan z adalah jumlah total kemunculan pattern yang ditemukan. AC sangat efisien untuk pencarian banyak pattern sekaligus dalam teks yang besar.

3.3 Fitur Fungsional dan Arsitektur Aplikasi yang Dibangun

Fitur fungsional utama dari sistem Applicant Tracking System (ATS) yang dikembangkan dalam proyek ini adalah kemampuan untuk melakukan pencarian, ekstraksi, dan visualisasi informasi dari dokumen CV digital berbasis PDF. Sistem ini bekerja dengan cara pertama-tama mengekstrak teks mentah dari file PDF menggunakan modul FileParser, lalu menyimpan hasil ekstraksi tersebut dalam memori (SearchEngine._preprocessed). Setelah itu, sistem memungkinkan pengguna untuk melakukan pencarian kata kunci melalui antarmuka grafis yang interaktif. Pengguna dapat memilih algoritma pencocokan pola seperti Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), atau Aho-Corasick untuk melakukan pencarian exact match, dan jika tidak ditemukan hasil yang cukup relevan, sistem akan menggunakan Levenshtein Distance untuk pencocokan fuzzy.

CV yang relevan akan ditampilkan dalam bentuk kartu hasil (CVCard) yang memuat nama, jumlah kecocokan kata kunci, keahlian, dan pengalaman kerja kandidat. Ketika pengguna mengklik tombol ringkasan (summary) atau detail (view CV), sistem akan menampilkan informasi lengkap pelamar yang diperoleh baik dari database (via get_applicant_profile_by_cv_path) maupun dari hasil ekstraksi teks menggunakan extract_cv_summary. Seluruh informasi ini ditampilkan dalam UI modern berbasis PyQt dengan elemen visual yang memperhatikan estetika dan kenyamanan penggunaan, termasuk animasi dan efek transparansi. Dengan demikian, sistem ini tidak hanya mengotomatisasi proses seleksi awal kandidat, tetapi juga menyajikan informasi penting dengan cara yang terstruktur dan mudah dipahami oleh pengguna.

```
TUBES3_KERJAKERAS/
├── .venv/
├── data/
├── doc/
├── src/
│   ├── database/
│   │   ├── __pycache__/
│   │   ├── __init__.py
│   │   ├── Database.py
│   │   ├── db.py
│   │   ├── Dockerfile
│   │   ├── Encryptor.py
│   │   ├── seed.sql
│   │   └── Seeder.py
│   │
│   ├── extraction/
```

```
│   │   ├── __pycache__/
│   │   ├── __init__.py
│   │   └── extractor.py
│   │
│   ├── FileParser/
│   │   ├── __pycache__/
│   │   ├── __init__.py
│   │   └── FileParser.py
│   │
│   ├── SearchEngine/
│   │   ├── __pycache__/
│   │   ├── __init__.py
│   │   └── SearchEngine.py
│   │
│   └── ui/
│       ├── __pycache__/
│       ├── __init__.py
│       ├── app.py
│       ├── detail_viewer.py
│       ├── result_panel.py
│       ├── search_panel.py
│       └── summary_page.py
├── .gitignore
├── .python-version
├── main.py
├── docker-compose.yml
├── README.md
├── requirements.txt
└── tubes3_seeding.sql
```

3.4 Contoh Ilustrasi Kasus

Misalkan didapat tiga string dari konversi tiga pdf:

CV1	"Experienced software engineer skilled in Java, Python, and SQL. Worked on backend systems and REST APIs."
CV2	"Data analyst with strong background in Excel, Tableau, and Python. Passionate about data visualization."
CV3	"Project manager experienced in Agile methodologies. Familiar with JIRA, Trello, and cross-functional team leadership."

3.4.1 KMP

dengan input:

keywords = ["Python"]

1. Bangun LPS array untuk Python

P	y	t	h	o	n
0	0	0	0	0	0

2. Iterasi text dan pattern
    a. Bandingkan dari indeks i = 0 sampai i = 41
    b. Ketika sampai i = 41, ditemukan text[41:47] = Python
    c. Semua karater cocok maka pattern ditemukan

i	Text[i]	Pattern[j]	Match?	Action
39	,	P	N	i++ karena j=0
40	‘ ‘	P	N	i++ karena j=0
41	P	P	Y	i++, j++
42	y	y	Y	i++, j++
43	t	t	Y	i++, j++
44	h	h	Y	i++, j++
45	o	o	Y	i++, j++
46	n	n	Y	j == pattern.length, maka found

3. Output:

    Pattern ditemukan di indeks ke-41, skor relevansi +1

3.4.2 BM

dengan input:

keywords = ["Python"]

1. Bad Character Table:

P	y	t	h	o	n
0	1	2	3	4	5

2. Good Suffix Table

P	y	t	h	o	n
6	6	6	6	6	6

3. Matching:
    a. Bandingkan substring 6 karakter dari text terhadap pattern dari kanan ke kiri.
    b. Saat text[41:47] = Python, lakukan pengecekan dari belakang pattern:
    c. Pattern ditemukan di posisi 41
4. Output:

    Pattern ditemukan di indeks 41, skor relevansi +1

3.4.3 AC

dengan input:

```
keywords = ["Python", "Java", "SQL"]
```

1. Preprocessing:
    a. Build Trie dari ketiga keyword
    b. Tambahkan Failure Link untuk setiap simpul
    c. Tambahkan Output Link:
        - Misal: simpul terakhir dari “Python” akan menyimpan [“Python”] sebagai output pattern yang cocok
2. Matching:
    a. Iterasi text dari kiri ke kanan (1x pass)
    b. Setiap karakter input menggerakkan automata
        - Saat membaca P → y → t → h → o → n, automata berada di node output ["Python"] → ditemukan match
        - Ulangi proses untuk "Java" dan "SQL" jika ditemukan
3. State Transition

