Laporan Tugas Besar 3

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

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

Disusun oleh:

Muhammad Fariz Rifqi Rizqullah	13523069
Muhammad Edo Raduputu Aprima	13523096
Muhammad Adha Ridwan	13523098

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

2025

Daftar Isi

Bab I
Deskripsi Tugas

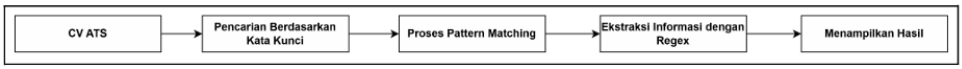
Di era digital ini, keamanan data dan akses menjadi semakin penting. Perkembangan proses rekrutmen tenaga kerja telah mengalami perubahan signifikan dengan memanfaatkan teknologi untuk meningkatkan efisiensi dan akurasi. Salah satu inovasi yang menjadi solusi utama adalah Applicant Tracking System (ATS), 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.

Gambar 1: CV ATS dalam Dunia Kerja (Sumber: Kaldera News)

Sistem ini bertujuan untuk mencocokkan kata kunci dari user terhadap isi CV pelamar kerja dengan pendekatan pattern matching menggunakan algoritma KMP (Knuth-Morris-Pratt) atau BM (Boyer-Moore). Semua proses dilakukan secara in-memory, tanpa menyimpan hasil pencarian—hanya data mentah (raw) CV yang disimpan. Pengguna (HR atau rekruter) akan memberikan input berupa daftar kata kunci yang ingin dicari (misalnya: ”python”, ”react”, dan ”sql”) serta jumlah CV yang ingin ditampilkan (misalnya Top 10 matches).



Gambar 2: Skema Implementasi Applicant Tracking System

Implementasi dari Sistem *Applicant Tracking System* mencakup komponen-komponen utama sebagai berikut:

1. **Ekstraksi Teks CV** – Sistem mampu melakukan ekstraksi teks dari CV dalam format PDF secara otomatis dan mengubahnya menjadi profil pelamar kerja. Setiap file CV dikonversi menjadi satu string panjang yang memuat seluruh teks dari dokumen tersebut.
2. **Pattern Matching** – Implementasi algoritma Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM) untuk melakukan pencarian kata kunci secara exact matching. Algoritma ini memungkinkan sistem untuk mengidentifikasi informasi penting dari CV pelamar dengan kecepatan dan akurasi yang optimal.
3. **Fuzzy Matching** – Apabila tidak ditemukan kecocokan secara persis, sistem melakukan fuzzy matching menggunakan algoritma Levenshtein Distance untuk menangani kesalahan ketik atau perbedaan minor pada input pengguna.
4. **Basis Data** – Sistem menggunakan basis data MySQL untuk menyimpan informasi hasil ekstraksi dari CV yang telah diunggah. Basis data menyimpan profil pelamar beserta lokasi penyimpanan file CV di dalam sistem.
5. **Antarmuka Pengguna** – Aplikasi desktop dengan antarmuka yang intuitif dan menarik, memungkinkan pengguna untuk memasukkan kata kunci pencarian, memilih algoritma pencocokan, dan melihat hasil pencarian secara real-time.
6. **Ekstraksi Informasi** – Menggunakan Regular Expression (Regex) untuk mengekstrak informasi penting dari CV meliputi identitas, ringkasan pelamar, keahlian, pengalaman kerja, dan riwayat pendidikan.
7. **Sistem Ranking** – CV yang ditampilkan diurutkan berdasarkan jumlah kecocokan kata kunci terbanyak, dengan pengguna dapat menentukan jumlah CV yang ingin ditampilkan.
8. **Analisis Performa** – Sistem menampilkan waktu pencarian untuk exact match menggunakan algoritma KMP/BM dan fuzzy match menggunakan Levenshtein Distance secara terpisah.

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 diharapkan terbentuk sebuah sistem yang dapat mengenali profil pelamar secara lengkap hanya dengan menggunakan CV digital.

Bab II
Landasan Teori

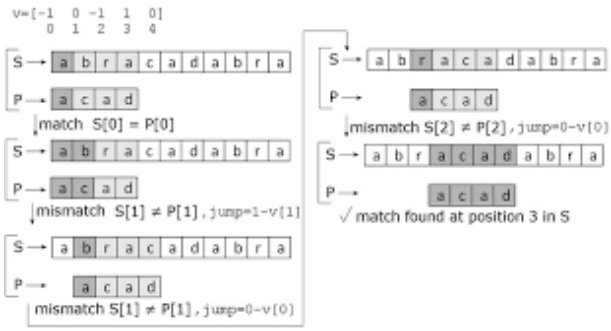
2.1. Algoritma *String Matching*

String matching adalah proses pencarian kemunculan suatu pola (*pattern*) dalam sebuah teks yang lebih panjang. Dalam konteks sistem ATS (*Applicant Tracking System*), algoritma *string matching* digunakan untuk mencari kata kunci tertentu dalam dokumen CV yang telah dikonversi menjadi format teks. Algoritma *string matching* yang efisien sangat penting untuk memproses volume data yang besar dengan waktu respons yang cepat.

Secara umum, algoritma *string matching* memiliki dua komponen utama yaitu *pattern* (P) yang merupakan *string* yang dicari dengan panjang m, dan *text* (T) yang merupakan *string* tempat pencarian dilakukan dengan panjang n. Tujuan utama algoritma *string matching* adalah menemukan semua kemunculan *pattern* P dalam *text* T dengan kompleksitas waktu yang optimal. Algoritma-algoritma klasik seperti *naive string matching* memiliki kompleksitas $O(nm)$, sedangkan algoritma yang lebih canggih seperti KMP dan *Boyer-Moore* dapat mencapai kompleksitas yang lebih baik.

2.2. Knuth-Morris-Pratt (KMP)

Algoritma *Knuth-Morris-Pratt* (KMP) adalah algoritma *string matching* yang dikembangkan oleh Donald Knuth, Vaughan Pratt, dan James Morris pada tahun 1977. Algoritma ini merupakan perbaikan dari algoritma *naive* dengan mengoptimalkan proses pencarian melalui *preprocessing pattern*. KMP menggunakan konsep *failure function* atau *prefix function* untuk menghindari perbandingan karakter yang tidak perlu. *Failure function* $\pi(i)$ menunjukkan panjang *prefix* terpanjang dari *pattern* $P[0...i]$ yang juga merupakan *suffix* dari $P[0...i]$.



Gambar 3: Algoritma Knuth-Morris-Pratt (Sumber: Medium)

Algoritma KMP terdiri dari dua tahapan utama yaitu *preprocessing* yang membangun *failure function* π untuk *pattern* P dengan kompleksitas $O(m)$ dimana m adalah panjang *pattern*, dan *searching* yang melakukan pencarian dengan memanfaatkan informasi dari *failure function* dengan kompleksitas $O(n)$ dimana

n adalah panjang *text*. Total kompleksitas algoritma KMP adalah $O(m+n)$ yang merupakan kompleksitas waktu linear yang optimal.

Keunggulan utama algoritma KMP adalah tidak pernah mundur pada *text* sehingga karakter *text* hanya dibaca sekali, memiliki kompleksitas waktu linear yang optimal, dan sangat cocok untuk pencarian *pattern* yang memiliki struktur berulang. Algoritma ini sangat efektif ketika *pattern* memiliki banyak karakter yang berulang atau ketika terdapat kemungkinan *partial match* yang sering terjadi.

2.3. Boyer-Moore (BM)

Algoritma *Boyer-Moore* adalah algoritma *string matching* yang dikembangkan oleh Robert Boyer dan J Strother Moore pada tahun 1977. Algoritma ini memiliki pendekatan yang berbeda dengan KMP, yaitu melakukan pencarian dari kanan ke kiri pada *pattern*. *Boyer-Moore* menggunakan dua heuristik utama yaitu *bad character heuristic* yang ketika terjadi *mismatch*, algoritma akan mencari karakter yang menyebabkan *mismatch* dalam *pattern*, dan *good suffix heuristic* yang memanfaatkan informasi dari *suffix* yang telah cocok untuk menentukan pergeseran.

