# Laporan Tugas Besar 3
# Strategi Algoritma

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

## Kelompok 30
## MeACavemanDoesntHaveReligion

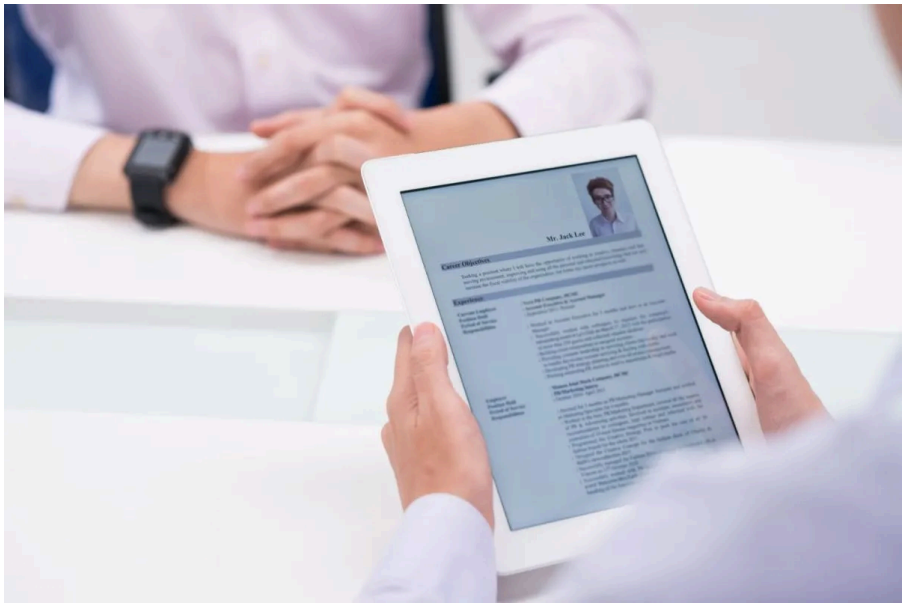

Anggota Kelompok:
Orvin Andika I A (13523017)
Ferdinand Gabe Tua S (13523051)
Salman Hanif (13523056)

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

**Bandung 2025**

# Daftar Isi

# Bab 1
# Deskripsi Tugas



**Gambar 1.** CV ATS dalam Dunia Kerja
(Sumber: https://www.antaranews.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.

# Bab 2
# Landasan Teori

## 2.1. Algoritma Knuth-Morris-Pratt

Algoritma KMP mencari pola dalam teks dari kiri ke kanan seperti algoritma *brute force.* Akan tetapi, algoritma KMP bergeser secara lebih pintar daripada algoritma *brute force.* Algoritma *brute force* akan bergeser ke kanan satu langkah setiap terjadi *mismatch,* sedangkan KMP akan bergeser sesuai dengan fungsi pinggirannya.

Fungsi pinggiran KMP akan menghitung subpola terbesar yang merupakan prefiks dan juga sufiks dari teks di posisi terjadinya *mismatch.* Misalnya, pola adalah abacab dan *mismatch* terjadi di indeks 4, maka algoritma akan bergeser jauh ukuran dari prefiks terbesar abaca yang juga sufiks dari baca, yaitu a. Maka, algoritma akan bergeser sehingga indeks 1 pola akan sejajar dengan tempat terjadinya *mismatch* dan pengecekan juga akan dimulai dari posisi itu. Pergeseran pintar ini membuat KMP lebih cepat dibandingkan *brute force.* Untuk menghitung fungsi pinggiran membutuhkan kompleksitas O(m) dan untuk pencarian *string* membutuhkan O(n) sehingga kompleksitasnya secara keseluruhan adalah O(m+n).

KMP memiliki keunggulan dan kekurangan tersendiri. Keunggulannya adalah algoritma ini tidak perlu mundur dalam pengecekan teks. Hal ini membuat KMP cocok digunakan untuk pengecekan *file* besar yang berasal dari sumber eksternal. Sementara itu, kekurangannya adalah algoritma ini tidak cocok untuk teks yang memiliki banyak variasi karakter. Semakin banyak karakter maka akan semakin banyak kemungkinan *mismatch.* Banyaknya variasi karakter juga membuat pengecekan cenderung gagal di awal, padahal KMP bagus jika pengecekan gagal di akhir.

