

Laporan Tugas Besar 3

IF2211 Strategi Algoritma



Disusun oleh:
CV Magang

| | |
|-------------------------------|----------|
| Adiel Rum | 10123004 |
| Zulfaqqar Nayaka Athadiansyah | 13523094 |
| Farrel Athalla Putra | 13523118 |

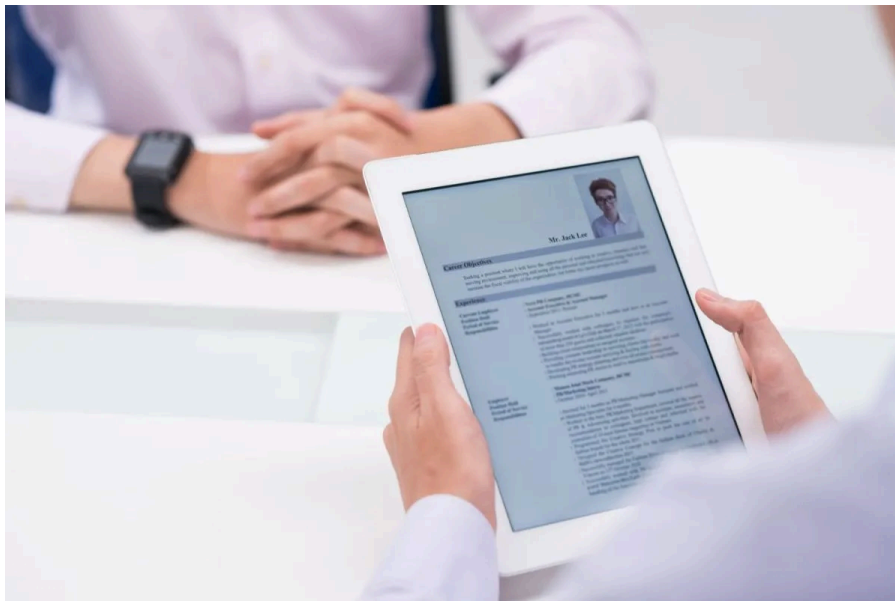
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Bandung
2025

Daftar Isi

| | |
|--|-----------|
| Daftar Isi..... | 1 |
| 1. Deskripsi Tugas..... | 3 |
| 2. Landasan Teori..... | 4 |
| 2.1. Algoritma Knuth–Morris–Pratt (KMP)..... | 4 |
| 2.2. Algoritma Boyer–Moore (BM)..... | 5 |
| 2.3. Algoritma Aho–Corasick (AC)..... | 7 |
| 2.4. Aplikasi yang Dibangun..... | 9 |
| 3. Analisis Pemecahan Masalah..... | 10 |
| 3.1. Langkah-langkah Pemecahan Masalah..... | 10 |
| 3.1.1. Ekstraksi Teks dari PDF..... | 10 |
| 3.1.2. Pencarian Kata Kunci..... | 10 |
| 3.1.3. Menampilkan Hasil..... | 10 |
| 3.2. Pemetaan Masalah..... | 10 |
| 3.3. Fitur..... | 10 |
| 3.3.1. Pencarian Kemunculan Kata Kunci pada Basis Data CV..... | 11 |
| 3.3.2. Pemilihan Algoritma Pencarian..... | 11 |
| 3.3.3. Menampilkan Ringkasan Data dan CV Applicant..... | 11 |
| 3.3.4. Enkripsi Data Applicant..... | 11 |
| 3.4. Arsitektur Aplikasi..... | 11 |
| 3.5. Skenario..... | 14 |
| 4. Implementasi dan Pengujian..... | 15 |
| 4.1. Spesifikasi Teknis..... | 15 |
| 4.1.1. Struktur Data..... | 15 |
| 4.1.2. Subprogram..... | 16 |
| 4.2. Tata Cara Penggunaan Aplikasi..... | 28 |
| 4.3. Pengujian..... | 29 |
| 4.3.1. Kasus Uji 1: KMP (1)..... | 29 |
| 4.3.2. Kasus Uji 1: KMP (2)..... | 29 |
| 4.3.3. Kasus Uji 2: BM (1)..... | 30 |
| 4.3.4. Kasus Uji 2: BM (2)..... | 30 |
| 4.3.5. Kasus Uji 3: AC (1)..... | 31 |
| 4.3.6. Kasus Uji 3: AC (2)..... | 31 |
| 4.3.7. Kasus Uji 4: Levenshtein Distance..... | 31 |
| 4.4. Analisis..... | 32 |
| 5. Penutup..... | 33 |
| 5.1. Kesimpulan..... | 33 |
| 5.2. Saran..... | 33 |

| | |
|----------------------------|-----------|
| 5.3. Refleksi..... | 33 |
| Lampiran..... | 34 |
| Daftar Pustaka..... | 34 |

1. Deskripsi Tugas



Gambar 1. CV ATS dalam Dunia Kerja
(Sumber: <https://www.antaraneews.com/>)

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.

Di dalam Tugas Besar 3 ini, Anda 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.

2. Landasan Teori

2.1. Algoritma Knuth–Morris–Pratt (KMP)

Algoritma Knuth-Morris-Pratt (KMP) adalah sebuah algoritma pencocokan string yang sangat efisien karena mampu menghindari perbandingan karakter yang berulang. Berbeda dengan pendekatan naif yang menggeser pola satu per satu saat terjadi ketidakcocokan, KMP memanfaatkan informasi dari pola itu sendiri untuk melakukan pergeseran yang cerdas. Kunci dari algoritma ini adalah tahap pra-pemrosesan pada pola untuk membangun sebuah larik bantu yang disebut *Longest Proper Prefix which is also Suffix* (LPS). Larik LPS ini menyimpan panjang prefiks terpanjang dari sebuah sub-pola yang juga merupakan sufiks dari sub-pola tersebut. Ketika ketidakcocokan terjadi antara teks dan pola, algoritma akan menggunakan nilai dari larik LPS untuk mengetahui seberapa jauh pola dapat digeser ke kanan tanpa melewatkan kemungkinan adanya kecocokan. Dengan demikian, KMP tidak perlu mengulang perbandingan pada karakter teks yang sebelumnya sudah cocok. Berikut adalah kode semu (*pseudocode*) dari algoritma pembangunan LPS dan algoritma KMP.

prosedur computeLPS(pattern)

KAMUS

m : panjang dari pattern

lps : larik integer berukuran m

length : panjang dari prefiks-sufiks terpanjang sebelumnya

i : iterator

ALGORITMA

while → **lps**ALGORITMA

 length ← 0

 lps ← 0

 i ← 1

 while (i < m) do

 if (pattern[i] == pattern[length]) then

 length ← length + 1

 lps[i] ← length

 i ← i + 1

 else

 if (length != 0) then

 length ← **lps**[length - 1]

 else

 lps[i] ← 0

 i ← i + 1

 endif

```

    endif
endwhile
→ lps

```

prosedur KMPSearch(text, pattern)

KAMUS

n : panjang dari text
 m : panjang dari pattern
 lps : larik LPS yang dihasilkan oleh computeLPS(pattern)
 i : indeks untuk text
 j : indeks untuk pattern
 positions : senarai untuk menyimpan posisi ditemukan

ALGORITMA

```

while (i < n) do
  if (pattern[j] == text[i]) then
    i ← i + 1
    j ← j + 1
  endif
  if (j == m) then
    tambahkan (i - j) ke positions
    j ← lps[j - 1]
  else if (i < n AND pattern[j] != text[i]) then
    if (j != 0) then
      j ← lps[j - 1]
    else
      i ← i + 1
    endif
  endif
endif
endwhile

```

Tahap pra-pemrosesan KMP untuk membangun larik LPS membutuhkan waktu $O(m)$, dengan m adalah panjang pola. Tahap pencariannya sendiri membutuhkan waktu $O(n)$, dengan n adalah panjang teks. Dengan demikian, kompleksitas waktu total dari algoritma KMP adalah $O(n+m)$.

2.2. Algoritma Boyer–Moore (BM)

Algoritma Boyer-Moore adalah salah satu algoritma pencocokan string yang paling efisien dalam praktiknya. Keunggulan utamanya terletak pada pendekatannya yang unik, yaitu melakukan perbandingan karakter dari **kanan ke kiri**, bukan dari kiri ke kanan seperti algoritma pada umumnya. Pendekatan ini memungkinkan algoritma untuk melakukan pergeseran (*shift*) pola dalam jarak yang besar setiap kali terjadi ketidakcocokan, sehingga secara signifikan mengurangi jumlah total perbandingan yang diperlukan. Untuk menentukan sejauh mana pola akan digeser, Boyer-Moore menggunakan dua strategi atau heuristik:

Heuristik Karakter Buruk (*Bad Character Heuristic*): Ketika terjadi ketidakcocokan antara karakter pada teks, katakanlah c , dengan karakter pada pola, heuristik ini akan menggeser pola ke kanan hingga karakter c pada teks sejajar dengan kemunculan karakter c yang paling kanan pada pola. Jika karakter c tidak ada dalam pola, maka pola dapat digeser sepenuhnya melewati posisi c pada teks.

Heuristik Karakter Buruk (*Bad Character Heuristic*): Ketika terjadi ketidakcocokan antara karakter pada teks, katakanlah c , dengan karakter pada pola, heuristik ini akan menggeser pola ke kanan hingga karakter c pada teks sejajar dengan kemunculan karakter c yang paling kanan pada pola. Jika karakter c tidak ada dalam pola, maka pola dapat digeser sepenuhnya melewati posisi c pada teks.