Text Pos	Cur Char	State Transisi	Output
41	P	Go to P Node	-
42	y	Go to y Node	-
43	t	Go to t Node	-
44	h	Go to h Node	-
45	o	Go to o Node	-
46	n	Go to n Node	“Python” ditemukan

4. Output

   Match: “Python” ditemukan di posisi 41

   Jika “Java” dan “SQL” juga ditemukan maka total skor 3

BAB 4 IMPLEMENTASI DAN PENGUJIAN

4.1 Spesifikasi Teknis Program (Struktur Data, Fungsi, dan Prosedur yang Dibangun)

4.1.1 Struktur Data

Struktur Data	**Penjelasan**
CVSummary	Terdiri atas: - summary (string): ringkasan pelamar - skills (list of string): daftar keahlian pelamar - job (list of string/dict): daftar pengalaman kerja pelamar - education (list of string/dict): daftar riwayat pendidikan pelamar
SearchResult	Terdiri atas: - path (Path/string): lokasi file CV - score (int): skor relevansi hasil pencarian
ApplicantProfile	Terdiri atas:

	- applicant_id (int): ID pelamar - first_name (string): nama depan - last_name (string): nama belakang - date_of_birth (string/DateTime): tanggal lahir - address (string): alamat - phone_number (string): nomor telepon - cv_path (string): lokasi file CV
ExperienceEntry	Terdiri atas: - start (string): tahun mulai - end (string): tahun selesai - position (string): jabatan/posisi
EducationEntry	Terdiri atas: - start (string): tahun mulai - end (string): tahun selesai - university (string): nama universitas - degree (string): gelar

4.1.2 Fungsi dan Prosedur

1. Database.py

   Berisi class dan fungsi utama untuk manajemen koneksi, query, dan operasi CRUD ke database MySQL, termasuk integrasi dengan enkripsi data jika diperlukan.

```python
import mysql.connector # harus pertama

class Database:
    def __init__(self, host, port, user, password, database, encryptor):
        self.host = host
        self.port = port
        self.user = user
        self.password = password
        self.database = database
        self.encryptor = encryptor

    def get_connection(self):
        try:
            conn = mysql.connector.connect(
                host=self.host,
                port=self.port,
                user=self.user,
                password=self.password,
                database=self.database
```

```
            )
            return conn
        except Exception as err:
            print(f"Database connection failed: {err}")
            raise

    def encrypt_table_columns(self, table_name, ids, columns,
weak_columns=None):
        weak_columns = weak_columns or []

        conn = self.get_connection()
        cursor = conn.cursor(dictionary=True)

        cursor.execute(f"SELECT * FROM {table_name}")
        records = cursor.fetchall()

        for record in records:
            update_needed = False
            encrypted_values = {}

            for col in columns:
                if col in record and record[col] is not None:
                    if col in weak_columns:
                        encrypted =
self.encryptor.weak_encrypt(str(record[col]))
                    else:
                        encrypted = self.encryptor.encrypt(str(record[col]))
                    encrypted_values[col] = encrypted
                    update_needed = True

            if update_needed:
                set_clause = ", ".join([f"{col} = %s" for col in
encrypted_values])
                where_clause = " AND ".join([f"{id_col} = %s" for id_col in
ids])
                sql = f"UPDATE {table_name} SET {set_clause} WHERE
{where_clause}"
                params = list(encrypted_values.values()) + [record[id_col]
for id_col in ids]
                cursor.execute(sql, params)

        conn.commit()
        cursor.close()
        conn.close()

    def run_query(self, query, params=None, decrypt_fields=None,
weak_fields=None):
        weak_fields = weak_fields or []
        conn = self.get_connection()
        cursor = conn.cursor(dictionary=True)
        try:
```

```
            cursor.execute(query, params or [])

            if query.strip().upper().startswith("SELECT"):
                results = cursor.fetchall()

                if decrypt_fields:
                    for row in results:
                        for field in decrypt_fields:
                            if field in row and row[field] is not None:
                                try:
                                    if isinstance(row[field], str):
                                        if field in weak_fields:
                                            row[field] =
self.encryptor.weak_decrypt(row[field])
                                        else:
                                            row[field] =
self.encryptor.decrypt(row[field])
                                except Exception as e:
                                    print(f"Failed to decrypt field
'{field}':", e)
                return results
            else:
                conn.commit()
                return cursor.rowcount
        finally:
            cursor.close()
            conn.close()

    def decrypt_value(self, encrypted_value, weak=False):
        if weak:
            return self.encryptor.weak_decrypt(encrypted_value)
        return self.encryptor.decrypt(encrypted_value)
```

2. db.py

   Berisi fungsi utilitas sederhana untuk koneksi database, query dasar, dan pengambilan data profil pelamar berdasarkan path CV.

```
import mysql.connector

def get_connection():
    return mysql.connector.connect(
        host="localhost",      # atau "127.0.0.1"
        port=3306,
        user="root",
        password="root",
        database="tubes3db"
    )
```

```python
# Contoh query
def test_connection():
    conn = get_connection()
    cursor = conn.cursor()
    cursor.execute("SHOW TABLES;")
    for (table_name,) in cursor.fetchall():
        print(table_name)
    cursor.close()
    conn.close()

def get_applicant_profile_by_cv_path(cv_path: str):
    conn = get_connection()
    cursor = conn.cursor(dictionary=True)
    query = """
        SELECT ap.first_name, ap.last_name, ap.date_of_birth, ap.address,
ap.phone_number
        FROM ApplicationDetail ad
        JOIN ApplicantProfile ap ON ad.applicant_id = ap.applicant_id
        WHERE ad.cv_path = %s
        LIMIT 1
    """
    cursor.execute(query, (cv_path,))
    result = cursor.fetchone()
    cursor.close()
    conn.close()
    return result


if __name__ == "__main__":
    test_connection()
```