## 2.2. Algoritma Boyer-Moore

Algoritma Boyer-Moore adalah algoritma *pattern matching* yang berdasarkan pada dua teknik, yaitu teknik *looking-glass*, yaitu mencari pola dalam teks dengan berjalan mundur dari akhir pola. Jika ada *mismatch*, akan ada tiga skenario yang harus dipilih berdasarkan situasi. Situasi pertama, jika karakter di teks yang tidak cocok ada di pola dan kemunculan terakhirnya di pola lebih kecil dari posisi sekarang, maka pola digeser hingga karakter tersebut sejajar dengan karakter yang sama di pola. Situasi kedua, jika jika karakter di teks yang tidak cocok ada di pola dan kemunculan terakhirnya di pola lebih kecil dari posisi sekarang, maka pola digeser ke kanan satu posisi. Terakhir, jika karakter yang tidak cocok tidak ada di pola, maka pola digeser sehingga awal pola berada di indeks ketidakcocokan ditambah satu.

Algoritma Boyer-Moore membutuhkan tabel yang berisi indeks kemunculan terakhir dari karakter dalam teks. Indeks kemunculan terakhir maksudnya posisi terakhir karakter tersebut dalam pola. Jika karakter tidak ada di dalam pola, maka akan diberi nilai -1. Algoritma ini bagus untuk teks dengan variasi karakter yang

besar karena semakin besar variasi karakter semakin besar kemungkinan karakter tersebut tidak ada di pola sehingga pergeseran akan lebih jauh.

## 2.3. Algoritma Aho-Corasick

Algoritma Aho-Corasick adalah algoritma yang dirancang untuk mencari banyak pola sekaligus dalam satu kali proses pembacaan teks. Algoritma ini memindai teks dari kiri ke kanan dan secara efisien menemukan semua kemunculan dari setiap kata kunci yang diberikan.

Mekanisme pintar Aho-Corasick terletak pada struktur data yang dibangunnya sebelum pencarian dimulai. Algoritma ini menggabungkan semua kata kunci ke dalam sebuah struktur data tunggal yang menyerupai *trie*, yang disebut *finite automaton*. Setelah pohon ini terbentuk, Aho-Corasick menambahkan *failure link* ke setiap simpul. Jika terjadi *mismatch* saat traversal pohon, alih-alih memulai ulang dari awal, algoritma akan mengikuti *failure link* ini ke prefiks terpanjang berikutnya yang valid yang juga ada di dalam kamus. Hal ini memungkinkan algoritma untuk terus maju, seolah-olah menjalankan puluhan KMP secara paralel.

Kompleksitas algoritma Aho-Corasick sangat efisien. Proses membangun *automaton* membutuhkan waktu O(L), di mana L adalah total panjang semua kata kunci. Proses pencarian pada teks membutuhkan waktu O(n + k), di mana n adalah panjang teks dan k adalah jumlah total kecocokan yang ditemukan. Jadi, kompleksitas keseluruhannya adalah O(L + n + k).

Aho-Corasick memiliki keunggulan dan kekurangan. Keunggulan terbesarnya adalah efisiensinya yang luar biasa untuk mencari banyak kata kunci dalam sekali jalan, menjadikannya ideal untuk aplikasi seperti analisis CV di mana kita mencari banyak *skill* sekaligus. Sementara itu, kekurangannya adalah adanya biaya awal untuk membangun *automaton*. Jika hanya ada satu kata kunci, KMP mungkin lebih cepat karena tidak ada *overhead* pembangunan struktur data yang kompleks.

## 2.4. Algoritma Levenshtein Distance

Algoritma Levenshtein Distance adalah metrik untuk mengukur jarak edit antara dua string. Algoritma ini menghitung jumlah minimum operasi penyisipan, penghapusan, dan penggantian karakter yang diperlukan untuk mengubah string pertama menjadi string kedua.

Cara kerja Levenshtein Distance didasarkan pada program dinamis. Algoritma ini membangun sebuah matriks dengan dimensi (panjang string 1 + 1) x (panjang string 2 + 1). Setiap sel dalam matriks ini diisi dengan nilai jarak Levenshtein antara i karakter pertama dari string 1 dan j karakter pertama dari string 2. Nilai setiap sel dihitung dengan mengambil nilai minimum dari tiga kemungkinan: biaya penghapusan (nilai sel di atas + 1), biaya penyisipan (nilai sel di kiri + 1), atau biaya penggantian (nilai sel di diagonal kiri atas + 0 jika karakternya sama, atau + 1 jika

berbeda). Jarak total antara kedua string adalah angka yang ada di sel pojok kanan bawah matriks.

Karena harus membangun dan mengisi seluruh matriks, kompleksitas Levenshtein Distance adalah O(m * n), di mana m dan n adalah panjang dari kedua string yang dibandingkan. Keunggulan utama algoritma ini adalah kemampuannya dalam melakukan pencocokan fuzzy, yaitu mengenali kesamaan antara dua string meskipun terdapat kesalahan ketik (typo).