Algoritma akan memilih pergeseran terjauh dari kedua heuristik tersebut untuk memaksimalkan efisiensi. Berikut adalah kode semu (*pseudocode*) dari algoritma Boyer-Moore

prosedur preprocess(pattern)

KAMUS

m : panjang dari pattern

bad_char : larik untuk heuristik karakter buruk (ukuran sesuai alfabet)

$good_suffix$: larik untuk heuristik sufiks baik

ALGORITMA

for i from 0 to $m-1$ do

$bad_char[pattern[i]] \leftarrow i$

endfor

→ $bad_char, good_suffix$

prosedur BMSearch(text, pattern)

KAMUS

n : panjang dari text

m : panjang dari pattern

$bad_char, good_suffix$: larik hasil dari preprocess(pattern)

s : indeks pergeseran pola pada teks

j : indeks untuk pattern (dari kanan ke kiri)

$positions$: senarai untuk menyimpan posisi ditemukan

ALGORITMA

$(bad_char, good_suffix) \leftarrow preprocess(pattern)$

$s \leftarrow 0$

while $(s \leq n - m)$ do

$j \leftarrow m - 1$

 while $(j \geq 0 \text{ AND } pattern[j] == text[s + j])$ do

$j \leftarrow j - 1$

```

endwhile
if (j < 0) then
    tambahkan s ke positions // Geser pola berdasarkan aturan sufiks baik
    untuk pola yang cocok penuh
    s ← s + good_suffix
else
    // Mismatch terjadi pada text[s + j]
    shift_bad_char ← j - bad_char[text[s + j]]
    shift_good_suffix ← good_suffix[j]
    // Geser pola sejauh nilai maksimum dari kedua heuristik
    s ← s + max(1, shift_bad_char, shift_good_suffix)
endif
endwhile
→ positions

```

Kompleksitas waktu untuk tahap pra-pemrosesan Boyer-Moore adalah $O(m+|\Sigma|)$, di mana m adalah panjang pola dan $|\Sigma|$ adalah ukuran alfabet. Pada kasus terburuk, kompleksitas pencariannya bisa mencapai $O(nm)$, namun dalam praktiknya untuk teks pada umumnya, algoritma ini seringkali memiliki performa sub-linear, menjadikannya lebih cepat dari KMP.

2.3. Algoritma Aho–Corasick (AC)

Selain Algoritma KMP dan Algoritma BM, Algoritma Aho-Corasick adalah sebuah metode pencocokan string yang dirancang untuk menemukan semua kemunculan dari sekumpulan pola (kamus) secara serentak di dalam sebuah teks. Efisiensinya yang tinggi didapat dengan memproses teks hanya dalam satu kali lintasan (*single pass*). Algoritma ini bekerja dengan membangun sebuah *finite automaton* atau mesin keadaan terbatas dari semua pola yang dicari. Struktur ini pada dasarnya menggabungkan pohon prefiks (*trie*) dengan konsep fungsi kegagalan dari algoritma KMP.

Proses pembentukan automaton terdiri dari dua tahap utama, pertama adalah tahap Pembangunan Trie. Semua pola dalam kamus dimasukkan ke dalam sebuah struktur data *trie*. Setiap simpul pada *trie* merepresentasikan sebuah prefiks, dan simpul yang menandai akhir dari sebuah pola akan ditandai sebagai simpul keluaran (*output node*).

Tahap kedua adalah tahap Pembangunan Tautan Kegagalan (*Failure Links*): Setelah *trie* terbentuk, setiap simpul di dalamnya akan diberi sebuah tautan kegagalan. Tautan ini menunjuk ke simpul lain yang merepresentasikan sufiks sejati (*proper suffix*) terpanjang dari string pada simpul saat ini, yang juga merupakan prefiks dari salah satu pola lain di dalam kamus. Tautan ini memungkinkan algoritma untuk melanjutkan pencocokan dari keadaan alternatif tanpa harus mundur (*backtracking*) pada teks saat terjadi ketidakcocokan.

Ketika pencarian dilakukan, algoritma akan membaca teks karakter per karakter sambil bergerak melalui keadaan-keadaan dalam automaton. Setiap kali mencapai simpul keluaran,

sebuah kecocokan akan dicatat. Berikut adalah kode semu (*pseudocode*) dari algoritma Aho-Corasick.

prosedur buildAutomaton(patterns)

KAMUS

root : simpul akar dari trie

queue : antrian untuk proses BFS dalam membangun tautan kegagalan

ALGORITMA

```

    root ← node_baru()
    for each pattern in patterns do
        node ← root
        for each char in pattern do
            if (node tidak punya anak untuk char) then
                node.anak[char] ← node_baru()
            endif
            node ← node.anak[char]
        endfor
        tambahkan pattern ke node.output
    endfor

    inisialisasi queue
    for each anak in root.anak do
        anak.failure ← root
        tambahkan anak ke queue
    endfor

    while (queue tidak kosong) do
        node_sekarang ← ambil_dari_queue()
        for each (char, anak) in node_sekarang.anak do
            tautan_kegagalan ← node_sekarang.failure
            while (tautan_kegagalan != null AND tautan_kegagalan tidak punya
anak untuk char) do
                tautan_kegagalan ← tautan_kegagalan.failure
            endwhile
            if (tautan_kegagalan != null) then
                anak.failure ← tautan_kegagalan.anak[char]
            else
                anak.failure ← root
            endif
            gabungkan anak.output dengan anak.failure.output
            tambahkan anak ke queue
        endfor
    endfor
    → root

```

prosedur ACSearch(text, patterns)**KAMUS**

root : automaton yang dihasilkan oleh buildAutomaton(patterns)

node_sekarang : simpul saat ini pada automaton

results : senarai untuk menyimpan semua kecocokan yang ditemukan

ALGORITMA

```

root ← buildAutomaton(patterns)
node_sekarang ← root
for i from 0 to panjang(text)-1 do
  char ← text\[i\]
  while (node_sekarang != null AND node_sekarang tidak punya anak
untuk char) do
    node_sekarang ← node_sekarang.failure
  endwhile
  if (node_sekarang == null) then
    node_sekarang ← root
  else
    node_sekarang ← node_sekarang.anak\[char\]
  endif
  if (node_sekarang.output tidak kosong) then
    for each pattern_ditemukan in node_sekarang.output do
      posisi ← i - panjang(pattern_ditemukan) + 1
      tambahkan (pattern_ditemukan, posisi) ke results
    endfor
  endif
endfor
→ results

```

Kompleksitas waktu dari algoritma Aho-Corasick sangat efisien. Jika m adalah total panjang semua pola dalam kamus, n adalah panjang teks, dan k adalah jumlah kecocokan yang ditemukan, maka kompleksitas waktu totalnya adalah $O(n+m+k)$.

2.4. Aplikasi yang Dibangun

Aplikasi berbasis GUI *desktop* yang diimplementasikan untuk tugas besar ini menggunakan *framework* pengembangan GUI PyQt dan bahasa pemrograman Python. Untuk mempercepat pencarian, program menerapkan *multithreading*. Penjelasan mendalam dari aplikasi ini beserta fitur-fiturnya akan diberikan pada subbab [Fitur](#) dan [Arsitektur Aplikasi](#).

<skrinsut>

3. Analisis Pemecahan Masalah

3.1. Langkah-langkah Pemecahan Masalah

Secara umum, pemecahan masalah yang akan dilakukan pada aplikasi kami dapat dideskripsikan oleh diagram berikut.

3.1.1. Ekstraksi Teks dari PDF

Tahap pertama adalah ekstraksi dan pemrosesan data. Pertama-tama, setiap file CV dari direktori data/data dibaca, mengekstrak teks mentah dari dalamnya, dan membersihkan residu-residu umum yang muncul dari konversi PDF. Teks mentah ini kemudian diidentifikasi dan diekstrak informasi terstruktur seperti keahlian, riwayat pekerjaan, dan pendidikan menggunakan serangkaian pola *regular expression*.

3.1.2. Pencarian Kata Kunci

Ketika pengguna melakukan kueri, proses pencarian dimulai. Aplikasi pertama-tama melakukan pencarian persis (exact match) menggunakan salah satu dari tiga algoritma yang dipilih (KMP, Boyer-Moore, atau Aho-Corasick). Algoritma ini akan memindai teks dari setiap CV untuk menemukan semua kemunculan kata kunci yang dimasukkan. Selanjutnya, untuk kata kunci yang tidak ditemukan pada tahap pertama, aplikasi melakukan pencarian *fuzzy* menggunakan algoritma jarak Levenshtein. Algoritma ini mencari kata-kata yang mirip secara ejaan dengan menghitung "jarak edit".

3.1.3. Menampilkan Hasil

Hasil dari kedua jenis pencarian tersebut digabungkan dan ditampilkan kepada pengguna. Hasil pencarian diurutkan berdasarkan jumlah total kecocokan kata kunci, dari yang paling banyak hingga paling sedikit. Pengguna dapat melihat detail lebih lanjut seperti ringkasan komprehensif dengan menggabungkan data pribadi yang sudah didekripsi dari basis data (seperti nama dan kontak) dengan detail keahlian, pengalaman, dan pendidikan yang diekstrak dari teks CV.

3.2. Pemetaan Masalah

Tiap kata kunci yang dimasukkan oleh pengguna akan berperan sebagai pola (*pattern*) pada algoritma pencocokan string (KMP, BM, AC), sementara teks hasil ekstraksi dari PDF berperan sebagai teks yang akan ditelusuri untuk menemukan kemunculan pola. Algoritma KMP, BM, dan AC menerima masukan tersebut, menerapkan pencarian berdasarkan algoritma masing-masing, lalu mengembalikan letak kemunculan dari pola yang diberikan.