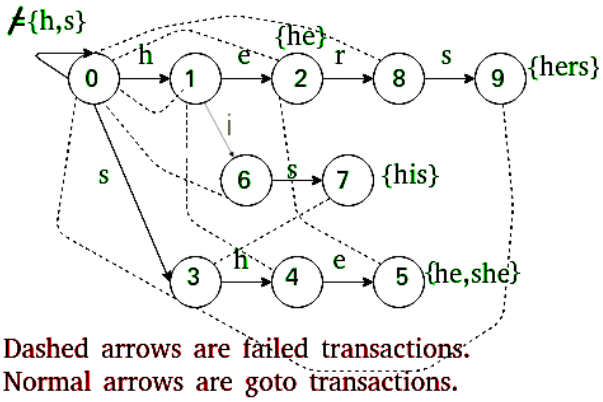
Gambar 4: Algoritma Booyer-Moore (Sumber: StudyGlance)

Algoritma *Boyer-Moore* terdiri dari tahapan *preprocessing* yang membangun *bad character table* dan *good suffix table* dengan kompleksitas $O(m+\sigma)$ dimana σ adalah ukuran alfabet, dan tahapan *searching* yang melakukan pencarian dari kanan ke kiri dengan memanfaatkan kedua heuristik. Kompleksitas pencarian *Boyer-Moore* memiliki *best case* $O(n/m)$, *worst case* $O(nm)$, dan *average case* $O(n)$.

Keunggulan algoritma *Boyer-Moore* adalah sangat efisien untuk *pattern* yang panjang, memiliki performa terbaik pada praktik nyata, dan dapat melakukan "*skip*" karakter dalam jumlah besar. Algoritma ini sangat cocok digunakan untuk pencarian dalam teks yang besar dengan *pattern* yang relatif panjang, seperti pencarian kata kunci dalam dokumen CV yang telah dikonversi menjadi teks.

2.4. Aho-Corasick

Algoritma *Aho-Corasick* adalah algoritma *string matching* yang dikembangkan oleh Alfred Aho dan Margaret Corasick pada tahun 1975. Algoritma ini dirancang khusus untuk *multi-pattern matching*, yaitu pencarian beberapa *pattern* sekaligus dalam satu kali *traversal text*. *Aho-Corasick* menggunakan struktur data *trie* (*prefix tree*) yang diperluas dengan *failure links* untuk menangani *mismatch*. Algoritma ini membangun sebuah *finite automaton* yang dapat mengenali semua *pattern* secara simultan.



Gambar 5: Algoritma Aho-Corasick (Sumber: GeeksForGeeks)

Komponen utama algoritma *Aho-Corasick* meliputi *trie construction* yang membangun *trie* dari semua *pattern*, *failure function* yang menambahkan *failure links* untuk menangani *mismatch*, dan *output function* yang menandai *node* yang merepresentasikan akhir dari suatu *pattern*. Tahapan algoritma terdiri dari *preprocessing* yang membangun *trie* dari semua *pattern*, menghitung *failure function* menggunakan BFS, dan menentukan *output function*, serta tahapan *searching* yang melakukan *traversal text* menggunakan *automaton* yang telah dibangun.

Kompleksitas algoritma *Aho-Corasick* untuk *preprocessing* adalah $O(\Sigma m)$ dimana Σ adalah total panjang semua *pattern*, kompleksitas *searching* adalah $O(n + z)$ dimana z adalah jumlah kemunculan *pattern*, dan kompleksitas ruang adalah $O(\Sigma m \times \sigma)$ untuk representasi *trie*. Keunggulan algoritma ini adalah sangat efisien untuk pencarian *multi-pattern*, memiliki kompleksitas waktu linear terhadap panjang *text*, dan dapat menemukan semua kemunculan dari semua *pattern* dalam satu kali *traversal*.

2.5. Aplikasi ATS (*Applicant Tracking System*)

Applicant Tracking System (ATS) adalah sistem perangkat lunak yang digunakan untuk mengelola proses rekrutmen dan seleksi karyawan. Komponen utama ATS meliputi *resume parsing* untuk ekstraksi informasi dari CV/*resume* dalam format PDF, *keyword matching* untuk pencocokan kata kunci dalam *filtering* kandidat, *database management* untuk penyimpanan dan pengelolaan data kandidat, dan *ranking system* untuk pemeringkatan kandidat berdasarkan kriteria tertentu.

Pada implementasinya, aplikasi ini menggunakan *framework* Flet untuk membangun *user interface* dan menggunakan MySQL sebagai *database management system*. Algoritma *String Matching* seperti *Knuth-Morris-Pratt* dan *Booyer-Moore* serta *Aho-Corasick* digunakan untuk melakukan *exact matching* pada hasil ekstrasi dari CV. Selain itu, jika tidak didapatkan kesamaan, aplikasi akan menggunakan *Levenshtein Distance* untuk melakukan *Fuzzy Matching* pada string hasil ekstrasi. Kemudian akan dilakukan ekstrasi informasi penting untuk menyusun kesimpulan pada setiap CV menggunakan *Regular Expression*.

Bab III
Analisis Pemecahan Masalah

3.1. Langkah Pemecahan Masalah

Untuk mengembangkan aplikasi Sistem ATS berbasic CV Digital, diperlukan beberapa tahapan yang harus diselesaikan secara sistematis:

3.1.1 Analisis Kebutuhan Sistem

Tahap pertama adalah mengidentifikasi kebutuhan sistem yang mencakup:

- Kemampuan ekstraksi teks dari dokumen PDF CV.
- Implementasi algoritma pattern matching untuk pencarian kata kunci.
- Sistem basis data untuk menyimpan profil pelamar dan lamaran.
- Antarmuka pengguna yang intuitif untuk HR/rekruter.
- Sistem ranking berdasarkan tingkat kecocokan kata kunci

3.1.2 Perancangan Arsitektur Sistem

Sistem ATS dirancang dengan arsitektur berlapis yang terdiri dari:

- **Presentation Layer:** Antarmuka pengguna berbasis desktop (GUI).
- **Business Logic Layer:** Implementasi algoritma pattern matching dan fuzzy matching.
- **Data Access Layer:** Koneksi dan operasi basis data MySQL.
- **Data Storage Layer:** Penyimpanan file CV dan basis data lamaran serta profil pelamar

3.1.3 Implementasi Algoritma String Matching

Algoritma *String Matching* yang diimplementasikan meliputi:

- Knuth-Morris-Pratt (KMP) untuk *exact matching*.
- Boyer-Moore (BM) untuk *exact matching*.
- Aho-Corasick (AC) untuk *exact matching*.
- Levenshtein Distance untuk *fuzzy matching*.

3.1.4 Implementasi Sistem Ekstrasi Informasi dari CV Digital

Tahap ini meliputi:

- Konversi PDF ke teks menggunakan *library* Python PyMuPDF.
- Implementasi *Regular Expression* untuk ekstraksi informasi terstruktur.
- Pengolahan dan normalisasi data hasil ekstraksi

3.2. Pemetaan Masalah

Masalah utama pada tugas ini adalah mencari kecocokan kata kunci pada teks hasil ekstraksi dari CV yang telah diekstrak. Sistem ATS yang dikembangkan memerlukan kemampuan untuk melakukan pencarian kata kunci secara efisien dan akurat terhadap ribuan dokumen CV yang telah dikonversi menjadi format teks. Karakteristik masalah ini sangat cocok untuk implementasi algoritma string matching karena melibatkan pencarian pattern (kata kunci) dalam text (isi CV) yang panjang dan tidak terstruktur.

Dalam konteks sistem ATS, setiap CV yang telah melalui proses ekstraksi dari format PDF akan menghasilkan satu string panjang yang berisi seluruh konten tekstual dari dokumen tersebut. String ini mencakup berbagai informasi seperti data pribadi, ringkasan profesional, pengalaman kerja, pendidikan, keahlian, dan informasi lainnya yang relevan. Tantangan utamanya adalah bagaimana melakukan pencarian kata kunci yang diinputkan oleh HR atau rekruter terhadap string panjang ini dengan efisiensi waktu yang optimal dan akurasi yang tinggi.

Permasalahan pencarian kata kunci dalam sistem ATS memiliki beberapa karakteristik khusus yang membuatnya sesuai untuk algoritma string matching. Pertama, ukuran teks yang besar karena setiap CV dapat mengandung ratusan hingga ribuan kata. Kedua, kebutuhan untuk mencari multiple pattern (beberapa kata kunci sekaligus) dalam satu proses pencarian. Ketiga, requirement untuk mendapatkan tidak hanya informasi keberadaan kata kunci, tetapi juga frekuensi kemunculannya untuk keperluan scoring dan ranking CV berdasarkan relevansi.

Masalah lainnya dalam sistem ATS adalah menangani variasi input dari pengguna yang mungkin mengandung kesalahan pengetikan (typo), singkatan yang berbeda, atau variasi penulisan dari kata kunci yang sama. Untuk mengatasi masalah ini, sistem mengimplementasikan *fuzzy matching* menggunakan algoritma *Levenshtein Distance* sebagai *fallback mechanism* ketika *exact matching* tidak menghasilkan kecocokan yang memadai.

Levenshtein Distance mengukur *minimum number of single-character edits* (insertions, deletions, atau substitutions) yang diperlukan untuk mengubah satu string menjadi string lainnya. Dalam konteks sistem ATS, algoritma ini digunakan untuk menghitung tingkat kemiripan antara kata kunci yang diinputkan pengguna dengan kata-kata yang ada dalam teks CV. Proses ini memungkinkan sistem untuk tetap menemukan CV yang relevan meskipun terdapat perbedaan minor dalam penulisan kata kunci.