3. Encryptor.py

   Berisi class atau fungsi untuk melakukan enkripsi dan dekripsi data sensitif (misal: nama, nomor telepon) sebelum disimpan ke database atau saat diambil.

```python
import hashlib
import os
import base64

class Encryptor:
    def __init__(self, password: str):
        self.key = hashlib.sha256(password.encode()).digest()

    def _xor_bytes(self, data: bytes) -> bytes:

        return bytes([b ^ self.key[i % len(self.key)] for i, b in
enumerate(data)])
```

```
    def encrypt(self, plaintext: str) -> str:
        data = plaintext.encode()
        iv = os.urandom(16)
        encrypted_data = self._xor_bytes(iv + data)
        return base64.urlsafe_b64encode(encrypted_data).decode()

    def decrypt(self, encrypted_text: str) -> str:
        encrypted_data = base64.urlsafe_b64decode(encrypted_text)
        decrypted = self._xor_bytes(encrypted_data)
        return decrypted[16:].decode()

    def weak_encrypt(self, plaintext: str) -> str:
        data = plaintext.encode()
        xor_data = self._xor_bytes(data)
        encoded = base64.urlsafe_b64encode(xor_data).decode()

        return encoded[:20]

    def weak_decrypt(self, encrypted_text: str) -> str:
        try:
            encrypted_data = base64.urlsafe_b64decode(encrypted_text + '===')
            decrypted = self._xor_bytes(encrypted_data)
            return decrypted.decode()
        except Exception:
            return "[DECRYPTION_ERROR]"
```

4. extractor.py

   Berisi fungsi-fungsi ekstraksi informasi penting dari string hasil ekstraksi PDF, seperti summary, skills, pengalaman kerja, dan pendidikan, biasanya menggunakan regex.

```
import re

skill_sections = [
    "skills", "skill highlights", "summary of skills"
]

experience_sections = [
    "work history", "work experience", "experience",
    "professional experience", "professional history"
]

education_sections = [
    "education", "education and training", "educational background",
    "teaching experience", "corporate experience"
]
```

```
all_sections = skill_sections + experience_sections + education_sections + [
    "summary", "highlights", "professional summary", "core qualifications",
"languages",
    "professional profile", "relevant experience", "affiliations",
"certifications", "qualifications",
    "accomplishments", "additional information", "core accomplishments",
"career overview", "core strengths",
    "interests", "professional affiliations", "online profile",
"certifications and trainings", "credentials",
    "personal information", "career focus", "executive profile", "military
experience", "community service",
    "presentations", "publications", "community leadership positions",
"license", "computer skills",
    "volunteer work", "awards and publications", "activities and honors",
"volunteer associations"
]

def extract_skills(text: str) -> list[str]:
    pattern = f"\n({'|'.join(map(re.escape,
skill_sections))})\n(.*?)(\n({'|'.join(map(re.escape, all_sections))})\n|$)"
    m = re.search(pattern, text, flags=re.IGNORECASE | re.DOTALL)
    if m:
        body = m.group(2).strip()
        skills = [s.strip() for s in re.split(r',|\n|;', body) if s.strip()]
        return skills
    return []


def extract_education(text: str) -> list[str]:
    patterns = [
        r"university of [a-zA-Z ]+",
        r"[a-zA-Z ]+ university",
        r"[a-zA-Z ]+ college",
        r"college of [a-zA-Z ]+",
        r"[a-zA-Z ]* institute of [a-zA-Z ]+",
        r"[a-zA-Z ]+ institute",
        r"[a-zA-Z ]+ high school",
        r"[a-zA-Z ]+ seminary",
        r"[a-zA-Z ]+ center",
        r"[a-zA-Z ]+ training program"
    ]

    education_matches = []
    for pattern in patterns:
        matches = re.findall(pattern, text, flags=re.IGNORECASE)
        education_matches.extend(matches)

    cleaned = list({match.strip() for match in education_matches if
match.strip()})
    return cleaned
```

```
def extract_job(text: str) -> list[str]:
    pattern = f"\n({'|'.join(map(re.escape,
experience_sections))})\n(.*?)(\n({'|'.join(map(re.escape,
all_sections))})\n|$)"
    m = re.search(pattern, flags=re.IGNORECASE | re.DOTALL, string=text)
    if not m:
        return []

    body = m.group(2)
    lines = body.splitlines()
    keywords = [
        "Director", "Manager", "Analyst", "Specialist", "Recruiter",
"Representative",
        "Coordinator", "Lead", "Consultant", "Volunteer", "Assistant",
"Technician",
        "Supervisor", "Associate", "Intern", "Counselor", "Advocate"
    ]
    results = []
    for line in lines:
        for kw in keywords:
            if kw.lower() in line.lower():
                found = re.findall(rf"[A-Z][a-zA-Z ]*{kw}[a-zA-Z ]*", line)
                results.extend(found)
                break
    return list(set(results))


def extract_cv_summary(text: str) -> dict:
    return {
        "skills": extract_skills(text),
        "job": extract_job(text),
        "education": extract_education(text)
    }
```