3.3. Fitur

3.3.1. Pencarian Kemunculan Kata Kunci pada Basis Data CV

Pengguna dapat mencari seluruh letak kemunculan suatu kata kunci yang ia berikan pada program sebagai masukan pada basis data CV. Pengguna dapat menspesifikkan berapa banyak kecocokan yang ia mau untuk ditampilkan. Aplikasi kemudian akan menyajikan waktu pencarian beserta *card* yang merepresentasikan data *applicant* dengan konten CV yang memuat kata kunci tersebut. Pada tiap *card*, aplikasi merincikan banyaknya kecocokan kata kunci pada CV.

3.3.2. Pemilihan Algoritma Pencarian

Pengguna dapat memilih algoritma yang ingin digunakan, baik KMP, BM, maupun AC.

3.3.3. Menampilkan Ringkasan Data dan CV *Applicant*

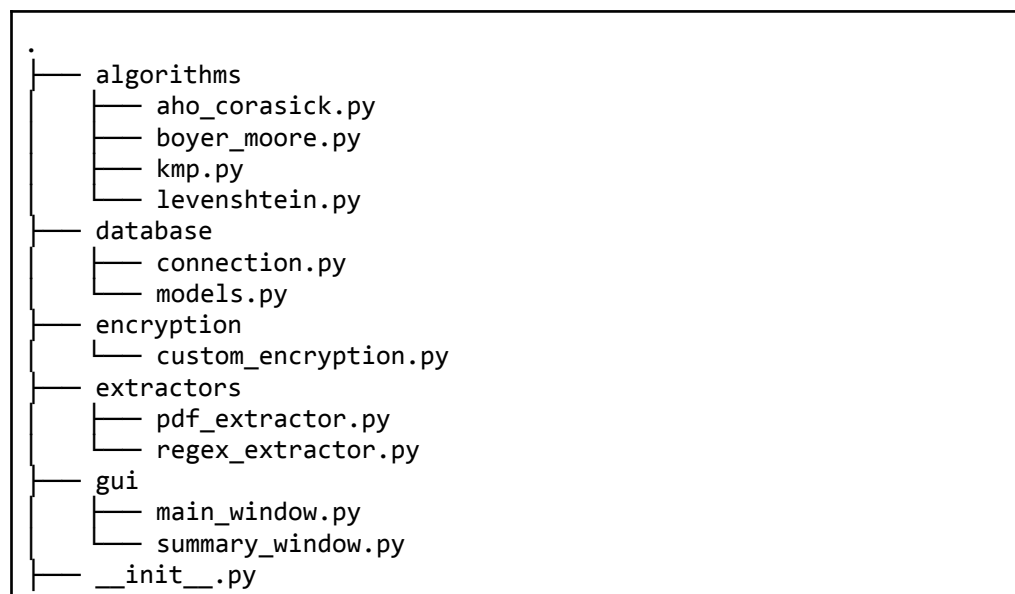
Dari *card* milik *applicant* yang muncul sebagai hasil pencarian, pengguna dapat mengakses jendela *popup* yang berisi ringkasan dari CV *applicant* yang bersangkutan. Pengguna juga dapat menampilkan dokumen CV *applicant*.

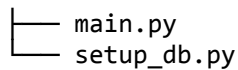
3.3.4. Enkripsi Data *Applicant*

Data *applicant* dalam basis data dienkripsi dengan kombinasi dari *Caesar cipher* dan XOR.

3.4. Arsitektur Aplikasi

Aplikasi ditulis dalam bahasa pemrograman Python dengan kaskas GUI PyQt. Berikut adalah struktur folder dari aplikasi yang dibuat.



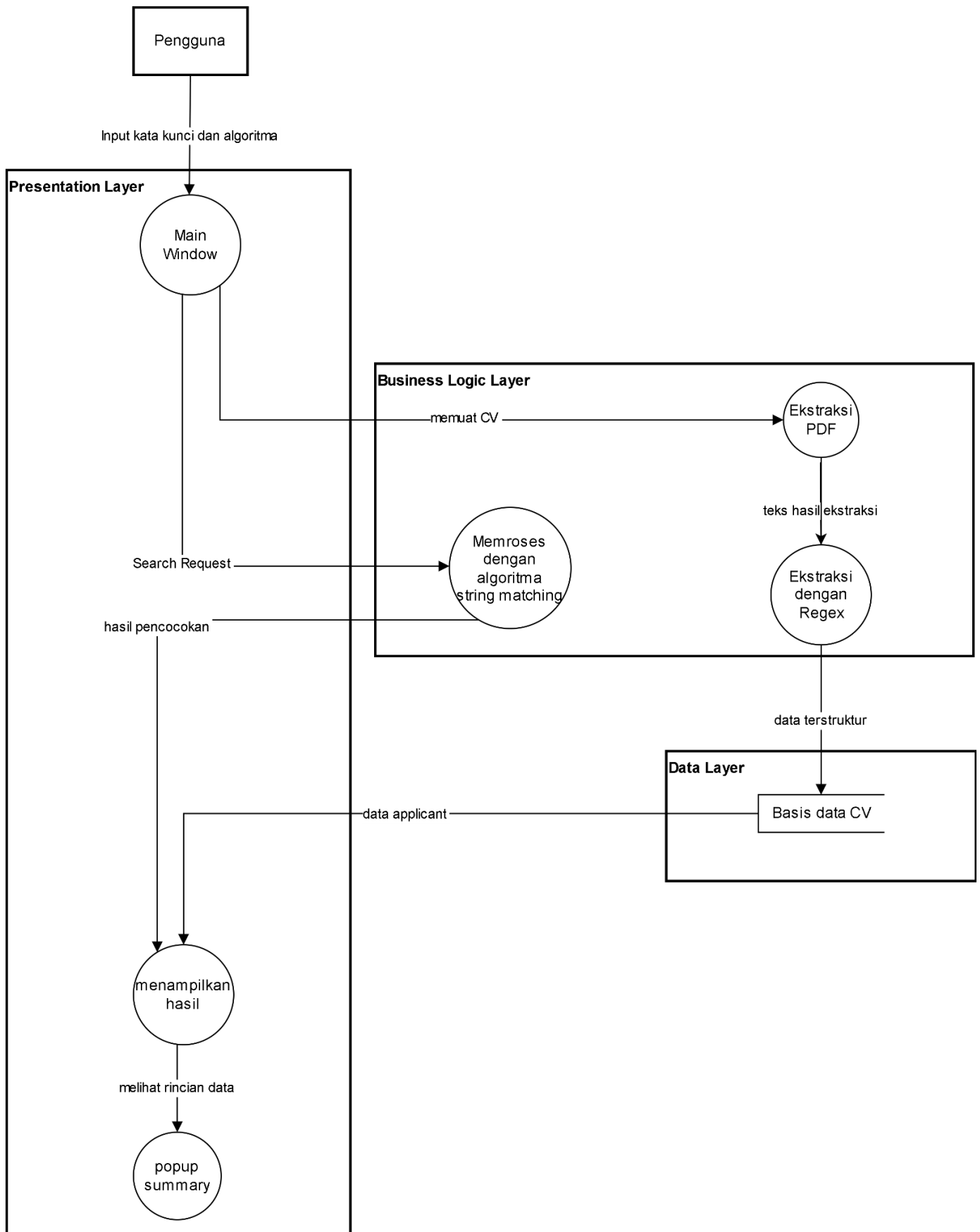


```
└─ main.py
└─ setup_db.py
```

Secara garis besar, aplikasi ini dirancang sesuai dengan *design pattern* Model-View-Controller (MVC) dengan empat *layer/lapis*:

- *presentation layer*: kode terkait GUI PyQT di folder `src/gui`
 - `main_window.py` mengimplementasikan *window* utama aplikasi dan mengintegrasikan algoritma pencarian, enkripsi data, dan koneksi ke basis data MySQL.
 - `summary_window.py` mengimplementasikan jendela *popup* untuk menyajikan rincian data CV.
- *business logic layer*: kode terkait algoritma pencocokan *string* di folder `src/algorithms` serta pemrosesan ekstraksi dan *parsing* teks dari PDF di folder `src/extractors`.
- *data access layer*: kode terkait koneksi aplikasi ke basis data MySQL beserta enkripsi data di folder `src/database` dan `src/encryption`.
- *database layer*: basis data MySQL yang menyimpan data *applicant* dan *path* ke CV.

Alur data dari aplikasi dirincikan oleh diagram alur data (*data flow diagram*) berikut.