## 2.5. Regular Expression

Regular Expression adalah sebuah bahasa formal untuk mendefinisikan pola pencarian teks yang kuat dan fleksibel. Regex memungkinkan pencarian yang lebih kompleks daripada sekadar kata kunci literal. Di balik layar, sebuah *engine* Regex akan menerjemahkan pola yang diberikan pengguna menjadi sebuah mesin *Finite Automaton*, biasanya *Nondeterministic Finite Automaton*. *Engine* ini kemudian memproses teks input karakter per karakter, berpindah dari satu keadaan ke keadaan lain di dalam *automaton* sesuai dengan aturan pola. Sebuah kecocokan ditemukan jika *engine* berhasil mencapai *accept state*.

Keunggulan terbesar Regex adalah fleksibilitasnya untuk mendefinisikan hampir semua jenis pola teks yang bisa dibayangkan. Di sisi lain, kekurangannya adalah sintaksnya yang bisa menjadi sangat rumit dan sulit dibaca

# 2.6. Pembangunan Aplikasi Desktop

Aplikasi desktop ini dibangun untuk membantu pencarian kata kunci yang sesuai dalam CV kandidat. Aplikasi ini akan menampilkan kandidat yang memiliki kecocokan terbesar dengan kata kunci. Pencarian kecocokan dibagi menjadi dua tipe yaitu *exact matching* dan *fuzzy matching*. Untuk *exact matching,* aplikasi ini memanfaatkan algoritma KMP, Boyer-Moore, dan Aho Corasick. Semantara itu, untuk *fuzzy matching,* program ini memanfaatkan algoritma Levenshtein Distance. Di sisi lain, untuk mengekstrak informasi seperti *skill,* riwayat pendidikan, dan pengalaman kerja, program ini memanfaatkan Regular Expression.

Aplikasi desktop ini dibangun sepenuhnya dalam bahasa Python. Untuk GUI, aplikasi ini menggunakan framework TKinter.Framework ini di.Di sisi lain, penyimpanan data CV dan data kandidat memanfaatkan basis data yang berbasis SQL.

# Bab 3
# Analisis Pemecahan Masalah

## 3.1. Langkah-Langkah Pemecahan Masalah

Masalah yang ingin dipecahkan program ini adalah pencarian CV yang memiliki kecocokan terbanyak dengan kata kunci yang dimasukkan pengguna. Langkah pemecahan masalah diawali dengan input kata kunci yang dicari pengguna. Pengguna juga bisa memilih algoritma yang ingin digunakan. Algoritma yang tersedia adalah KMP, Boyer-Moore, dan Aho-Corasick. Pengguna juga harus memasukkan jumlah kandidat teratas yang ingin ditampilkan. Setelah semua terisi, pengguna bisa menekan tombol *search.*

Program akan memulai dengan mengubah CV menjadi sebuah *flat text* panjang. Jika basis data dienkripsi, program akan menjalankan algoritma dekripsi terlebih dahulu. Kemudian, program akan melakukan proses *pattern matching* sesuai dengan algoritma yang dipilih. Jika algoritma yang dipilih adalah KMP atau BM maka pencocokan kata kunci dilakukan satu per satu. Sementara itu, jika menggunakan Aho Corasick maka pencocokan seluruh kata kunci akan dilakukan bersamaan. Jika jumlah CV yang memiliki kecocokan sama atau lebih dari jumlah yang diinginkan pengguna, maka program hanya akan menampilkan n kandidat teratas, dengan n adalah jumlah yang diinput pengguna. Jika hasilnya kurang dari n, maka program akan menjalankan algoritma yang mengukur kecocokan dengan memanfaatkan Levenshtein Distance. Pengguna bisa melihat ringkasan dari CV pengguna atau *file* asli CV. Ringkasan dibentuk dengan memanfaatkan Regular Expression.

## 3.2. Pemetaan Masalah

Untuk menggunakan algoritma KMP, BM, dan Aho Corasick, diperlukan sebuah teks dan pola yang ingin dicari dalam teks tersebut. Pada kasus ini, teks yang digunakan adalah isi CV dan pola yang digunakan adalah input kata kunci pengguna yang dipisahkan oleh tanda koma.

Agar bisa menggunakan algoritma KMP, BM, dan Aho Corasick dengan optimal, diperlukan sebuah fungsi untuk menghasilkan sebuah *flat text* panjang dari *file* CV. Program juga harus *parsing* input pengguna menjadi sebuah array berisi kata kunci. Isi dari array dan *flat text* yang sudah dihasilkan akan dijadikan argumen untuk algoritma KMP dan BM. Hal ini dilakukan untuk semua isi array satu per satu. Untuk algoritma Aho Corasick, kecocokan seluruh kata kunci bisa dicari dalam satu waktu.