5. FileParser.py

   Berisi fungsi untuk membaca dan mengekstrak teks dari file PDF menjadi string, menggunakan pustaka seperti pdfminer.

```
from pdfminer.high_level import extract_text
from pathlib import Path

class FileParser:
    @staticmethod
    def GetRawText(file: Path) -> str:
        """
        Fastest way to extract raw text from a PDF using pdfminer.
```

```
        Args:
            file (Path): Path object of the pdf file.

        Returns:
            str: Extracted plain text from the PDF.
        """
        return extract_text(str(file))
```

6. SearchEngine.py

   Berisi class SearchEngine dan metode-metodenya untuk inisialisasi data, pencarian exact/fuzzy, serta penyimpanan hasil preprocessing teks CV dalam dictionary.

```
from typing import List, Dict, Tuple, Callable
from pathlib import Path
from collections import deque
from concurrent.futures import ProcessPoolExecutor
from FileParser import FileParser

class SearchEngine:
    _preprocessed: Dict[Path, str] = {}

    @staticmethod
    def Initialize():
        base_path = Path('../data')
        pdf_files = [file for file in base_path.glob("*/*.pdf") if
file.is_file()]
        with ProcessPoolExecutor() as executor:
            futures = [executor.submit(FileParser.GetRawText, file) for file
in pdf_files]
            for file, future in zip(pdf_files, futures):
                try:
                    text = future.result()
                    SearchEngine._preprocessed[file] = text
                    print(f"PROCESSING {file}.")
                except Exception as e:
                    print(f"Failed to parse {file}: {e}")

    @staticmethod
    def SearchExact(keywords: List[str], type: str, max: int = 5) ->
List[Tuple[Path, int]]:
        match type:
            case "KMP":
                matcher = SearchEngine._build_kmp(keywords)
            case "BM":
                matcher = SearchEngine._build_bm(keywords)
            case "AC":
```

```
                ac_root = SearchEngine._build_ac(keywords)
                matcher = lambda text: SearchEngine._run_ac(text, ac_root)
            case _:
                raise ValueError(f"Invalid algorithm '{type}'. Must be 'KMP',
'BM', or 'AC'.")

        results: List[Tuple[Path, int]] = []
        for path, text in SearchEngine._preprocessed.items():
            count = matcher(text)
            if count > 0:
                results.append((path, count))

        results.sort(key=lambda x: x[1], reverse=True)
        return results[:max]

    @staticmethod
    def SearchFuzzy(keywords: List[str], max_distance: int, max: int = 5) ->
List[Tuple[Path, int]]:
        def levenshtein(a: str, b: str) -> int:
            if len(a) < len(b):
                return levenshtein(b, a)

            if len(b) == 0:
                return len(a)

            previous_row = list(range(len(b) + 1))
            for i, c1 in enumerate(a):
                current_row = [i + 1]
                for j, c2 in enumerate(b):
                    insertions = previous_row[j + 1] + 1
                    deletions = current_row[j] + 1
                    substitutions = previous_row[j] + (c1 != c2)
                    current_row.append(min(insertions, deletions,
substitutions))
                previous_row = current_row

            return previous_row[-1]

        results: List[Tuple[Path, int]] = []
        for path, text in SearchEngine._preprocessed.items():
            words = text.split()
            count = 0
            for keyword in keywords:
                for word in words:
                    if levenshtein(keyword, word) <= max_distance:
                        count += 1
            if count > 0:
                results.append((path, count))

        results.sort(key=lambda x: x[1], reverse=True)
        return results[:max]
```

```
    @staticmethod
    def _build_kmp(keywords: List[str]) -> Callable[[str], int]:
        def _build_lps(pat: str) -> List[int]:
            lps = [0] * len(pat)
            length = 0
            for i in range(1, len(pat)):
                while length and pat[i] != pat[length]:
                    length = lps[length - 1]
                if pat[i] == pat[length]:
                    length += 1
                    lps[i] = length
            return lps

        compiled = [(kw, _build_lps(kw)) for kw in keywords if kw]

        def matcher(text: str) -> int:
            count = 0
            for pat, lps in compiled:
                i = j = 0
                while i < len(text):
                    if text[i] == pat[j]:
                        i += 1
                        j += 1
                        if j == len(pat):
                            count += 1
                            j = lps[j - 1]
                    else:
                        j = lps[j - 1] if j else 0
                        if j == 0 and text[i] != pat[0]:
                            i += 1
            return count

        return matcher

    @staticmethod
    def _build_bm(keywords: List[str]) -> Callable[[str], int]:
        def bad_char_table(pat: str) -> Dict[str, int]:
            return {c: i for i, c in enumerate(pat)}

        def good_suffix_table(pat: str) -> List[int]:
            m = len(pat)
            shift = [0] * (m + 1)
            border = [0] * (m + 1)
            i, j = m, m + 1
            border[i] = j
            while i:
                while j <= m and pat[i - 1] != pat[j - 1]:
                    if shift[j] == 0:
                        shift[j] = j - i
                    j = border[j]
```

```
                i -= 1
                j -= 1
                border[i] = j
            j = border[0]
            for i in range(m + 1):
                if shift[i] == 0:
                    shift[i] = j
                if i == j:
                    j = border[j]
            return shift

        compiled = [(kw, bad_char_table(kw), good_suffix_table(kw)) for kw in
keywords if kw]

        def matcher(text: str) -> int:
            count = 0
            for pat, bad, good in compiled:
                m, n = len(pat), len(text)
                s = 0
                while s <= n - m:
                    j = m - 1
                    while j >= 0 and pat[j] == text[s + j]:
                        j -= 1
                    if j < 0:
                        count += 1
                        s += good[0]
                    else:
                        s += max(j - bad.get(text[s + j], -1), good[j + 1],
1)
            return count

        return matcher

    @staticmethod
    def _build_ac(keywords: List[str]):
        class Node:
            def __init__(self):
                self.children: Dict[str, Node] = {}
                self.fail: 'Node' = None
                self.output: List[str] = []

        root = Node()
        for word in keywords:
            node = root
            for char in word:
                if char not in node.children:
                    node.children[char] = Node()
                node = node.children[char]
            node.output.append(word)

        queue = deque()
```

```
        for child in root.children.values():
            child.fail = root
            queue.append(child)

        while queue:
            current = queue.popleft()
            for char, child in current.children.items():
                fail = current.fail
                while fail and char not in fail.children:
                    fail = fail.fail
                child.fail = fail.children[char] if fail and char in
fail.children else root
                child.output += child.fail.output
                queue.append(child)

        return root

    @staticmethod
    def _run_ac(text: str, root) -> int:
        node = root
        count = 0
        for char in text:
            while node and char not in node.children:
                node = node.fail
            if not node:
                node = root
                continue
            node = node.children[char]
            count += len(node.output)
        return count
```