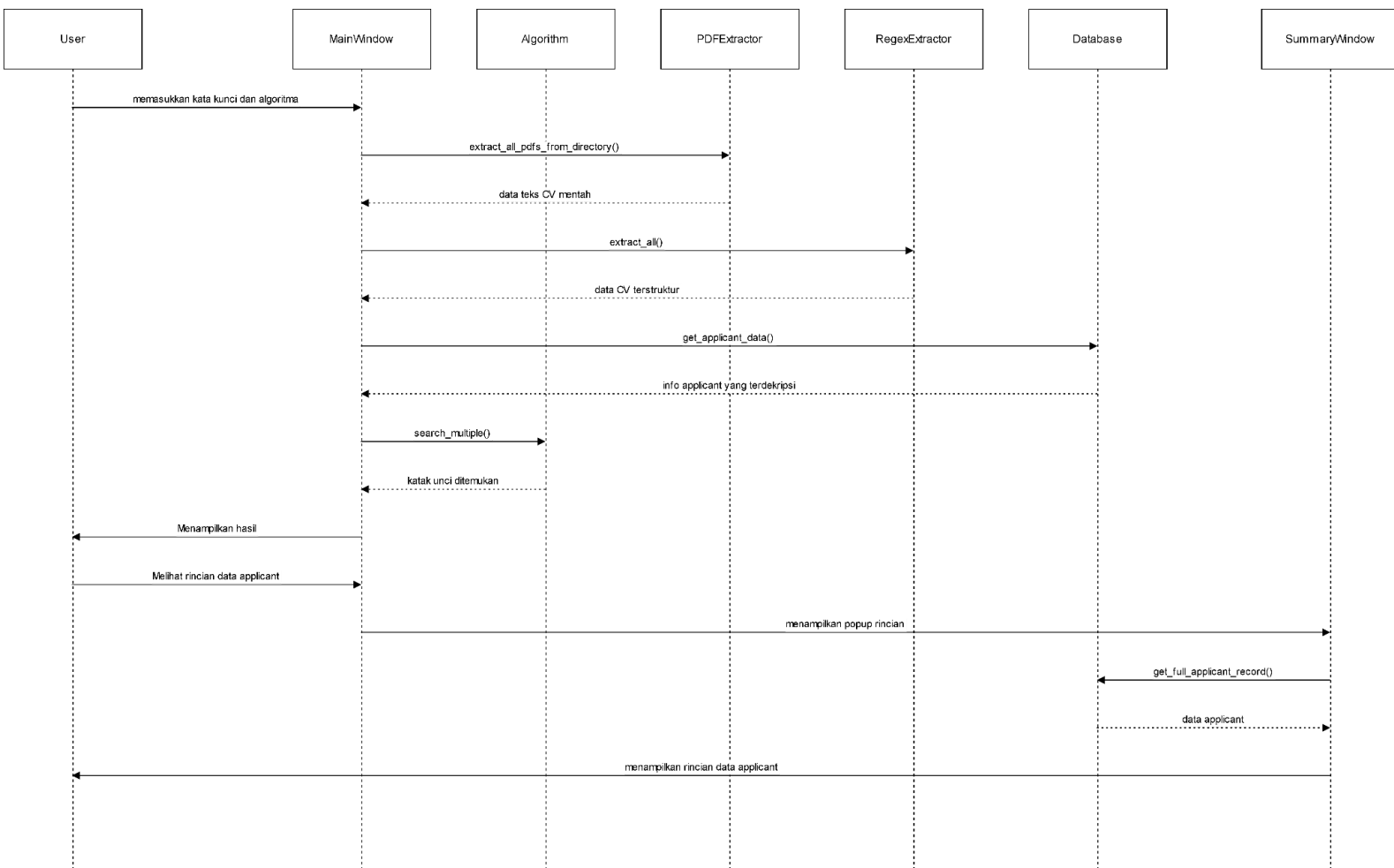
3.5. Skenario

Mula-mula, ketika aplikasi dijalankan, ia akan mengekstraksi teks dengan `self.pdf_extractor.extract_all_pdfs_from_directory(self.data_path)` `cv['extracted_info'] = self.regex_extractor.extract_all(cv['text'])` yang menghasilkan data CV terstruktur dan disimpan secara *runtime* di aplikasi utama, sementara *path* ke CV disimpan ke basis data beserta profil yang telah di-*seeding*.

Misalkan Farrel memasukkan kata kunci “finance” dan memilih algoritma Aho-Corasick dan ingin menampilkan 5 kemunculan teratas. `search_cvs` akan dipanggil dan memilih algoritma BM. Di sini akan dilakukan *looping* terhadap tiap data CV yang ada di `self.cv_data` di mana metode `search_multiple` (ini kami bikin ada di semua algoritma) akan dipanggil dengan teks `cv['text']` dan *pattern* “finance”.

Setiap CV yang mengandung kata “finance” akan disimpan ke *list* `exact_results`. Setiap elemen dalam *list* ini adalah sebuah *dictionary* yang berisi data CV, detail kecocokan, dan jumlah total kecocokan (`total_count`). Hasil pencarian kemudian diurutkan menurun berdasarkan `match_count`. Aplikasi kemudian akan menampilkan hasil sebagai *card* di laman utama.

Secara umum, alur kerja aplikasi dideskripsikan dengan diagram sekuens berikut.



4. Implementasi dan Pengujian

4.1. Spesifikasi Teknis

4.1.1. Struktur Data

Data yang disimpan ke basis data MySQL dalam tabel ApplicantProfile direpresentasikan dalam program dengan model berikut.

```

{
  'first_name': first_name,
  'last_name': last_name,
  'address': address,
  'phone_number': phone_number
}
  
```

Data yang di-*parsing* oleh kelas RegexExtractor menggunakan metode extract_all disimpan secara *runtime* dengan format sebagai berikut.


```
{
    'personal_info': {}, 'summary': '', 'skills': [],
    'experience': [], 'education': []
}
```

Seluruh data CV secara terpusat disimpan dalam *dictionary* di MainWindow sebagai atribut `cv_data`. Formatnya (meskipun tidak didefinisikan secara eksplisit di program kami karena atributnya ditambah secara terpisah) adalah sebagai berikut.

```
{
    'path': str
    'text': str # hasil PDFExtractor
    'category': str # kategori pekerjaan
    'extracted_info': dict # Hasil dari RegexExtractor
    'name': str
    'applicant_id': int
    'db_first_name': str
    'db_last_name': str
    'db_phone': str
    'db_address': str
    'db_dob': str
}
```

Untuk demonstrasi, kami menggunakan *seeder* yang disediakan oleh asisten. Akan tetapi, ada beberapa *application* yang sayangnya tidak dipasangkan dengan suatu *applicant*. Akibatnya, kami membuat *seeding* sendiri di `utils/seed.py`.

Terakhir, hasil pencarian di-*passing* sebagai struktur data berikut ke CVCard untuk ditampilkan ke GUI.

```
{
    'path': result['cv']['path'],
    'name': result['cv']['name'],
    'match_count': result['total_count'],
    'keywords_found': {k: v['count'] for k, v in result['matches'].items()},
    'extracted_info': result['cv']['extracted_info'],
    'text': result['cv']['text'],
    'applicant_id': result['cv'].get('applicant_id'),
    'db_first_name': result['cv'].get('db_first_name', ''),
    'db_last_name': result['cv'].get('db_last_name', ''),
    'db_phone': result['cv'].get('db_phone', ''),
    'db_address': result['cv'].get('db_address', ''),
    'db_dob': result['cv'].get('db_dob', '')
}
```

4.1.2. Subprogram

Berikut adalah *code snippet* dari sejumlah subprogram inti dalam aplikasi, khususnya dari *backend* yang menampung logika utama aplikasi.

src\algorithms\kmp.py

Kelas implementasi dari algoritma Knuth-Morris-Patt.

```
#####
#####
## @file kmp.py
## Ini isinya implementasi algoritma Knuth-Morris-Pratt (KMP)
#####
#####

from typing import List, Dict

class KMP:
    def __init__(self):
        self.name = "KMP"
        self._lps_cache: Dict[str, List[int]] = {}

    # @brief Menghitung Longest Prefix Suffix array dari pola
    # @param pattern: Pola yang dipakai untuk pattern matching
    # @return: LPS array yang berisi panjang prefix yang juga suffix untuk setiap indeks
    def _compute_lps(self, pattern: str) → List[int]:
        if pattern in self._lps_cache:
            return self._lps_cache[pattern]

        m: int = len(pattern)
        lps: List[int] = [0] * m
        length: int = 0
        i: int = 1

        while i < m:
            if pattern[i] == pattern[length]:
                length += 1
                lps[i] = length
                i += 1
            else:
                if length != 0:
                    length = lps[length - 1]
                else:
                    lps[i] = 0
                    i += 1
        self._lps_cache[pattern] = lps
        return lps

    # @brief fungsi utama untuk algoritma KMP
    # @param text: Teks yang akan dicari
    # @param pattern: Pola yang akan dicocokkan
    # @return: List posisi di mana pola ditemukan dalam teks, kalau gaada ya kosong
    def search(self, text: str, pattern: str, is_lowercase: bool = False) → List[int]:
        if not text or not pattern:
            return []

        n: int = len(text)
        m: int = len(pattern)

        if m > n:
            return []

        # prep
        text_lower: str = text.lower() if is_lowercase else text
        pattern_lower: str = pattern.lower()
        positions: list[int] = []
```

```

# ambil dari cache atau hitung baru
lps: List[int] = self._compute_lps(pattern_lower)

i: int = 0 # index untuk teks
j: int = 0 # index untuk pola

while i < n:
    if text_lower[i] == pattern_lower[j]:
        i += 1
        j += 1

    if j == m:
        positions.append(i - j)
        j = lps[j - 1]
    elif i < n and text_lower[i] != pattern_lower[j]:
        if j != 0:
            j = lps[j - 1]
        else:
            i += 1
    return positions

# @brief Mencari beberapa pola dalam teks
# @param text: Teks yang akan dicari
# @param patterns: List pola yang akan dicocokkan
# @return: Dictionary yang bentuknya kayak <pola, <posisi, count>>
def search_multiple(self, text: str, patterns: list[str]) → dict[str, dict[str, int]]:
    results: dict[str, dict[str, int]] = {}
    text = text.lower()
    for pattern in patterns:
        positions = self.search(text, pattern, is_lowercase=True)
        if positions:
            results[pattern] = {
                'positions': positions,
                'count': len(positions)
            }
    return results

# @brief membersihkan cache LPS
# @return: None
def clear_cache(self):
    self._lps_cache.clear()