3.3. Fitur Fungsional dan Arsitektur Aplikasi

3.3.1 Fitur Fungsional

Aplikasi ini menyediakan beberapa fungsionalitas utama yang dirancang untuk memudahkan proses penyaringan kandidat berdasarkan CV mereka.

ID	Fitur Fungsional	Penjelasan
F01	Pengguna dapat menentukan kata kunci pencarian	Pengguna dapat memasukkan satu atau lebih kata kunci (dipisahkan koma) yang relevan dengan kualifikasi yang dicari, seperti keahlian teknis, pengalaman, dll.
F02	Pengguna dapat memilih algoritma pencocokan	Pengguna dapat memilih algoritma pencocokan string yang akan digunakan, yaitu **KMP**, **BM**, atau **AC**.
F03	Pengguna dapat menentukan jumlah hasil teratas	Pengguna dapat menentukan jumlah maksimal kandidat paling relevan yang ingin ditampilkan pada hasil pencarian.
F04	Pengguna dapat memulai proses pencarian dengan satu tombol	Pengguna dapat memulai proses pencocokan CV dengan menekan tombol ”Search”.
F05	Pengguna dapat melihat hasil pencarian dalam bentuk kartu kandidat	Hasil pencarian ditampilkan dalam format *grid* yang berisi kartu-kartu ringkasan untuk setiap kandidat yang cocok.
F06	Pengguna dapat melihat ringkasan detail kandidat	Dengan menekan tombol ”Summary”, pengguna dapat melihat jendela *pop-up* yang berisi informasi lebih detail mengenai profil, keahlian, riwayat pekerjaan, dan pendidikan kandidat.

3.3.2 Arsitektur Aplikasi

Aplikasi ini dibangun sebagai aplikasi desktop menggunakan *framework* **Flet** dengan bahasa pemrograman **Python**. Berikut adalah elemen utama dari arsitektur aplikasi ini:

1. **Antarmuka Pengguna (Flet Framework)**
   Antarmuka Pengguna dibangun sepenuhnya menggunakan Flet. Komponen utama UI didefinisikan dalam kelas `CVApp` pada file `app.py`, yang mencakup:
   - **Panel Kontrol (`_build_left_column`):** Mengelola semua input

dari pengguna, seperti `field` untuk *keyword*, pilihan algoritma, jumlah maksimal hasil yang diinginkan dan juga statistik pencarian seperti *exact time* dan *fuzzy time* dari pencarian yang dilakukan

- **Panel Hasil (`_build_right_column`):** Menampilkan hasil dari pencarian dalam sebuah `GridView` yang berisi komponen kartu dari setiap data.
- **Layer Modal (`_build_modal_layer` dan `_build_tips_modal_layer`):** Mengatur tampilan modal atau *pop-up* untuk *summary* detail CV dan juga modal untuk tips yang berisi bantuan penjelasan cara penggunaan aplikasi.

2. **Logika Aplikasi dan Akses Data (Python & SQLAlchemy)**
   Logika di sisi *backend* diatur dalam beberapa lapisan untuk memisahkan kegunaan dan tanggung jawab:
   - **Data Service (`data_service.py`):** Berguna sebagai penghubung logika UI dengan fungsional *backend*.
   - **Controller (`atsController.py`):** Mengelola alur data.
   - **Repository (`atsRepository.py`):** Bertanggung jawab untuk berinteraksi dengan database melalui *query*.
   - **Model (`models.py`):** Mendefinisikan skema tabel *database* sebagai kelas pada Python.
3. **Mesin Pencocokan CV (*Matcher Engine*)**
   Inti fungsionalitas aplikasi berada di dalam kelas `Matcher` pada file `matcher.py`. Komponen ini bertugas untuk:
   - **Ekstraksi Teks:** Membaca dan mengekstrak konten teks dari file CV berformat PDF menggunakan *library* **PyMuPDF**.
   - **Pencocokan String:** Menjalankan algoritma **KMP**, **BM**, atau **AC** pada teks untuk menemukan kecocokan kata kunci.
4. **Lingkungan Database (Docker & MySQL)**
   Database **MySQL** dijalankan dalam sebuah *container* yang dikelola oleh **Docker Compose**. File `docker-compose.yml` mendefinisikan layanan, sementara `init.sql` bertanggung jawab membuat skema dan mengisi data awal.

Bab IV
Implementasi dan Pengujian

4.1. Implementasi

4.1.1 Objek Kelas

- Kelas CVApp

```
class ATSController:

    def _init_(self):
        # Menginisialisasi controller dan koneksi ke database
        pass

    def create_applicant(self, full_name, email, phone=None,
    ↪  address=None, date_of_birth=None):
        # Membuat applicant baru jika email belum terdaftar
        pass

    def get_applicant(self, applicant_id):
        # Mengambil detail applicant berdasarkan ID
        pass

    def get_applicant_by_email(self, email):
        # Mengambil detail applicant berdasarkan email
        pass

    def get_all_applicants(self, page=1, page_size=50):
        # Mengambil semua data applicant dengan paginasi
        pass

    def search_applicants(self, name_pattern):
        # Mencari applicant berdasarkan pola nama
        pass

    def create_application(self, applicant_id, position,
    ↪  company=None, cv_path=None, application_date=None,
    ↪  status='pending'):
        # Membuat aplikasi untuk seorang applicant
        pass

    def get_application(self, application_id):
        # Mengambil detail aplikasi berdasarkan ID
        pass

    def get_applications_by_applicant(self, applicant_id):
        # Mengambil semua aplikasi untuk applicant tertentu
        pass

    def get_all_applications(self, page=1, page_size=50):
        # Mengambil semua aplikasi dengan paginasi
        pass
```

```
    def get_application_count(self):
        # Mengambil total jumlah aplikasi
        pass

    def get_applicant_count(self):
        # Mengambil total jumlah applicant
        pass

    def get_applicant_with_applications(self, applicant_id):
        # Mengambil detail applicant beserta aplikasinya
        pass

    def get_application_with_applicant(self, application_id):
        # Mengambil aplikasi beserta detail applicant yang
        ↪ terkait
        pass

    def get_dashboard_stats(self):
        # Mengambil statistik dasar mengenai applicant dan
        ↪ aplikasi
        pass

    def get_monthly_application_stats(self, year=None,
    ↪ month=None):
        # Mengambil statistik aplikasi bulanan untuk tahun dan
        ↪ bulan tertentu
        pass

    def test_connection(self):
        # Menguji koneksi database dan mengembalikan jumlah
        ↪ applicant
        pass

    def health_check(self):
        # Melakukan pengecekan kesehatan sistem ATS
        pass
```

- Kelas atsController

```
class ATSController:

    def __init__(self):
        # Menginisialisasi controller dan koneksi ke database
        pass

    def create_applicant(self, full_name: str, email: str,
    ↪ phone: str = None,
                         address: str = None, date_of_birth:
                         ↪ date = None) -> Dict:
        # Membuat applicant baru jika email belum terdaftar
        pass

    def get_applicant(self, applicant_id: int) -> Dict:
        # Mengambil detail applicant berdasarkan ID
        pass
```

```
    def get_applicant_by_email(self, email: str) -> Dict:
        # Mengambil detail applicant berdasarkan email
        pass

    def get_all_applicants(self, page: int = 1, page_size: int
    ↪  = 50) -> Dict:
        # Mengambil semua data applicant dengan paginasi
        pass

    def search_applicants(self, name_pattern: str) -> Dict:
        # Mencari applicant berdasarkan pola nama
        pass

    def create_application(self, applicant_id: int, position:
    ↪  str, company: str = None,
                           cv_path: str = None,
                           ↪  application_date: date = None,
                           status: str = 'pending') -> Dict:
        # Membuat aplikasi untuk seorang applicant
        pass

    def get_application(self, application_id: int) -> Dict:
        # Mengambil detail aplikasi berdasarkan ID
        pass

    def get_applications_by_applicant(self, applicant_id: int)
    ↪  -> Dict:
        # Mengambil semua aplikasi untuk applicant tertentu
        pass

    def get_all_applications(self, page: int = 1, page_size:
    ↪  int = 50) -> Dict:
        # Mengambil semua aplikasi dengan paginasi
        pass

    def get_applicant_count(self) -> Dict:
        # Mengambil total jumlah applicant

    def get_application_count(self) -> Dict:
        # Mengambil total jumlah aplikasi
        pass
pass

    def get_applicant_with_applications(self, applicant_id:
    ↪  int) -> Dict:
        # Mengambil detail applicant beserta aplikasinya
        pass

    def get_application_with_applicant(self, application_id:
    ↪  int) -> Dict:
        # Mengambil aplikasi beserta detail applicant yang
        ↪  terkait
        pass

    def get_dashboard_stats(self) -> Dict:
        # Mengambil statistik dasar mengenai applicant dan
        ↪  aplikasi
```

```
        pass

    def get_monthly_application_stats(self, year: int = None,
    ↪  month: int = None) -> Dict:
        # Mengambil statistik aplikasi bulanan untuk tahun dan
        ↪  bulan tertentu
        pass

    def test_connection(self) -> Dict:
        # Menguji koneksi database dan mengembalikan jumlah
        ↪  applicant
        pass

    def health_check(self) -> Dict:
        # Melakukan pengecekan kesehatan sistem ATS
        pass
```