7. InfoWindow.py

   Berisi komponen UI (PyQt) dari aplikasi

```
from PyQt5.QtWidgets import QDialog, QTextEdit, QVBoxLayout

class InfoWindow(QDialog):
    def __init__(self, title, content, parent=None):
        super().__init__(parent)
        self.setWindowTitle(title)
        self.setMinimumSize(400, 300)

        layout = QVBoxLayout(self)
        text = QTextEdit()
        text.setReadOnly(True)
        if isinstance(content, dict):
            formatted = ""
            for section in ["skills", "job", "education"]:
```

```
                items = content.get(section, [])
                if items:
                    formatted += f"{section.capitalize()}:\n"
                    formatted += "\n".join(f"• {item}" for item in items)
                    formatted += "\n\n"
            text.setPlainText(formatted.strip())
        else:
            text.setPlainText(str(content))
        layout.addWidget(text)
```

4.2 Penjelasan Tata Cara Penggunaan Program

Proses kerja program dimulai dari tahap inisialisasi, di mana sistem akan memuat seluruh file CV dalam format PDF yang terdapat pada folder ../data/. Setiap file akan diproses menggunakan FileParser.GetRawText untuk mengekstrak seluruh isi teks CV, yang kemudian disimpan dalam struktur in-memory berupa dictionary _preprocessed pada kelas SearchEngine. Pada saat yang sama, apabila database belum berisi data awal, file Seeder.py akan dijalankan untuk menanamkan data dari file seed.sql, kemudian seluruh data penting pada tabel ApplicantProfile dan ApplicationDetail akan dienkripsi menggunakan Encryptor, baik dengan metode standar (encrypt) maupun versi ringkas (weak_encrypt) untuk data yang kurang sensitif seperti nomor telepon.

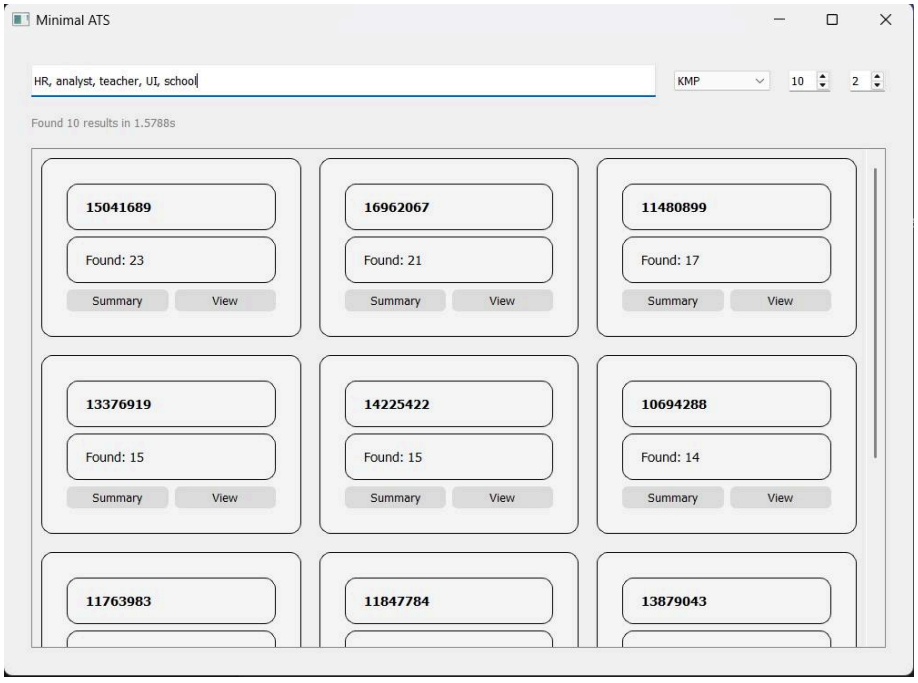
Setelah inisialisasi selesai, pengguna dapat mulai melakukan pencarian kandidat melalui antarmuka grafis (GUI) yang interaktif. Pengguna akan memasukkan satu atau lebih kata kunci (misalnya: "React", "Python", "SQL"), memilih algoritma pencocokan yang diinginkan, antara Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), atau Aho-Corasick, dan menentukan jumlah hasil yang ingin ditampilkan. Ketika tombol pencarian ditekan, sistem akan menggunakan algoritma pattern matching yang dipilih untuk memindai seluruh teks CV yang telah diproses sebelumnya. Proses pencarian dilakukan secara exact match terlebih dahulu menggunakan algoritma yang dipilih, dan jika tidak ditemukan hasil yang relevan, sistem akan secara otomatis menjalankan pencarian fuzzy menggunakan algoritma Levenshtein Distance untuk mengakomodasi kesalahan ketik atau variasi penulisan kata kunci.