```

src\algorithms\boyer_moore.py

Kelas implementasi dari algoritma Boyer-Moore.

```

#####
#####
## @file boyer_moore.py
## Ini isinya implementasi algoritma Boyer-Moore (BM)
#####
#####

from typing import List, Tuple

class BoyerMoore:
    def __init__(self):
        self.name = "Boyer-Moore"

```

```

self._pattern_cache = {} # biar gak recompute yg udh ada

# @brief Menghitung tabel 'bad character' untuk pergeseran
# @param pattern: Pola string yang akan dianalisis
# @return: Tabel (list) yang memetakan setiap karakter ke posisi terakhirnya dalam pola
def _bad_char_heuristic(self, pattern: str) → list[int]:
    bad_char = [-1] * 256 # ascii table
    for i in range(len(pattern)):
        bad_char[ord(pattern[i])] = i

    return bad_char

# @brief Menghitung tabel 'good suffix' untuk pergeseran yang lebih optimal
# @param pattern: Pola string yang akan diproses
# @return: Tabel (list) yang berisi jarak pergeseran berdasarkan sufiks yang cocok
def _good_suffix_heuristic(self, pattern: str) → list[int]:
    m: int = len(pattern)
    suffix: list[int] = [0] * m
    good_suffix: list[int] = [m] * m

    # Compute suffix array
    suffix[m - 1] = m
    g: int = m - 1
    f: int = m - 1

    for i in range(m - 2, -1, -1):
        if i > g and suffix[i + m - 1 - f] < i - g:
            suffix[i] = suffix[i + m - 1 - f]
        else:
            if i < g:
                g = i
            f = i
            while g ≥ 0 and pattern[g] == pattern[g + m - 1 - f]:
                g -= 1
            suffix[i] = f - g

    # kasus 1: yang pernah muncul cuma prefix dari good suffix sbg suffix pattern
    j: int = 0
    for i in range(m - 1, -1, -1):
        if suffix[i] == i + 1:
            while j < m - 1 - i:
                if good_suffix[j] == m:
                    good_suffix[j] = m - 1 - i
                j += 1

    # kasus 2: good suffix pernah muncul
    for i in range(m - 1):
        good_suffix[m - 1 - suffix[i]] = m - 1 - i

    return good_suffix

# @brief Melakukan pra-pemrosesan pola dengan menghitung tabel bad char & good suffix
# @details Mengecek cache dulu, kalau polanya sudah pernah diproses, langsung kembalikan hasilnya biar gak recompute.
# @param pattern: Pola yang akan diproses
# @return: Tuple yang isinya tabel bad character dan tabel good suffix
def _preprocess_pattern(self, pattern: str) → tuple[list[int], list[int]]:
    pattern_key = (pattern, len(pattern))

    if pattern_key in self._pattern_cache:
        return self._pattern_cache[pattern_key]

```

```

bad_char = self._bad_char_heuristic(pattern)
good_suffix = self._good_suffix_heuristic(pattern)

self._pattern_cache[pattern_key] = (bad_char, good_suffix)
return bad_char, good_suffix

# @brief fungsi utama untuk algoritma Boyer-Moore
# @param text: Teks yang akan dicari
# @param pattern: Pola yang akan dicocokkan
# @return: List berisi semua posisi awal ditemukannya pola, kalau gaada ya kosong
def search(self, text: str, pattern: str) → list[int]:
    if not text or not pattern:
        return []

    n: int = len(text)
    m: int = len(pattern)

    if m > n:
        return []

    # Convert to lowercase for case-insensitive search
    text: str = text.lower()
    pattern: str = pattern.lower()

    # ambil preprocessing atau hitung baru
    bad_char, good_suffix = self._preprocess_pattern(pattern)

    positions = []
    s: int = 0 # shift of pattern with respect to text

    while s ≤ n - m:
        j = m - 1

        # Keep reducing index j while characters match
        while j ≥ 0 and pattern[j] == text[s + j]:
            j -= 1

        if j < 0:
            # Pattern found
            positions.append(s)

            # Shift pattern using good suffix heuristic
            s += good_suffix[0] if s + m < n else 1
        else:
            # Shift pattern using max of bad char and good suffix
            bad_char_shift = max(1, j - bad_char[ord(text[s + j])])
            good_suffix_shift = good_suffix[j]
            s += max(bad_char_shift, good_suffix_shift)

    return positions

# @brief Mencari kemunculan pertama dari pola dalam teks
# @param text: Teks yang akan dicari
# @param pattern: Pola yang akan dicocokkan
# @return: Posisi awal kemunculan pertama pola, atau -1 kalau gaada
def search_first(self, text: str, pattern: str) → int:
    n: int = len(text)
    m: int = len(pattern)
    if m > n:
        return -1

    bad_char, good_suffix = self._preprocess_pattern(pattern)

```

```

s: int = 0
while s ≤ n - m:
    j: int = m - 1

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

    if j < 0:
        return s # kemunculan pertama

    bad_char_shift = max(1, j - bad_char[ord(text[s + j])])
    good_suffix_shift = good_suffix[j]
    s += max(bad_char_shift, good_suffix_shift)
    return -1

# @brief Menghitung jumlah kemunculan pola dalam teks
# @param text: Teks yang akan dicari
# @param pattern: Pola yang akan dicocokkan
# @return: Jumlah total kemunculan pola
def count_occurrences(self, text: str, pattern: str) → int:
    return len(self.search(text, pattern))

# @brief Mencari beberapa pola sekaligus dalam satu teks
# @param text: Teks yang akan dicari
# @param patterns: List berisi pola-pola yang mau dicari
# @return: Dictionary yang isinya hasil pencarian, bentuknya pola → {posisi, jumlah}
def search_multiple(self, text, patterns):
    """
    Search for multiple patterns in text
    Returns dictionary with pattern as key and list of positions as value
    """
    results = {}
    for pattern in patterns:
        text = text.lower()
        pattern = pattern.lower()
        positions = self.search(text, pattern)
        if positions:
            results[pattern] = {
                'positions': positions,
                'count': len(positions)
            }
    return results

def clear_cache(self):
    self._pattern_cache.clear()

```

src\algorithms\aho_corasick.py

Kelas implementasi dari algoritma Aho-Corasick.

```

#####
#####
## @file aho_corasick.py
## Ini isinya implementasi algoritma Aho-Corasick (AC)
#####
#####

from collections import deque, defaultdict

```

```

from typing import List, Dict, Tuple, Any

## @brief Representasi sebuah node dalam automaton Aho-Corasick.
class AhoCorasick:
    def __init__(self):
        self.name = "Aho-Corasick"
        self._automaton_cache: Dict[frozenset, Any] = {}

    class Node:
        __slots__ = ['children', 'failure', 'output']
        def __init__(self):
            self.children = {}
            self.failure = None
            self.output = []

    # @brief Membangun automaton Aho-Corasick dari sekumpulan pola.
    # @param patterns: List pola (string) yang akan dimasukkan ke dalam automaton.
    # @return: Root node dari automaton yang telah dibangun.
    def build_automaton(self, patterns: List[str]) → Node:
        if not patterns:
            return None

        # cache key
        patterns_key = frozenset(p.lower() for p in patterns)
        if patterns_key in self._automaton_cache:
            return self._automaton_cache[patterns_key]

        # konversi pola ke tuple untuk menghindari masalah mutable types
        lower_to_original = {}
        for pattern in set(patterns):
            lower = pattern.lower()
            if lower not in lower_to_original:
                lower_to_original[lower] = pattern

        root = self.Node()

        # Membangun trie
        for i, (lower, original) in enumerate(lower_to_original.items()):
            node = root
            for char in lower:
                if char not in node.children:
                    node.children[char] = self.Node()
                node = node.children[char]
            node.output.append((i, original))

        # Membangun failure links menggunakan BFS
        queue = deque()

        # Inisialisasi failure links untuk anak-anak root
        for char, child in root.children.items():
            child.failure = root
            queue.append(child)

        # BFS untuk membangun failure links
        while queue:
            current = queue.popleft()
            for char, child in current.children.items():
                queue.append(child)

            # Cari failure link untuk child
            f = current.failure
            while f and char not in f.children:

```

```

        f = f.failure

        child.failure = f.children[char] if f and char in f.children else root

        # Gabungin output dari failure link ke child
        child.output = list(dict.fromkeys(child.output + child.failure.output))

    # Simpan ke cache
    self._automaton_cache[patterns_key] = root
    return root

# @brief Mencari satu pola dalam teks (untuk kompatibilitas dengan algoritma lain).
# @param text: Teks yang akan dicari.
# @param pattern: Pola tunggal yang akan dicocokkan.
# @return: List berisi posisi di mana pola ditemukan.
def search(self, text: str, pattern: str) → List[int]:
    results = self.search_multiple(text, [pattern])
    if pattern in results:
        return results[pattern]['positions']
    return []

# @brief Fungsi utama untuk mencari beberapa pola sekaligus secara efisien.
# @param text: Teks yang akan dicari.
# @param patterns: List pola yang akan dicocokkan.
# @return: Dictionary yang isinya <pola, <'positions', 'count'>>. Kalau kosong ya kosong.
def search_multiple(self, text: str, patterns: List[str], root=None) → Dict[str, Dict[str, Any]]:
    if not text or not patterns:
        return {}

    if root is None:
        root = self.build_automaton(patterns)
    if root is None:
        return {}

    # Search in text
    results = defaultdict(lambda: {'positions': [], 'count': 0})
    text_lower = text.lower()
    current = root

    for i, char in enumerate(text_lower):
        # Follow failure links until we find a match or reach root
        while current and char not in current.children:
            current = current.failure

        if current:
            current = current.children[char]
        else:
            current = root

        # Check all patterns that end at current position
        for _, pattern in current.output:
            start_pos = i - len(pattern) + 1
            results[pattern]['positions'].append(start_pos)
            results[pattern]['count'] += 1

    # Convert defaultdict to regular dict and filter empty results
    return {k: v for k, v in results.items() if v['positions']}

# @brief Menghitung jumlah kemunculan sebuah pola dalam teks.
# @param text: Teks yang akan dicari.
# @param pattern: Pola yang akan dihitung.
# @return: Jumlah kemunculan pola (integer).