- Kelas DataService

```
class DataService:
    def __init__(self):
        # Menginisialisasi controller ATS dan matcher untuk
        ↪  pencarian kandidat

    def get_all_text(self) -> str:
        # Mengambil semua teks dari aplikasi yang ada untuk
        ↪  pencocokan kata kunci

    def get_total_cvs(self):
        # Mengambil total aplikasi dari dashboard stats

    def search_candidates(self, keywords: list, top_n: int,
    ↪  algorithm: str):
        # Mencari kandidat berdasarkan kata kunci dan
        ↪  algoritma pencocokan
```

- Kelas BaseRepository

```
class BaseRepository:

    def __init__(self):
        # Initializes the database configuration and session
        ↪  maker
        pass

    @contextmanager
    def get_session(self):
        # Provides a session for database interaction
        pass
```

- Kelas ApplicantRepository

```
class ApplicantRepository(BaseRepository):

    def create_applicant(self, applicant_data: Dict) ->
    ↪  Optional[ApplicantProfile]:
        # Creates a new applicant and saves it to the database
        pass

    def get_applicant_by_id(self, applicant_id: int) ->
    ↪  Optional[ApplicantProfile]:
        # Retrieves an applicant details by ID
        pass

    def get_applicant_by_email(self, email: str) ->
    ↪  Optional[ApplicantProfile]: ->
        # Retrieves an applicant details by email
        pass

    def get_all_applicants(self, limit: Optional[int] = None,
    ↪  offset: Optional[int] = None) ->
    ↪  List[ApplicantProfile]:
        # Retrieves all applicants with pagination support
        pass

    def delete_applicant(self, applicant_id: int) -> bool:
        # Deletes an applicant by their ID
        pass

    def search_applicants_by_name(self, name_pattern: str) ->
    ↪  List[ApplicantProfile]:
        # Searches for applicants by name pattern
        pass

    def get_applicants_count(self) -> int:
        # Retrieves the total count of applicants in the
        ↪  database
        pass
```

- Kelas ApplicationRepository

```
class ApplicationRepository(BaseRepository):

    def create_application(self, application_data: Dict) ->
    ↪  Optional[ApplicationDetail]:
        # Creates a new application for an applicant and saves
        ↪  it to the database
        pass

    def get_application_by_id(self, application_id: int) ->
    ↪  Optional[ApplicationDetail]:
        # Retrieves an application's details by ID
        pass

    def get_applications_by_applicant(self, applicant_id: int)
    ↪  -> List[ApplicationDetail]:
```

```
        # Retrieves all applications for a specific applicant
        pass

    def get_all_applications(self, limit: Optional[int] =
    ↪  None, offset: Optional[int] = None) ->
    ↪  List[ApplicationDetail]:
        # Retrieves all applications with pagination support
        pass

    def delete_application(self, application_id: int) -> bool:
        # Deletes an application by its ID
        pass

    def get_applications_count(self) -> int:
        # Retrieves the total count of applications in the
        ↪  database
        pass
```

- Kelas ApplicantProfile

```
class ApplicantProfile(Base):
    __tablename__ = 'ApplicantProfile's

    def __repr__(self):
        # Menyediakan representasi string yang mudah dibaca
        ↪  untuk profil applicant
        pass

    def to_dict(self):
        # Mengonversi objek profil applicant menjadi format
        ↪  dictionary
        pass
```

- Kelas ApplicationDetail

```
class ApplicationDetail(Base):
    __tablename__ = 'ApplicationDetail'

    def __repr__(self):
        # Menyediakan representasi string yang mudah dibaca
        ↪  untuk detail aplikasi
        pass

    def to_dict(self):
        # Mengonversi objek detail aplikasi menjadi format
        ↪  dictionary
        pass
```

- Kelas DatabaseConfig

```
class DatabaseConfig:
    def __init__(self):
        # Menginisialisasi konfigurasi database dengan nilai
        ↪  default atau variabel lingkungan
```

```
        pass

    def get_engine(self):
        # Mengembalikan engine SQLAlchemy untuk menghubungkan
        ↪ ke database
        pass

    def get_session_maker(self):
        # Mengembalikan sessionmaker untuk membuat sesi
        ↪ database
        pass
```

- Kelas AhoCorasick

```
class AhoCorasick:
    """
    Implementasi algoritma Aho-Corasick untuk pattern matching
    ↪ multiple strings.
    Digunakan untuk mencari beberapa pattern sekaligus dalam
    ↪ satu text.
    """

    def _init_(self, words):
        """
        Inisialisasi struktur data untuk Aho-Corasick
        ↪ automaton.
        """
        # Inisialisasi max_states, max_characters, output
        ↪ function, failure function, goto function
        # Konversi semua words ke lowercase untuk
        ↪ case-insensitive search
        pass

    def __build_matching_machine(self):
        """
        Membangun finite state automaton untuk pattern
        ↪ matching.
        Membuat trie structure dan menghitung failure
        ↪ function.
        """
        # Membangun trie dari semua keywords
        # Menghitung failure function menggunakan
        ↪ breadth-first search
        pass

    def __find_next_state(self, current_state, next_input):
        """
        Mencari state selanjutnya berdasarkan input character.
        Menggunakan goto function dan failure function.
        """
        # Menentukan state transition berdasarkan input
        ↪ character
        pass

    def search_words(self, text) -> Dict:
        """
```

```
        Mencari semua occurrences dari semua keywords dalam
        ↪ text.
        Returns dictionary dengan hasil matching untuk setiap
        ↪ keyword.
        """
        # Traverse text menggunakan automaton dan hitung
        ↪ matches untuk setiap keyword
        pass
```

- Kelas Matcher

```
class Matcher:
    """
    Class utama untuk melakukan text matching dengan berbagai
    ↪ algoritma.
    Mendukung exact matching, fuzzy matching, dan pattern
    ↪ matching algorithms.
    """

    def __init__(self, sources: List[Tuple[str, str]],
    ↪ queries: List[str]):
        """
        Inisialisasi matcher dengan source files dan queries.
        """
        # Inisialisasi source paths, queries, dan extract text
        ↪ dari semua files
        pass

    def extract_text(self, path: str, case: int) -> str:
        """
        Mengextract text dari file (currently supports PDF).
        case: 0 untuk lowercase, 1 untuk original case.
        """
        # Extract text dari PDF menggunakan PyMuPDF (fitz)
        # Clean text dengan regex dan convert case sesuai
        ↪ parameter
        pass

    def _extract_texts_concurrently(self) -> List[str]:
        """
        Extract text dari multiple files secara concurrent
        ↪ menggunakan ThreadPoolExecutor.
        """
        # Menggunakan concurrent.futures untuk parallel text
        ↪ extraction
        pass

    def set_keywords(self, queries: List[str]):
        """
        Set keywords baru untuk matching dan reset semua
        ↪ calculations.
        """
        # Update queries list dan reset timing calculations
        pass
```

```
    def _get_important_information(self, text: str) ->
    ↪  Dict[str, Union[str, List[str]]]:
        """
        Extract informasi penting dari CV text menggunakan
        ↪  regex patterns.
        """
        # Extract nama, tanggal lahir, alamat, email, phone,
        ↪  skills, education, experience
        pass

    def _extract_from_pdf(self, path: str) -> str:
        """
        Extract text dari PDF file menggunakan PyMuPDF.
        """
        # Buka PDF dan extract text dari semua pages
        pass

    def match(self, method: str, threshold: float = 0.7) ->
    ↪  Dict:
        """
        Melakukan text matching menggunakan method yang
        ↪  dipilih.
        Supports: 'exact', 'KMP', 'BM', 'AC', 'fuzzy'
        """
        # Pilih algoritma matching berdasarkan method
        ↪  parameter
        # Untuk exact matches yang gagal, gunakan fuzzy
        ↪  matching sebagai fallback
        pass

    def _exact_match_1_query(self, text: str, query: str) ->
    ↪  int:
        """
        Melakukan exact string matching untuk satu query
        ↪  menggunakan built-in find().
        """
        # Menggunakan string.find() untuk mencari semua
        ↪  occurrences
        pass

    def _exact_match(self, text: str, queries: List[str]) ->
    ↪  Dict:
        """
        Melakukan exact matching untuk semua queries.
        """
        # Iterate semua queries dan hitung total matches
        pass

    def _KMP_match_1_query(self, text: str, query: str) ->
    ↪  int:
        """
        Implementasi algoritma Knuth-Morris-Pratt untuk
        ↪  pattern matching.
        Menggunakan prefix function untuk efficient string
        ↪  searching.
        """
        # Implementasi KMP algorithm dengan prefix function
```

```
        pass

    def _KMP_match(self, text: str, queries: List[str]) ->
    ↪  Dict:
        """
        Melakukan KMP matching untuk semua queries.
        """
        # Iterate semua queries menggunakan KMP algorithm
        pass

    def _BM_match_1_query(self, text: str, query: str) -> int:
        """
        Implementasi algoritma Boyer-Moore untuk pattern
        ↪  matching.
        Menggunakan bad character heuristic untuk skip
        ↪  characters.
        """
        # Implementasi Boyer-Moore algorithm dengan bad
        ↪  character table
        pass

    def _BM_match(self, text: str, queries: List[str]) ->
    ↪  Dict:
        """
        Melakukan Boyer-Moore matching untuk semua queries.
        """
        # Iterate semua queries menggunakan Boyer-Moore
        ↪  algorithm
        pass

    def _fuzzy_match(self, text: str, queries: List[str],
    ↪  threshold: float) -> Dict:
        """
        Melakukan fuzzy matching untuk semua queries dengan
        ↪  threshold tertentu.
        Menggunakan Levenshtein distance untuk menghitung
        ↪  similarity.
        """
        # Implement fuzzy matching menggunakan similarity
        ↪  threshold
        pass
```