Hasil pencarian akan ditampilkan pada panel hasil dalam bentuk kartu CV (CVCard) yang memuat informasi nama, jumlah kecocokan kata kunci, daftar keterampilan, serta pengalaman kerja. Jika data kandidat terdapat dalam basis data, maka informasi personal seperti nama lengkap, tanggal lahir, alamat, dan nomor telepon akan didekripsi secara otomatis melalui fungsi run_query dari kelas Database, yang secara internal menggunakan Encryptor.decrypt untuk membaca data terenkripsi. Pengguna juga dapat melihat ringkasan dari setiap kandidat melalui tombol “Summary”, yang menampilkan informasi yang telah diekstrak seperti daftar keahlian, riwayat pekerjaan, dan pendidikan

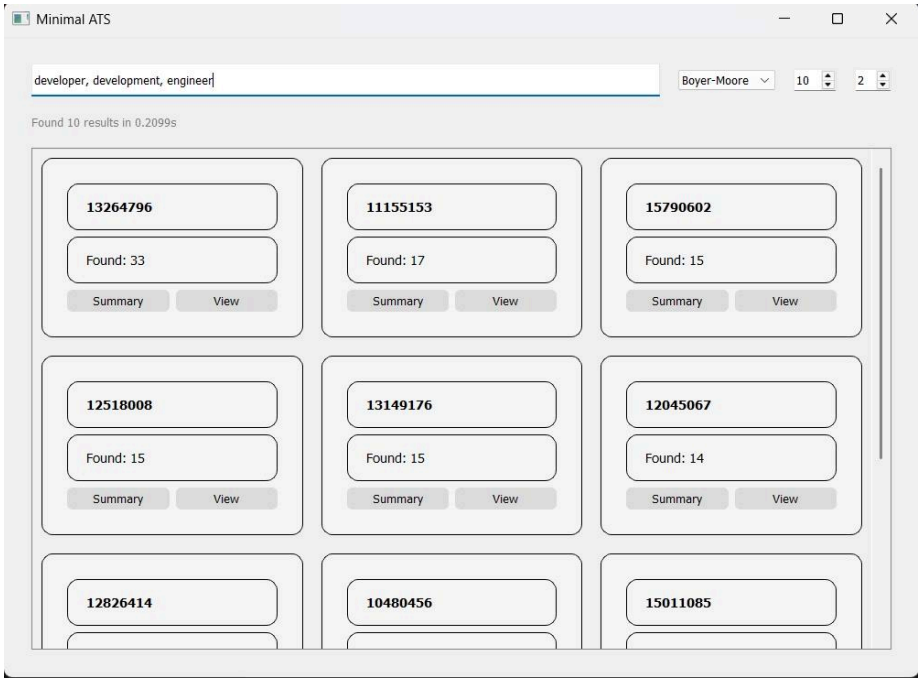
dalam format visual yang terstruktur. Tombol “View CV” juga tersedia untuk melihat tampilan detail dari setiap kandidat. Dengan demikian, seluruh proses dilakukan secara efisien, aman, dan mendukung kecepatan pencarian tinggi melalui pendekatan in-memory dan teknik pattern matching.

4.3 Hasil Pengujian

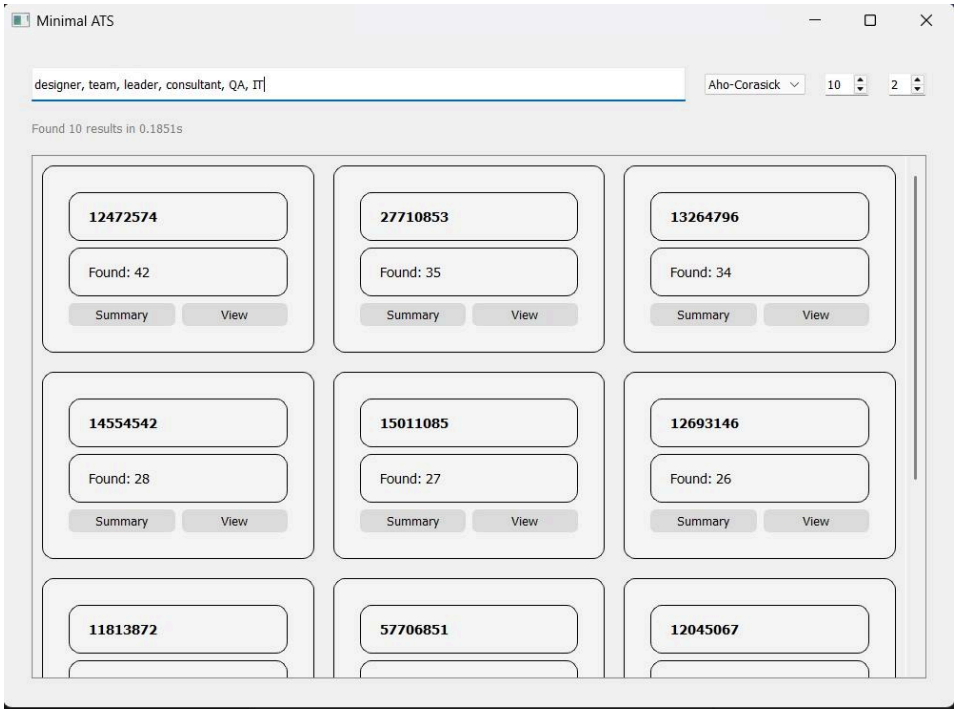
1. Algoritma Knuth-Morris-Pratt, panjang keyword 5, (HR, analyst, teacher, UI, school)