```



```
def count_occurrences(self, text: str, pattern: str) → int:
    return len(self.search(text, pattern))

# @brief Alias untuk search_multiple untuk menekankan keunggulan Aho-Corasick.
# @param text: Teks yang akan dicari.
# @param patterns: List pola yang akan dicocokkan.
# @return: Hasil dari search_multiple.
def search_all_patterns(self, text: str, patterns: List[str]) → Dict[str, Dict[str, Any]]:
    return self.search_multiple(text, patterns)
```

src\algorithms\levenshtein.py

Kelas implementasi dari algoritma jarak Levenshtein.

```
#####
#####
## @file levenshtein.py
## Ini isinya implementasi algoritma Levenshtein Distance untuk
## pencarian fuzzy (fuzzy string matching).
#####
#####

import re
from collections import defaultdict
from typing import List, Dict, Tuple, Any, Optional

class LevenshteinDistance:
    def __init__(self, threshold: int = 0.65, cache_limit: int = 1000):
        self.threshold = threshold # Similarity threshold (0.7 = 70% similar)
        self._distance_cache = {} # Cache untuk menghindari perhitungan berulang
        self.cache_limit = 1000
        self._word_cleaner = re.compile(r'^\w\s+', re.UNICODE)
        self._whitespace = re.compile(r'\s+')
        self._access_counter = 0

    def _evict_cache(self):
        if len(self._distance_cache) ≥ self.cache_limit:
            lru_key = min(self._distance_cache.keys(), key=lambda k: self._distance_cache[k][1])
            del self._distance_cache[lru_key]

    # @brief Menghitung jarak Levenshtein antara dua string.
    # @param s1: String pertama.
    # @param s2: String kedua.
    # @return: Jarak Levenshtein (integer) antara s1 dan s2.
    def calculate_distance(self, s1, s2, max_distance: Optional[int] = None):
        if not s1:
            return len(s2)
        if not s2:
            return len(s1)
        if s1 == s2:
            return 0

        # Convert to lowercase for case-insensitive comparison
        s1 = s1.lower()
        s2 = s2.lower()
        m, n = len(s1), len(s2)

        # cek cache
        cache_key = (s1, s2, max_distance)
```

```

if cache_key in self._distance_cache:
    self._access_counter += 1
    value, _ = self._distance_cache[cache_key]
    self._distance_cache[cache_key] = (value, self._access_counter)
    return value

# biar efisien, kalau panjang s1 < s2, tukar aja
if m < n:
    return self.calculate_distance(s2, s1)

# skrg amanah m ≥ n
prev_row = list(range(n + 1))
curr_row = [0] * (n + 1)

for i in range(1, m + 1):
    curr_row[0] = i
    diagonal_min = float('inf')

    for j in range(1, n + 1):
        if s1[i-1] == s2[j-1]:
            curr_row[j] = prev_row[j-1]
        else:
            curr_row[j] = 1 + min(
                prev_row[j],      # deletion
                curr_row[j-1],    # insertion
                prev_row[j-1]     # substitution
            )

        diagonal_min = min(diagonal_min, curr_row[j])

    if max_distance is not None and diagonal_min > max_distance:
        return max_distance + 1

# Swap rows
prev_row, curr_row = curr_row, prev_row

# Simpan hasil ke cache
self._evict_cache()
self._access_counter += 1
self._distance_cache[cache_key] = (prev_row[n], self._access_counter)
return prev_row[n]

# @brief Menghitung persentase kemiripan antara dua string.
# @param s1: String pertama.
# @param s2: String kedua.
# @return: Skor kemiripan (float) antara 0.0 dan 1.0.
def calculate_similarity(self, s1, s2):
    if not s1 and not s2:
        return 1.0

    max_len = max(len(s1), len(s2))
    max_distance = int(max_len * (1 - self.threshold)) + 1
    distance = self.calculate_distance(s1, s2, max_distance)

    if max_len == 0:
        return 1.0

    similarity = 1 - (distance / max_len)
    return similarity

# @brief Melakukan pencarian fuzzy untuk beberapa kata kunci sekaligus.
# @param text: Teks sumber untuk pencarian.

```

```

# @param keywords: List kata kunci yang akan dicari.
# @param min_similarity: Ambang batas kemiripan minimum.
# @return: Dictionary yang key-nya adalah kata kunci dan value-nya list kata-kata mirip yang ditemukan.
def fuzzy_search(self, text: str, keywords: List[str], min_similarity: float = None) → Dict[str, List[Dict[str, Any]]]:
    if min_similarity is None:
        min_similarity = self.threshold

    results = defaultdict(list)

    # preprocessing
    processed_keywords = [(kw.lower(), len(kw.split())) for kw in keywords]

    # token
    word_matches = list(re.finditer(r'\b[\w#+.-]+\b', text))
    words = [m.group(0) for m in word_matches]
    positions = [m.start() for m in word_matches]

    # Buat token windows untuk setiap kata kunci
    max_kw_len = max(length for _, length in processed_keywords)

    token_windows = []
    for window_size in range(1, max_kw_len + 1):
        for i in range(len(words) - window_size + 1):
            phrase = ' '.join(words[i:i + window_size])
            token_windows.append({
                'word': phrase,
                'position': positions[i]
            })

    # cache untuk menghindari perhitungan berulang
    similarity_cache = {}

    for keyword, kw_len in processed_keywords:
        for token in token_windows:
            key_pair = (token['word'], keyword)
            if key_pair in similarity_cache:
                similarity = similarity_cache[key_pair]
            else:
                similarity = self.calculate_similarity(token['word'], keyword)
                similarity_cache[key_pair] = similarity

            if similarity ≥ min_similarity:
                results[keyword].append({
                    'word': token['word'],
                    'similarity': similarity,
                    'position': token['position']
                })

    return results

# @brief Mengatur nilai ambang batas kemiripan.
# @param threshold: Nilai ambang batas baru (antara 0.0 dan 1.0).
# @return: None.
def set_threshold(self, threshold):
    if 0.0 ≤ threshold ≤ 1.0:
        self.threshold = threshold
    else:
        raise ValueError("Threshold must be between 0.0 and 1.0")

# test driver

```

```

if __name__ == "__main__":
    import time
    ld = LevenshteinDistance()
    print(f"threshold: {ld.threshold}\n")

    # Contoh 1: Typo sederhana (substitusi)
    s1, s2 = "python", "pyhton"
    distance1 = ld.calculate_distance(s1, s2)
    similarity1 = ld.calculate_similarity(s1, s2)
    print(f"Jarak antara '{s1}' dan '{s2}': {distance1}")
    print(f"Tingkat kemiripan: {similarity1:.2f}\n")

    # Contoh 2: Perbedaan panjang (insersi/delesi)
    s3, s4 = "java", "javascript"
    distance2 = ld.calculate_distance(s3, s4)
    similarity2 = ld.calculate_similarity(s3, s4)
    print(f"Jarak antara '{s3}' dan '{s4}': {distance2}")
    print(f"Tingkat kemiripan: {similarity2:.2f}\n")

    # 3. Uji coba kasus penggunaan utama: fuzzy_search
    # Ini adalah skenario yang paling relevan untuk aplikasi ATS Anda.
    print("---- 2. Uji Coba fuzzy_search pada Teks CV ----")

    sample_cv_text = """
    Nama: NAYAKA
    Pengalaman: Software Engineer dengan fokus pada python dan pengembangan web.
    Mahir dalam Java, Javascript, dan C++. Mencari posisi sebagai 'project manager'.
    Saya juga pernah mengerjakan proyek dengan nod.js.
    """

    keywords_to_find = ["python", "java", "node.js", "project management", "C#"]

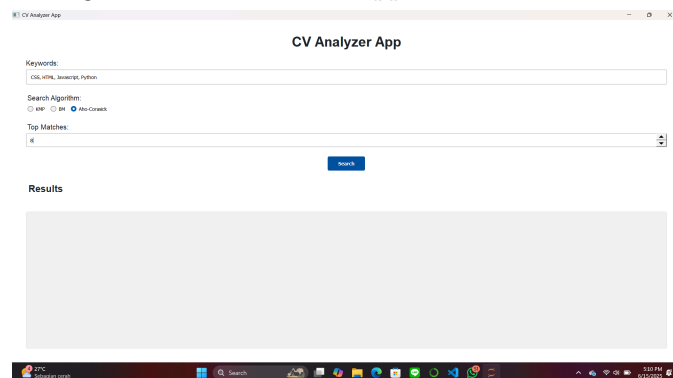
    print(f"Teks CV yang akan dipindai:\n---\n{sample_cv_text.strip()}\n---\n")
    print(f"Kata kunci yang dicari: {keywords_to_find}")
    print(f"Threshold kemiripan yang digunakan: {ld.threshold}")
    print("\n>>> Hasil Pencarian Fuzzy:")

    # Panggil metode fuzzy_search yang sudah dioptimalkan
    # (menggunakan re.finditer untuk mendapatkan posisi)
    start_time = time.time()
    fuzzy_results = ld.fuzzy_search(sample_cv_text, keywords_to_find)
    end_time = time.time()
    print(f"\nWaktu eksekusi: {end_time - start_time:.4f} detik\n")
    if not fuzzy_results:
        print("Tidak ada kata yang mirip ditemukan.")
    else:
        # Cetak hasilnya dengan rapi
        for keyword, matches in fuzzy_results.items():
            print(f"\n Kata Kunci '{keyword}' ditemukan mirip dengan:")
            for match in matches:
                print(
                    f"    - Kata: '{match['word']}' "
                    f"(Kemiripan: {match['similarity']:.2f}, "
                    f"Posisi: {match['position']})"
                )

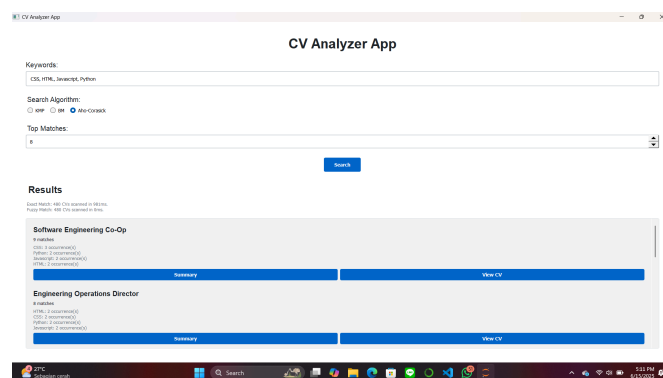
```