4.1.2 Fungsi dan Prosedur

- Fungsi levenshtein-distance

```
def levenshtein_distance(s1: str, s2: str) -> int:
    """
    Menghitung jarak Levenshtein antara dua string menggunakan
    ↪ dynamic programming.
    Jarak Levenshtein adalah jumlah minimum operasi edit yang
    ↪ diperlukan.
    """
    # Implementasi dynamic programming untuk menghitung edit
    ↪ distance
    pass
```

- Fungsi calculate-similarity

```
def calculate_similarity(str1: str, str2: str) -> float:
    """
    Menghitung tingkat kesamaan antara dua string berdasarkan
    ↪ jarak Levenshtein.
    Returns nilai antara 0.0 - 1.0 dimana 1.0 berarti identik.
    """
    # Menggunakan rumus: (panjang_terpanjang -
    ↪ jarak_levenshtein) / panjang_terpanjang
    pass
```

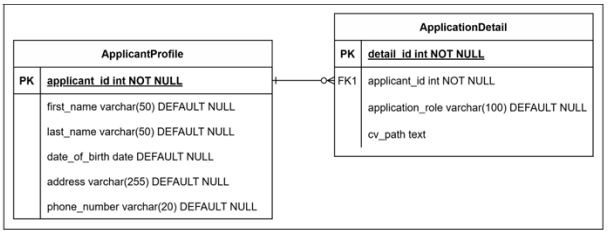
- Fungsi fuzzy-match-1-query

```
def fuzzy_match_1_query(text: str, query: str, threshold:
↪ float) -> int:
    """
    Melakukan fuzzy matching untuk satu query terhadap text
    ↪ dengan threshold tertentu.
    Membandingkan setiap kata dalam text dengan query.
    """
    # Memisahkan text menjadi kata-kata dan bandingkan dengan
    ↪ query menggunakan similarity
    pass
```

- Fungsi fuzzy-match-worker

```
def fuzzy_match_worker(j, i, text, query, threshold):
    """
    Worker function untuk multiprocessing fuzzy matching.
    Digunakan untuk parallel processing pada multiple
    ↪ documents.
    """
    # Memanggil fuzzy_match_1_query dan mengembalikan hasil
    ↪ dengan index
    pass
```

4.1.3 Skema Database



Gambar 6: Skema Database (Sumber: Spesifikasi)

- Tabel ApplicantProfile

```
CREATE TABLE ApplicantProfile (
    applicant_id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    date_of_birth DATE,
    address VARCHAR(255),
    phone_number VARCHAR(20)
)ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

- Tabel ApplicantProfile

```
CREATE TABLE ApplicationDetail (
    detail_id INT AUTO_INCREMENT PRIMARY KEY,
    applicant_id INT NOT NULL,
    application_role VARCHAR(100),
    cv_path TEXT,
    FOREIGN KEY (applicant_id) REFERENCES
    ↪  ApplicantProfile(applicant_id)
)ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

4.1.4 Library

- Flet.
- Concurrent.
- SQLAlchemy.
- dotenv.
- os.
- typing.

4.2. Penggunaan Aplikasi

4.2.1 Run Aplikasi

1. Clone Repository

   `git clone https://github.com/Fariz36/Tubes3_10Internship0Job.git`

2. Buka folder src

   `cd src`

3. Sync Dependencies

   `uv sync`

4. Jalankan Database Docker

   (Windows)

   Setelah menjalankan Docker Dekstop

   run `cd database`

   `docker compose up -d`

   (Linux)

   buka folder Database, `cd database`

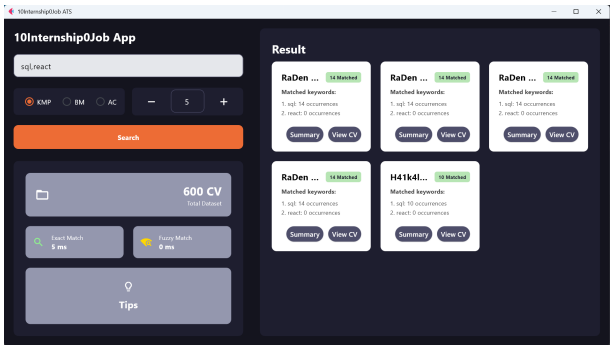
   `sudo docker-compose up -d`

5. Jalankan Aplikasi

   Kembali ke folder src

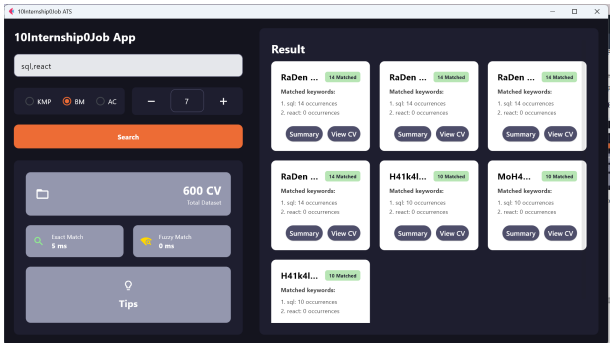
   `uv run ./app/app.py`

4.2.2 Pencarian dengan Knuth-Morris-Pratt



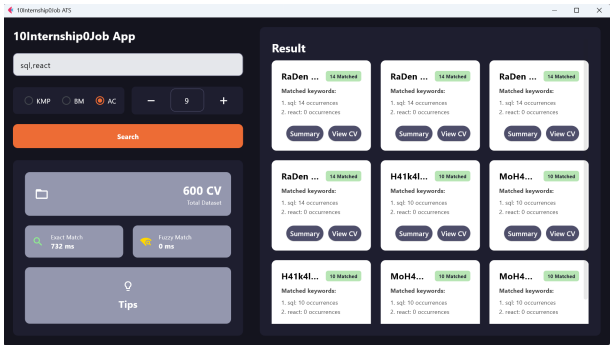
Gambar 7: Pencarian dengan KMP (Sumber: Koleksi Pribadi)

4.2.3 Pencarian dengan Booyer-Moore



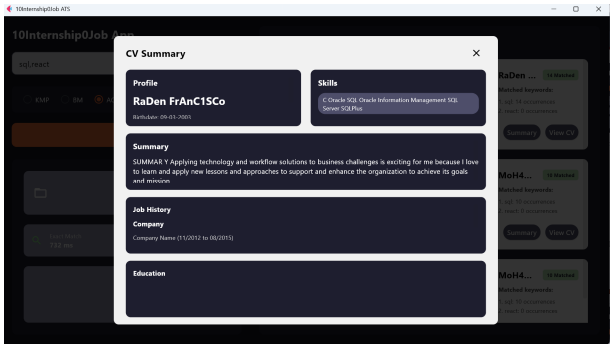
Gambar 8: Pencarian dengan BM (Sumber: Koleksi Pribadi)