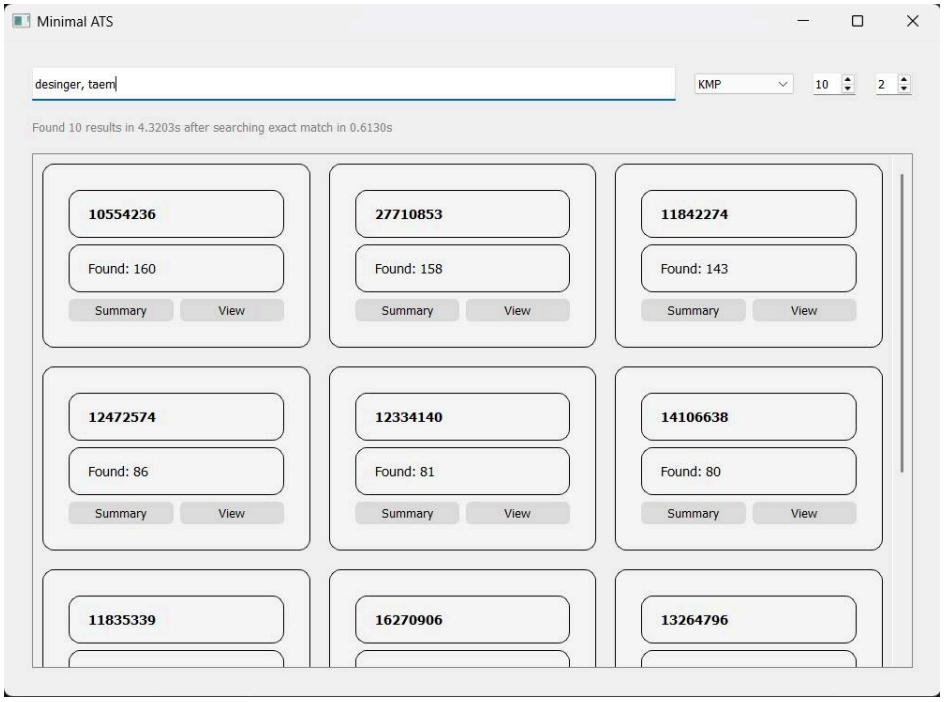
2. Algoritma Boyer-Moore, panjang keyword 3 , (developer, development, engineer)



3. Algoritma Aho-Corasick, panjang keyword 6, (designer, team, leader, consultant, QA, IT)



4. Algoritma Knuth-Morris-Pratt, panjang keyword 2, keyword typo, (desinger, taem)



4.4 Analisis Hasil Pengujian

Dalam sistem ini, diimplementasikan tiga algoritma pencarian string eksak, yaitu Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), dan Aho-Corasick (AC), serta satu algoritma pencarian tidak eksak berbasis Levenshtein Distance untuk kebutuhan fuzzy search. Masing-masing algoritma memiliki karakteristik, kelebihan, dan kekurangan yang membuatnya lebih sesuai untuk kasus-kasus tertentu.

Aho-Corasick (AC) merupakan algoritma paling efisien untuk pencarian multi-kata kunci dalam satu pemindaian teks. Dengan membangun struktur trie dan tabel failure links, AC mampu mencari semua kata kunci sekaligus dalam kompleksitas waktu linier terhadap panjang teks dan total panjang kata kunci. Hal ini menjadikannya sangat cocok untuk kasus seperti penyaringan banyak istilah dalam satu dokumen atau ketika pengguna ingin mencari banyak kata kunci sekaligus. Kelebihan utama AC adalah skalabilitas dan efisiensi saat menangani banyak kata kunci. Namun, kekurangannya adalah kompleksitas implementasi dan konsumsi memori yang lebih tinggi karena perlu membangun automata.

Boyer-Moore (BM) dikenal sebagai salah satu algoritma paling cepat untuk pencarian kata tunggal dalam teks yang besar. Ia bekerja dari belakang pola (kata kunci) dan memanfaatkan dua heuristik utama, yaitu bad character rule dan good suffix rule, untuk melakukan lompatan besar saat terjadi ketidaksesuaian. Kelebihan BM adalah

performanya yang sangat baik dalam praktik, terutama pada teks panjang dengan pola pencarian yang jarang cocok. Namun, BM tidak efisien untuk pencarian multi-kata karena harus dilakukan satu per satu, dan performanya menurun untuk pola yang sangat pendek.

Knuth-Morris-Pratt (KMP) adalah algoritma pencarian berbasis prefix table yang menghindari perbandingan ulang karakter dalam pola. Meskipun secara teoritis efisien (kompleksitas linier terhadap panjang teks dan pola), KMP dalam praktik sering kalah cepat dibanding BM karena tidak memiliki mekanisme lompatan besar saat pencocokan gagal. Kelebihan KMP adalah implementasinya yang relatif sederhana dan deterministik, tetapi kurang efisien untuk pencarian banyak kata atau pada dokumen dengan panjang teks sangat besar.