4.2. Tata Cara Penggunaan Aplikasi

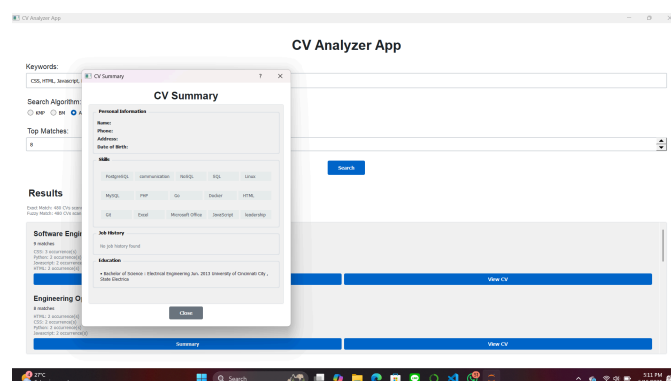
1. Mulai instance aplikasi dengan langkah yang ada di readme
2. Pada kolom “Keywords”, masukkan kueri yang ingin dicari
3. Pada kolom “Search Algorithm”, pilih algoritma yang ingin dipakai
4. Pada kolom “Top Matches”, masukkan berapa CV yang ingin dimunculkan
5. Tekan tombol “search”
6. Jika ingin melihat summary applicant, tekan tombol “summary”
7. Jika ingin melihat CV asli dari applicant, tekan tombol “view CV”



Contoh masukkan



Contoh luaran



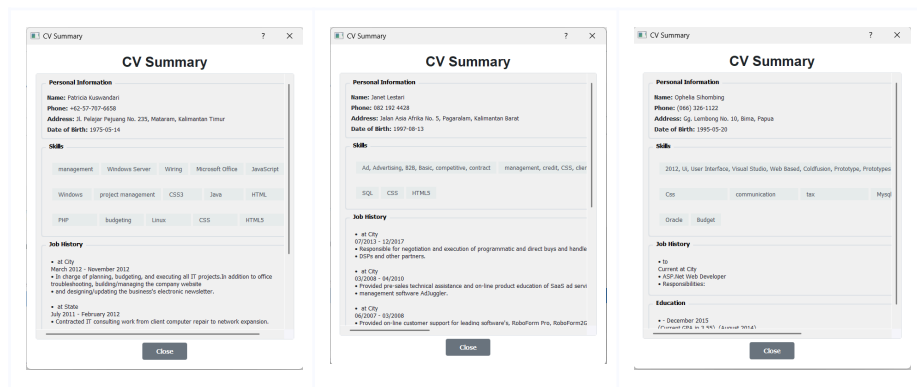
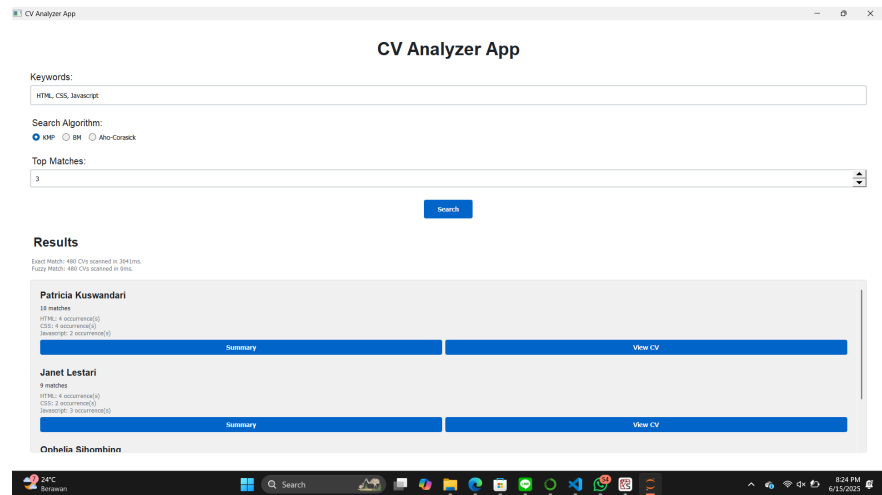
Contoh summary