4.2.4 Pencarian dengan Aho-Corasick



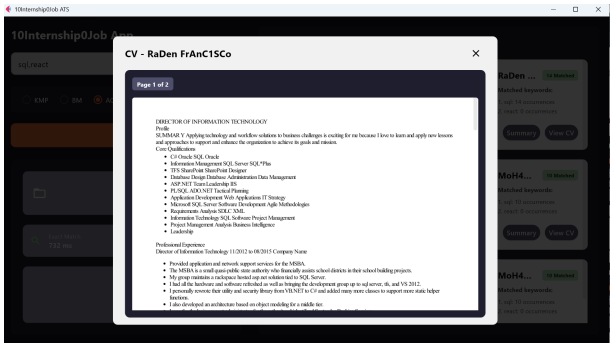
Gambar 9: Pencarian dengan AC (Sumber: Koleksi Pribadi)

4.2.5 Popup CV Summary



Gambar 10: CV Summary (Sumber: Koleksi Pribadi)

4.2.6 Popup View CV



Gambar 11: CV View (Sumber: Koleksi Pribadi)

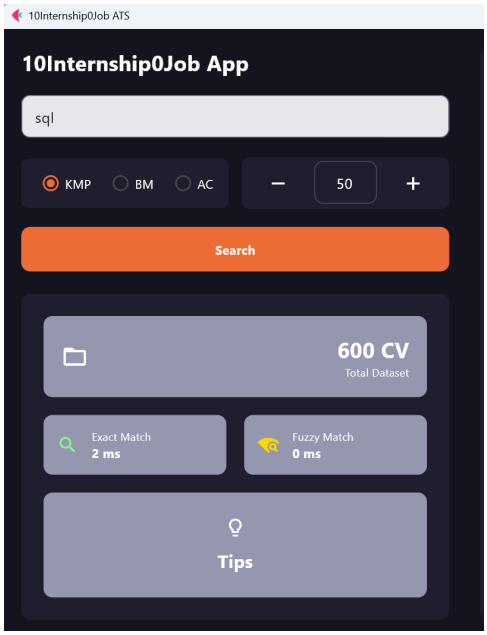
4.3. Pengujian

Berikut hasil pengujian dari berbagai tes yang diujikan, mulai *test-case* umum hingga *stress-case* untuk menilai performa program, dengan *benchmark* 50 CV.