Sebagai pelengkap, sistem ini juga mengimplementasikan fuzzy search menggunakan algoritma Levenshtein Distance, yang menghitung jumlah operasi edit minimal untuk mengubah satu kata menjadi kata lain. Fuzzy search sangat bermanfaat ketika kata kunci mungkin mengandung kesalahan ketik atau variasi penulisan, dan ketika pencarian eksak gagal menemukan hasil. Keunggulannya adalah fleksibilitas dan kemampuan mendeteksi kemiripan, namun kekurangannya adalah komputasi yang mahal, terutama pada teks panjang dan kumpulan data besar, karena kompleksitasnya kuadratik terhadap panjang kata yang dibandingkan.

Secara keseluruhan, Aho-Corasick merupakan algoritma yang paling efisien dan disarankan untuk kasus pencarian multi-kata kunci pada dokumen besar, terutama dalam konteks sistem ATS (Applicant Tracking System) ini. Boyer-Moore unggul untuk pencarian cepat satu kata dalam teks besar, sedangkan KMP berguna sebagai solusi ringan dan deterministik dalam kasus umum. Fuzzy search sebaiknya digunakan sebagai fallback ketika pencarian eksak tidak menemukan hasil, terutama dalam konteks toleransi kesalahan pengetikan atau variasi bahasa.

BAB 5 KESIMPULAN DAN SARAN

5.1 Kesimpulan

Dalam Tugas Besar 3 ini, kami berhasil membuat sebuah sistem Applicant Tracking System (ATS) berbasis desktop yang dapat menganalisis dan mencocokkan CV digital menggunakan teknik pattern matching dalam bahasa Python. Sistem ini mengimplementasikan:

- Algoritma pencocokan string: Knuth-Morris-Pratt (KMP) Boyer-Moore (BM), dan Aho-Corasick (sebagai bonus)
- Algoritma fuzzy matching: Levenshtein Distance, untuk menangani typo atau variasi kata kunci.
- Ekstraksi data otomatis dari PDF: menggunakan Regex.
- Antarmuka pengguna (UI) desktop yang interaktif dan intuitif.
- Integrasi basis data MySQL untuk menyimpan profil pelamar dan jalur file CV.

Dengan pendekatan ini, sistem dapat mencari kandidat berdasarkan kata kunci dengan efisiensi dan akurasi tinggi, mempercepat proses rekrutmen secara digital, dan memberikan ringkasan informasi kandidat hanya dari CV digital yang diunggah.

Berdasarkan pengujian, didapatkan bahwa Aho-Corasick merupakan algoritma yang paling efisien dan disarankan untuk kasus pencarian multi-kata kunci pada dokumen besar, terutama dalam konteks sistem ATS (Applicant Tracking System) ini. Boyer-Moore unggul untuk pencarian cepat satu kata dalam teks besar, sedangkan KMP berguna sebagai solusi ringan dan deterministik dalam kasus umum. Fuzzy search sebaiknya digunakan sebagai fallback ketika pencarian eksak tidak menemukan hasil, terutama dalam konteks toleransi kesalahan pengetikan atau variasi bahasa.

5.2 Saran

Untuk pengembangan lebih lanjut, kami menyarankan beberapa peningkatan seperti fitur antarmuka pengguna (UI) dapat ditingkatkan dengan menambahkan elemen interaktif yang lebih responsif untuk meningkatkan pengalaman pengguna. Selain itu, eksplorasi lebih lanjut terhadap algoritma lain yang lebih modern dan adaptif dapat membantu dalam meningkatkan efisiensi dan akurasi sistem.

5.3 Refleksi

Pengerjaan tugas ini memberikan wawasan mendalam tentang bagaimana teori algoritma string/pattern matching diterapkan dalam konteks aplikasi nyata. Tantangan terbesar bukan hanya pada penerapan algoritma itu sendiri, tetapi juga dalam merancang aplikasi yang andal, responsif, dan mudah digunakan. Melalui proyek ini, kami belajar pentingnya menggabungkan pengetahuan teknis dengan pemahaman kebutuhan pengguna. Pengalaman ini dapat memperkuat pemahaman kami bahwa efisiensi algoritma dan kejelasan tampilan merupakan hal yang krusial dalam pengembangan aplikasi yang berkualitas.

LAMPIRAN

A. Github

https://github.com/DitaMaheswari05/Tubes3_kerjakeras.git

B. Video

-

C. Tabel Pemeriksaan

	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 *Knuth-Morris-Pratt (KMP)* dan *Boyer-Moore (BM)* dapat menemukan kata kunci dengan benar.	✔	
5	Algoritma Levenshtein Distance dapat mengukur kemiripan kata kunci dengan benar.	✔	
6	Aplikasi dapat menampilkan *summary CV applicant*.	✔	
7	Aplikasi dapat menampilkan *CV applicant* secara keseluruhan.	✔	
8	Membuat laporan sesuai dengan spesifikasi.	✔	

9	Membuat bonus enkripsi data profil *applicant*.	✔	
10	Membuat bonus algoritma Aho-Corasick.	✔	
11	Membuat bonus video dan diunggah pada Youtube.		✖

DAFTAR PUSTAKA

https://www-geeksforgeeks-org.translate.goog/dsa/kmp-algorithm-for-pattern-searching/?_x_tr_sl=en&_x_tr_tl=id&_x_tr_hl=id&_x_tr_pto=tc

https://www-geeksforgeeks-org.translate.goog/dsa/boyer-moore-algorithm-for-pattern-searching/?_x_tr_sl=en&_x_tr_tl=id&_x_tr_hl=id&_x_tr_pto=tc

https://www-geeksforgeeks-org.translate.goog/dsa/aho-corasick-algorithm-pattern-searching/?_x_tr_sl=en&_x_tr_tl=id&_x_tr_hl=id&_x_tr_pto=tc

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/2024-2025/23-Pencocokan-string-(2025).pdf