4.3. Pengujian

4.3.1. Kasus Uji 1: KMP (1)

Keywords : HTML, CSS, Javascript

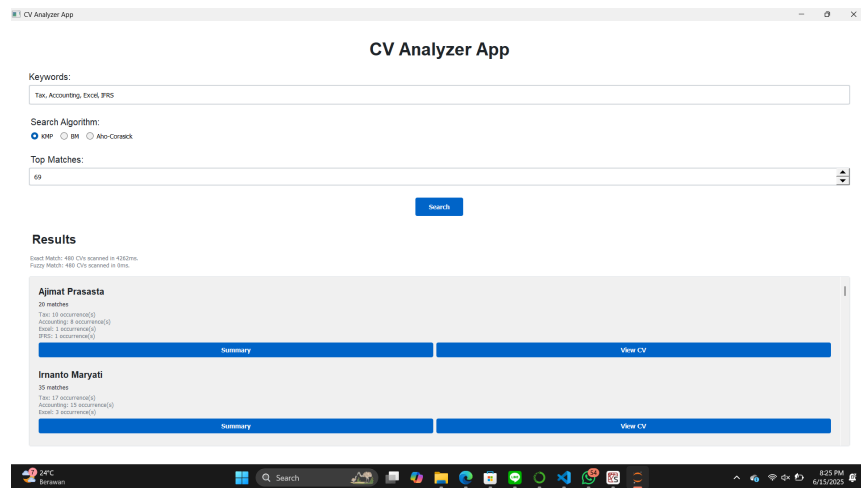
Top Matches : 3



4.3.2. Kasus Uji 1: KMP (2)

Keywords : Tax, Accounting, Excel, IFRS

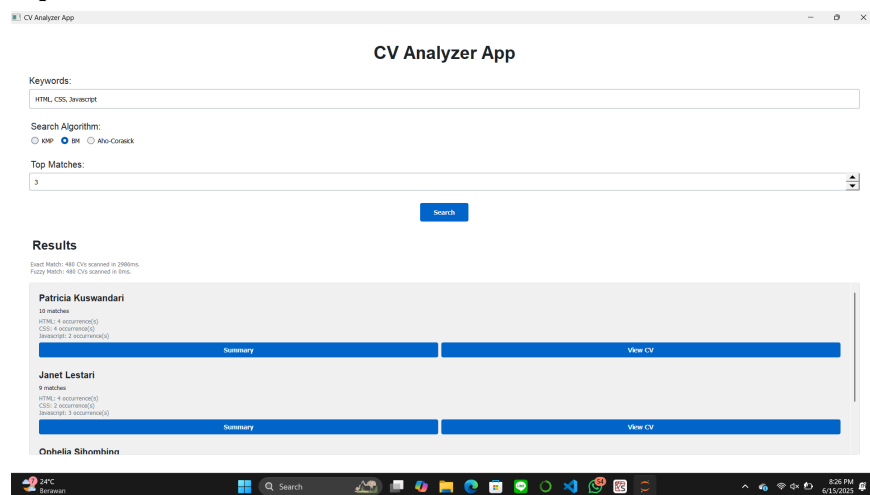
Top Matches : 69



4.3.3. Kasus Uji 2: BM (1)

Keywords : HTML, CSS, Javascript

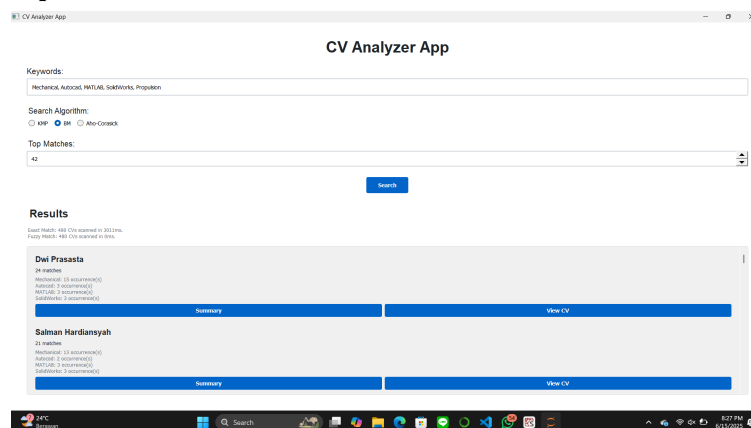
Top Matches : 3



4.3.4. Kasus Uji 2: BM (2)

Keywords : Mechanical, Autocad, MATLAB, SolidWorks, Propulsion

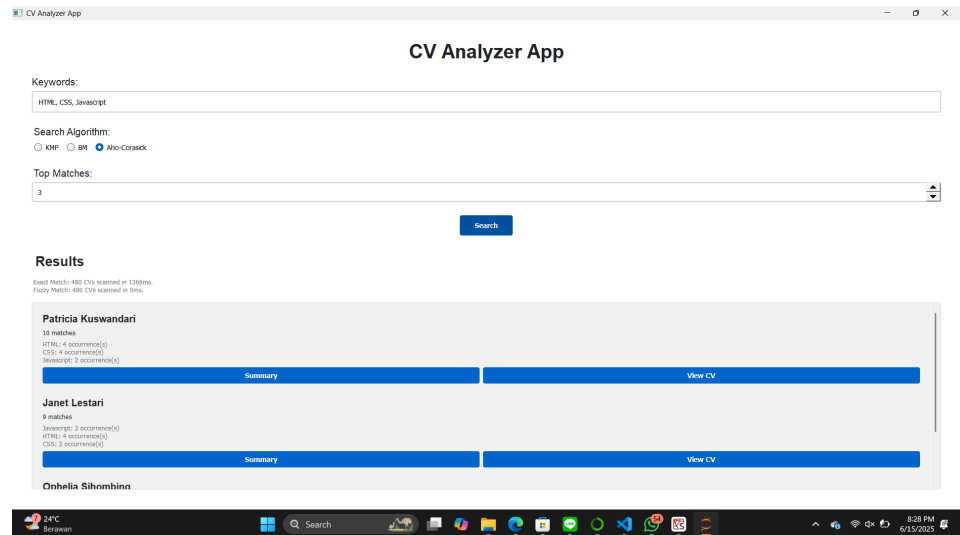
Top Matches : 42



4.3.5. Kasus Uji 3: AC (1)

Keywords : HTML, CSS, Javascript

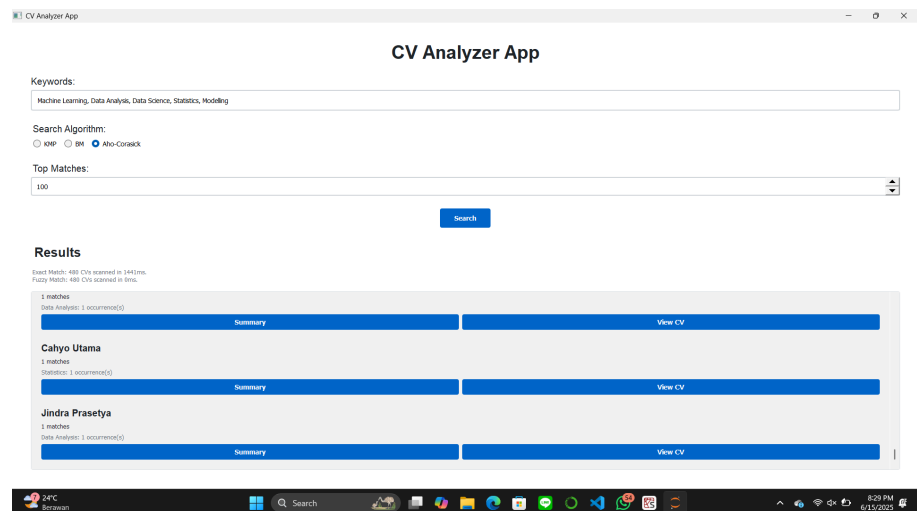
Top Matches : 3



4.3.6. Kasus Uji 3: AC (2)

Keywords : Machine Learning, Data Analysis, Data Science, Statistics, Modelling

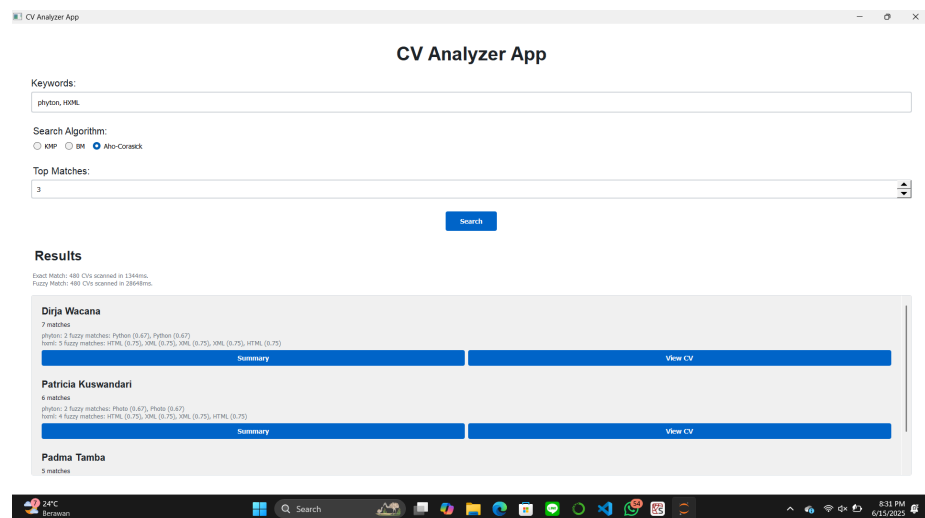
Top Matches : 100



4.3.7. Kasus Uji 4: Levenshtein Distance

Keywords : pythonn, HXML

Top Matches : 3



4.4. Analisis

Pengujian perbandingan algoritma pencarian string (KMP, Boyer-Moore, Aho-Corasick) dilakukan dengan parameter identik, yaitu kata kunci "HTML, CSS, Javascript" dan batasan tiga hasil teratas (Top Matches 3). Hasil pengujian menunjukkan bahwa Aho-Corasick adalah yang tercepat dengan 1366ms, diikuti oleh Boyer-Moore dengan 2986ms, dan KMP menjadi yang terlambat dengan 3041ms. Temuan ini konsisten dengan teori efisiensi algoritma pencarian *string*, di mana Aho-Corasick lebih unggul dari Boyer-Moore, dan Boyer-Moore lebih efisien dari KMP, terutama untuk pencarian multi-pola atau teks panjang. Data empiris ini mendukung bahwa Aho-Corasick adalah pilihan optimal untuk skenario pencarian berkecepatan tinggi.

Dari test case levenshtein distance juga dapat terlihat bahwa algoritma *fuzzy matching* yang dilakukan dapat dengan efektif menangkap string yg “mirip” dengan kueri. Pada kasus uji tersebut diberikan kata kunci “phyton” dan “HXML”, dan algoritma jarak Levenshtein berhasil menemukan string yang mirip dengan keywords tersebut yaitu “python” (dengan kemiripan 86% dengan “phyton”), “HTML” (dengan kemiripan 75% dengan “HXML”), dan “XML” (dengan kemiripan 75% dengan “HXML”).

Dari hasil pengujian diperoleh juga bahwa variasi parameter “Top Matches” tidak berpengaruh terhadap waktu pencarian, maupun pemilihan kandidat, karena cara kerja algoritma yang melakukan pencarian pada semua CV di database lepas dari parameter “Top Matches”

5. Penutup

5.1. Kesimpulan

Dalam Tugas Besar ini berhasil dikembangkan sebuah aplikasi *Applicant Tracking System* (ATS) yang fungsional untuk mempermudah proses rekrutmen. Aplikasi ini mampu mengekstrak informasi dari CV berformat PDF dengan regex, melakukan pencarian kata kunci, dan menampilkan ringkasan profil pelamar secara efisien.

Tiga algoritma pencocokan string, yaitu Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), dan Aho-Corasick (AC), berhasil diimplementasikan untuk fitur pencarian kata kunci. Pengujian performa menunjukkan bahwa algoritma Aho-Corasick merupakan yang paling cepat untuk pencarian multi-pola, diikuti oleh Boyer-Moore dan KMP, yang sejalan dengan teori kompleksitas algoritma.

5.2. Saran

| |
|---|
| <i>Farrel Athalla Putra</i> |
| Semoga tugas besar STIMA terus memberikan kejutan untuk seterusnya! |

| |
|--|
| <i>Adiel Rum</i> |
| Tugas besar stima tak pernah gagal menantang mahasiswanya! |

| |
|--------------------------------------|
| <i>Zulfaqqar Nayaka Athadiansyah</i> |
| Gacor king. |

5.3. Refleksi

| |
|--|
| <i>Farrel Athalla Putra</i> |
| Pembuatan tugas besar di tengah UAS membutuhkan pembagian waktu ekstra. Terima kasih teman sekelompok yang sudah bekerja sama! |

| |
|---|
| <i>Adiel Rum</i> |
| Terima kasih Farrel dan Nay sudah adopt saya XD |

| |
|--------------------------------------|
| <i>Zulfaqqar Nayaka Athadiansyah</i> |
|--------------------------------------|

Mantap Farrel hyper carry

Lampiran

Tautan repositori GitHub: https://github.com/nayakazna/Tubes3_CVMagang.git

Tautan video bonus: https://youtu.be/Shn_ZGC64fw

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

Daftar Pustaka

S. Halim dan F. Halim, *Competitive Programming 3: The New Lower Bound of Programming Contests; Handbook for ACM ICPC and IOI Contestants*, 2013.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, dan C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.