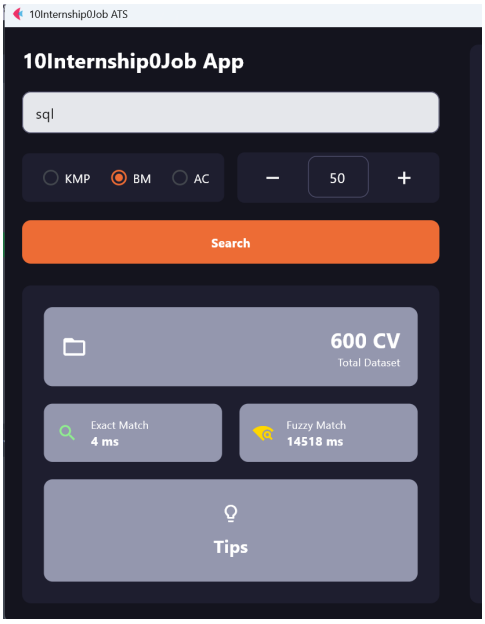
a. Test Case Basic Single

`keyword-basic-single.txt`

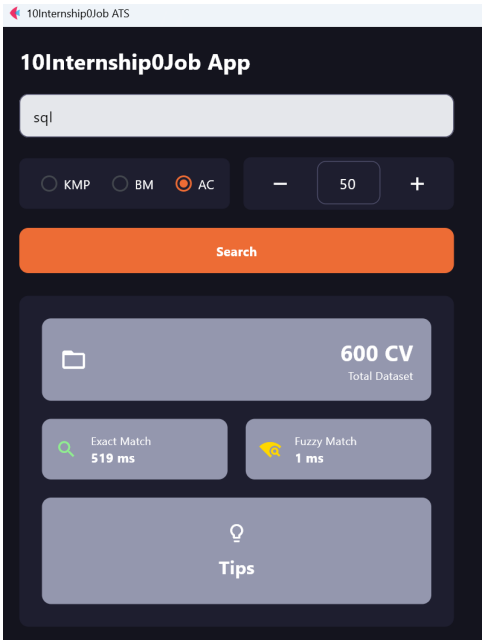
```
sql
```



Gambar 12: Pengujian KMP pada TC `keyword-basic-single.txt`



Gambar 13: Percobaan BM pada TC `keyword-basic-single.txt`

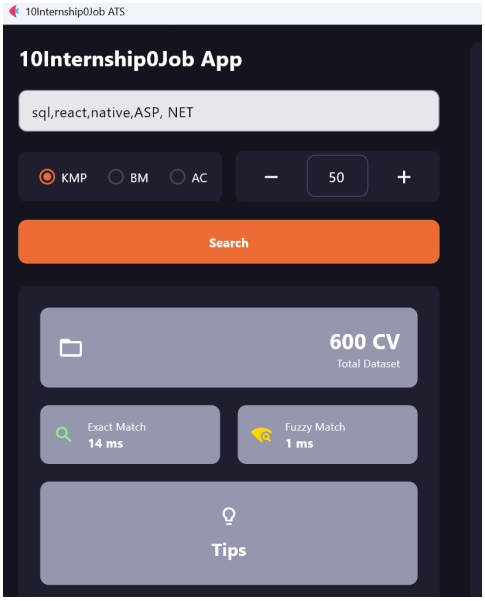


Gambar 14: Percobaan AC pada TC `keyword-basic-single.txt`

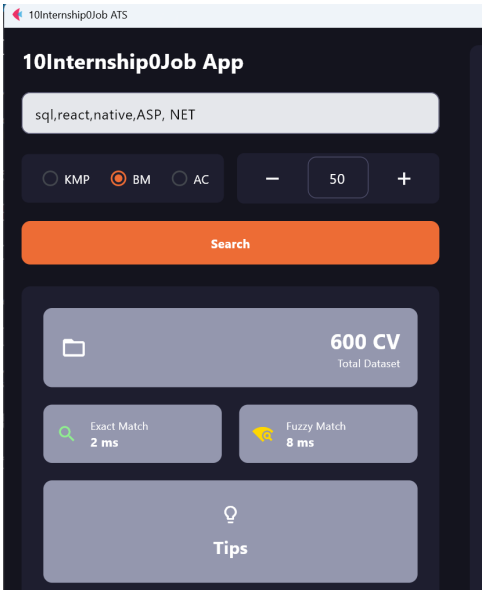
b. Test Case Basic Multiple

`keyword-basic-multiple.txt`

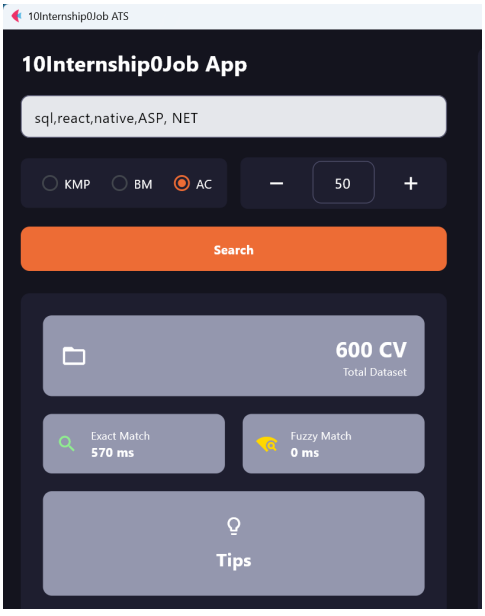
```
sql,react,native,ASP, NET
```



Gambar 15: Pengujian KMP pada TC `keyword-basic-multiple.txt`



Gambar 16: Pengujian BM pada TC `keyword-basic-multiple.txt`

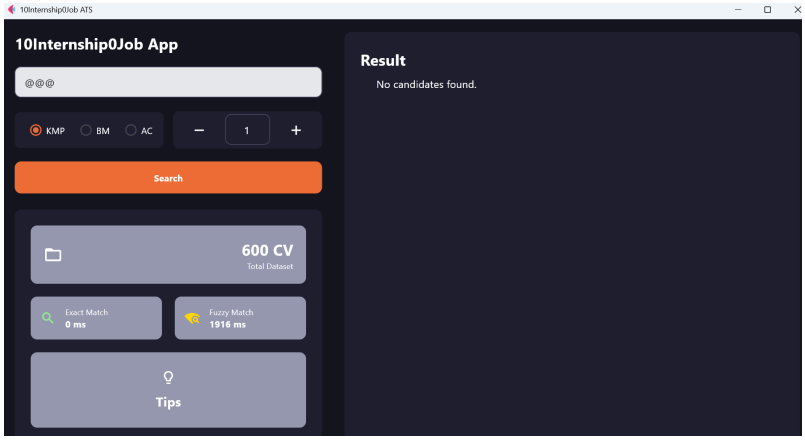


Gambar 17: Pengujian AC pada TC `keyword-basic-multiple.txt`

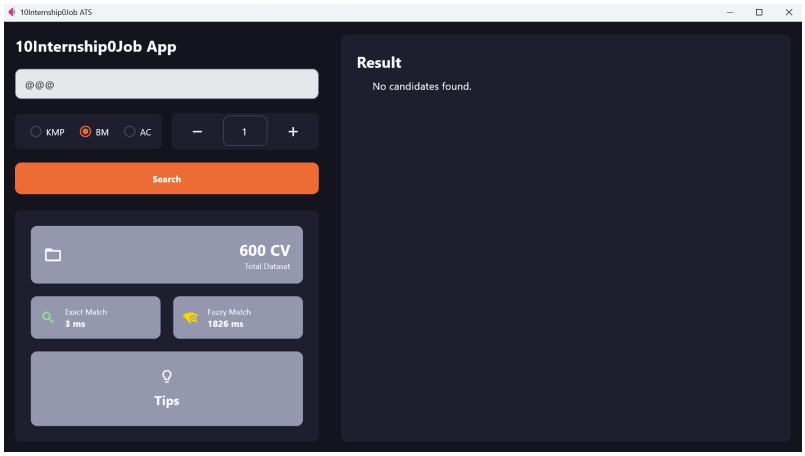
c. Test Case Special Keyword

`special-keyword-single.txt`

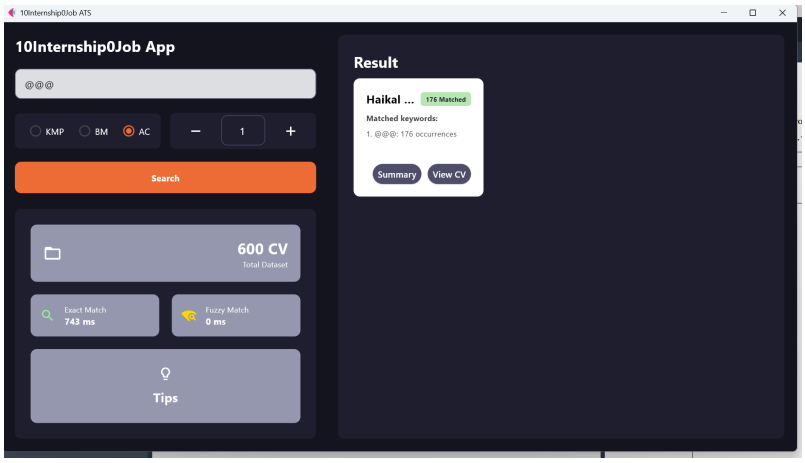
```
@@@
```



Gambar 18: Pengujian KMP pada TC `keyword-special-single.txt`



Gambar 19: Pengujian BM pada TC `keyword-special-single.txt`

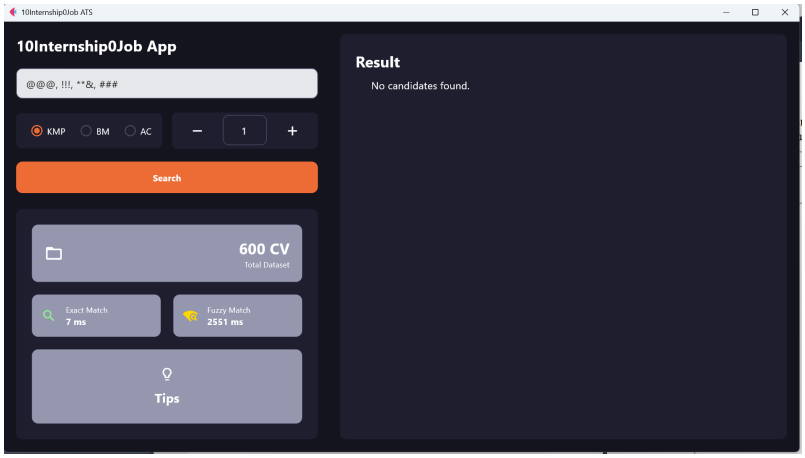


Gambar 20: Pengujian AC pada TC `keyword-special-single.txt`

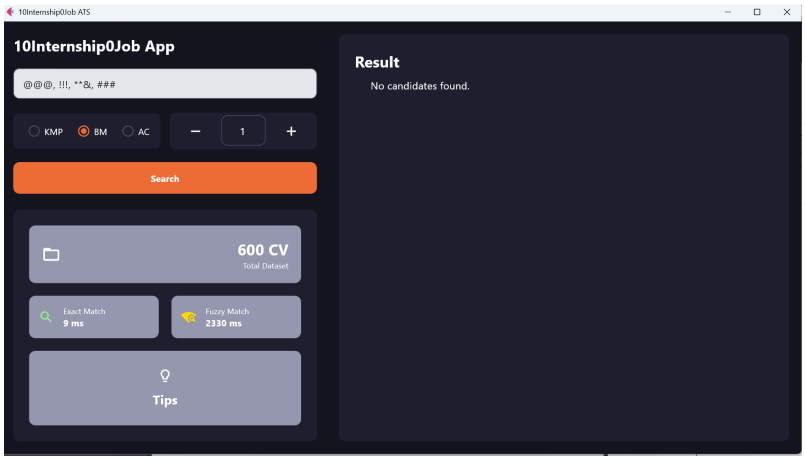
d. Test Case Special Multiple

`keyword-special-multiple.txt`

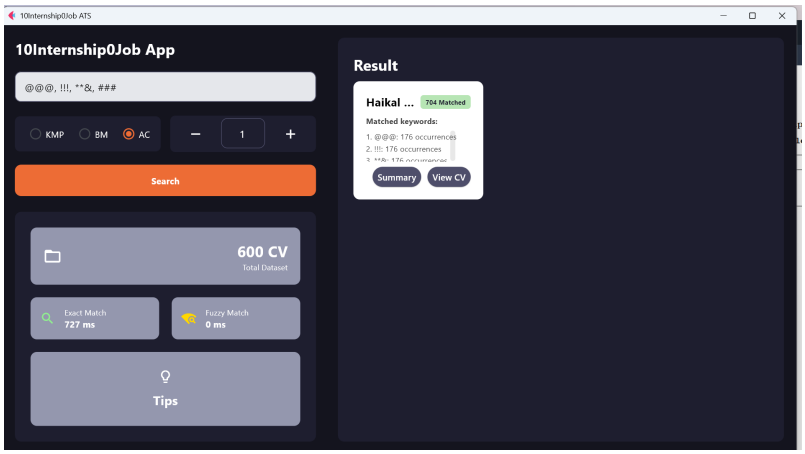
```
@@@, !!!, **&, ###
```



Gambar 21: Pengujian KMP pada TC `keyword-special-multiple.txt`



Gambar 22: Pengujian BM pada TC `keyword-special-multiple.txt`



Gambar 23: Pengujian AC pada TC `keyword-special-multiple.txt`

4.4. Analisis dan Pembahasan

Dalam pengembangan sistem pelacakan pelamar (ATS) ini, pemilihan algoritma pencocokan string menjadi krusial untuk memastikan kecepatan dan akurasi dalam memindai dokumen CV. Program ini mengimplementasikan dua algoritma *exact matching* utama, yaitu `Knuth-Morris-Pratt (KMP)` dan `Boyer-Moore (BM)`. Sebagai fitur bonus, diimplementasikan pula algoritma `Aho-Corasick` yang dirancang khusus untuk pencarian multi-kata kunci secara simultan.

Selain itu, untuk menangani kasus ketidakcocokan persis seperti kesalahan pengetikan (*typo*), digunakan algoritma *fuzzy matching* `Levenshtein Distance`. Untuk ekstraksi informasi terstruktur dari teks CV, seperti riwayat pendidikan dan pengalaman kerja, sistem memanfaatkan `Regular Expression (Regex)`.

Berdasarkan implementasi, `Boyer-Moore` secara umum menunjukkan performa yang lebih cepat untuk pencarian satu kata kunci (pattern) yang relatif panjang. `KMP` menawarkan performa yang lebih konsisten, terutama jika kata kunci memiliki pola internal yang berulang. Namun, ketika pengguna mencari banyak kata kunci sekaligus, `Aho-Corasick` menjadi jauh lebih unggul karena dapat menemukan semua kata kunci dalam satu kali proses pembacaan teks.

Analisis lebih lanjut untuk tiap algoritma, dipaparkan sebagai berikut:

4.4.1 Knuth-Morris-Pratt (KMP)

Algoritma `Knuth-Morris-Pratt` melakukan pencocokan string dengan menghindari perbandingan yang berulang pada teks. Kunci dari KMP adalah tahap pra-pemrosesan pada *pattern* (kata kunci) untuk membangun sebuah tabel ***Longest Proper Prefix which is also Suffix*** (LPS). Tabel LPS ini menyimpan panjang prefiks terpanjang yang juga merupakan sufiks untuk setiap bagian dari *pattern*. Saat terjadi ketidakcocokan, KMP menggunakan tabel LPS untuk menggeser *pattern* secara cerdas tanpa perlu menggerakkan pointer pada teks ke belakang. Hal ini membuatnya sangat efisien. Dalam konteks ATS, KMP efektif karena pencarian kata kunci seperti ”Python” atau ”React” dilakukan pada gabungan teks dari seluruh CV. Kompleksitas waktu pra-pemrosesan KMP adalah $O(m)$ dan kompleksitas waktu pencariannya adalah $O(n)$, dengan m adalah panjang *pattern* dan n adalah panjang teks.

4.4.2 Boyer-Moore (BM)

Algoritma `Boyer-Moore` adalah salah satu algoritma pencocokan string tercepat dalam praktiknya untuk pencarian satu *pattern*. Berbeda dengan KMP, BM melakukan pencocokan dari karakter terakhir *pattern* ke karakter pertama. Keunggulannya terletak pada dua aturan heuristik yang digunakan saat terjadi ketidakcocokan: *Bad Character Heuristic* dan *Good Suffix Heuristic*. Heuristik ini memungkinkan BM untuk melakukan lompatan yang sangat jauh pada teks, sering kali melebihi panjang *pattern* itu sendiri. Pada sistem ATS ini, di mana teks CV bisa sangat panjang, BM sering kali lebih unggul dari KMP untuk pencarian satu kata kunci. Kompleksitas waktu terbaiknya bisa mencapai $O(n/m)$, meskipun pada kasus terburuk (worst-case) kompleksitasnya adalah $O(nm)$. Namun, kasus terburuk ini sangat jarang terjadi pada teks bahasa alami seperti dalam CV.

4.4.3 Aho-Corasick

Algoritma `Aho-Corasick` adalah ekstensi dari KMP yang dirancang untuk mencari semua kemunculan dari sekumpulan *keywords* secara bersamaan dalam satu kali lintasan teks. Algoritma ini pertama-tama membangun struktur data *finite state machine* yang berbentuk seperti pohon *Trie* dari semua kata kunci. Kemudian, ditambahkan "failure links" yang mirip dengan tabel LPS pada KMP, yang berfungsi untuk menavigasi automata saat terjadi ketidakcocokan. Dalam sistem ATS, jika pengguna memasukkan banyak kata kunci (misal: "Java, Spring, Microservices, Kafka"), Aho-Corasick secara umum mengungguli algoritma lain. Alih-alih menjalankan pencarian berulang kali, algoritma ini memproses teks CV satu kali saja untuk menemukan semua kata kunci. Kompleksitas waktunya adalah $O(\sum m_i)$ untuk membangun automata dan $O(n+k)$ untuk pencarian, di mana n adalah panjang teks, $\sum m_i$ adalah total panjang semua *keywords*, dan k adalah jumlah total kecocokan yang ditemukan.

4.4.4 Levenshtein Distance

`Levenshtein Distance` bukanlah algoritma pencarian, melainkan sebuah metrik untuk mengukur "jarak" atau perbedaan antara dua string. Algoritma ini menghitung jumlah minimum operasi *single-character edit* (insersi, delesi, atau substitusi) yang diperlukan untuk mengubah satu string menjadi string yang lain. Dalam implementasi

program ini, `Levenshtein Distance` digunakan untuk *fuzzy matching*. Ketika algoritma pencocokan eksak tidak menemukan hasil, sistem akan menggunakan `Levenshtein Distance` untuk mencari kata dalam CV yang "paling mirip" dengan kata kunci yang dicari. Jika jaraknya berada di bawah ambang batas (*threshold*) yang ditentukan, kata tersebut dianggap sebagai kecocokan yang relevan. Ini sangat berguna untuk mengatasi kesalahan pengetikan oleh pengguna. Kompleksitas waktunya adalah $O(m \times n)$, di mana m dan n adalah panjang kedua string yang dibandingkan.

4.4.5 Regular Expression (Regex)

`Regular Expression (Regex)` adalah sebuah sekuens karakter yang mendefinisikan sebuah pola pencarian. Regex tidak digunakan untuk mencari apakah sebuah kata kunci ada atau tidak, melainkan untuk mengekstrak informasi yang memiliki pola terstruktur dari dalam teks. Dalam sistem ATS ini, Regex adalah tulang punggung dari fitur ekstraksi ringkasan CV (*CV Summary*). Setelah sebuah CV yang relevan ditemukan, sistem menggunakan pola-pola Regex yang telah didefinisikan untuk menemukan dan mengekstrak informasi spesifik seperti: nama, alamat email, nomor telepon, riwayat pendidikan, dan pengalaman kerja. Regex sangat kuat untuk parsing data semi-terstruktur seperti yang ditemukan dalam berbagai format CV.

Seluruh analisis di atas dapat dirangkum menjadi tabel berikut:

Tabel 1: Perbandingan Algoritma String-Matching

Algoritma	**Preprocessing Time**	**Searching Time**	**Kasus Penggunaan Terbaik**
KMP	$O(m)$	$O(n)$	Teks dengan alfabet kecil atau *pattern* yang memiliki banyak pola berulang.
Boyer-Moore	$O(m + \|\Sigma\|)$	$O(n/m)$ (best)	Teks panjang dengan alfabet besar (seperti bahasa alami) dan *pattern* yang relatif panjang.
Aho-Corasick	$O(\sum m_i)$	$O(n + k)$	Mencari banyak *pattern* $(m_1, m_2, ...)$ sekaligus dalam sekali proses.

Bab V
Penutup

5.1. Kesimpulan

Pada Tugas Besar 3 IF2211 Strategi Algoritma ini, algoritma *string matching* berupa *Knuth-Morris-Pratt*, *Boyer-Moore*, dan *Aho-Corasick* dapat digunakan dalam melakukan *exact matching* pada hasil ekstraksi CV kandidat. Selain itu, jika tidak ditemukan kesamaan, algoritma *Levenshtein Distance* dapat melakukan *fuzzy matching* sebagai solusi agar sistem tetap bisa menemukan CV yang relevan walaupun terdapat perbedaan kecil dalam penulisan *query*.

Kami berhasil untuk mengimplementasikan algoritma *string matching* dalam aplikasi Sistem ATS. Selain menyediakan algoritma *pathfinding* yang wajib diimplementasikan, program kami juga mengimplementasikan algoritma *string matching* tambahan.

Selain itu, pengguna dapat melihat modal *summary* dari CV para kandidat yang berisi informasi-informasi dari kandidat tersebut, *summary* ini dibuat menggunakan *Regular Expression* untuk ekstraksi informasi penting dan terstruktur.

5.2. Saran

Pelaksanaan Tugas Besar 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:

fariz

Tolong asisten ngasih tubes nya jangan pas uas plis :(

adhahutao

stop tubes di uas pls, or uas di tubes, either way.

edo

keren

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

5.3. Tautan

Repository program dapat diakses melalui tautan berikut:

`https://github.com/Fariz36/Tubes3_10Internship0Job`

5.4. Lampiran

Tabel 2: 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 *Knuth-Morris-Pratt (KMP)* dan *Boyer-Moore (BM)* dapat menemukan kata kunci dengan benar	✓	
5	Aplikasi *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	[**Bonus**] Membuat enkripsi data profil *applicant*		✓
10	[**Bonus**] Membuat algoritma *Aho-Corasick*	✓	
11	[**Bonus**] Membuat video dan diunggah pada Youtube		✓

Referensi

Rinaldi Munir. (2025). "Bahan Kuliah IF2211 Strategi Algoritma: Pencocokan String" Diakses pada 15 Juni 2025. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf

Rinaldi Munir. (2025). "Bahan Kuliah IF2211 Strategi Algoritma: String Matching dengan Regex" Diakses pada 15 Juni 2025. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/24-String-Matching-dengan-Regex-(2025).pdf

cp-algorithms.com. (2025). "Prefix function. Knuth-Morris-Pratt algorithm" Diakses pada 15 Juni 2025. https://cp-algorithms.com/string/prefix-function.html

cp-algorithms.com. (2025). "Boyer-Moore algorithm" Diakses pada 15 Juni 2025. https://cp-algorithms.com/string/boyer-moore.html

GeeksforGeeks. (2025). "Aho-Corasick Algorithm for Pattern Searching" Diakses pada 15 Juni 2025. https://www.geeksforgeeks.org/dsa/aho-corasick-algorithm-pattern-searching/