## 3.3. Fitur Fungsional dan Arsitektur Aplikasi

Aplikasi desktop ini dirancang untuk membantu pengguna untuk mencari CV yang sesuai dengan kata kunci yang dimasukkan pengguna. Aplikasi ini menyediakan tiga algoritma untuk melakukan *pattern matching.* Algoritma-algoritma itu adalah algoritma Knuth-Morris-Pratt, Boyer-Moore, dan Aho-Corasick. Aplikasi ini juga memberikan

fleksibilitas agar pengguna bisa menentukan berapa CV kandidat yang harus ditampilkan. Aplikasi ini juga mendukung *fuzzy matching* dengan memanfaatkan Levenshtein Distance jika hasil dari *exact matching* masih kurang dari jumlah yang diinginkan pengguna. Pengguna juga bisa melihat ringkasan dari CV maupun melihat *file* asli CV tersebut.

Aplikasi ini dibangun sepenuhnya dengan bahasa Python. Framework yang digunakan untuk GUI adalah Tkinter. Tkinter sudah termasuk dalam pustaka standar Python sehingga tidak perlu melakukan instalasi tambahan. Tkinter juga dipilih karena kemudahannya baik dari segi penggunaan maupun pembangunan aplikasi. Untuk penyimpanan data, aplikasi ini menggunakan *database* yang berbasis MySQL. *Database* ini digunakan untuk menyimpan data para kandidat dan *path* penyimpanan CV di perangkat.

## 3.4. Contoh Ilustrasi Kasus

### 3.4.1. KMP

Demi memperjelas bagaimana algoritma ini bekerja kita dapat mengamati contoh kasus dimana kita akan mencari keberadaan sebuah pola seperti dibawah ini

A B A B C A B A B

pada sebuah text dengan pola seperti dibawah ini

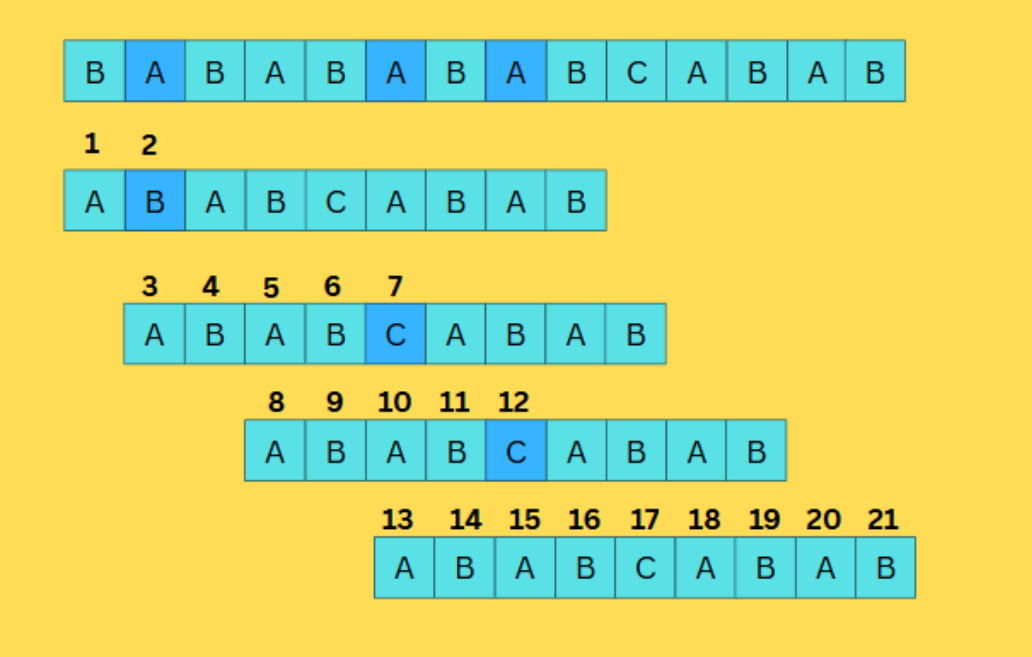
B A B A B A B A B C A B A B

Dalam algoritma Knuth-Morris-Pratt (KMP), demi menemukan suatu pola dalam teks diperlukan sebuah tabel preproses yang dikenal sebagai *longest prefix sufix table*. Tabel ini merepresentasikan panjang prefiks terpanjang dari pola. Tujuan utama dari tabel ini adalah untuk menghindari pemeriksaan ulang terhadap karakter-karakter yang telah dibandingkan sebelumnya ketika terjadi ketidaksesuaian. Berikut ini disajikan tabel pre proses yang digunakan untuk menemukan text itu.

| A | B | A | B | C | A | B | A | B |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 0 | 1 | 2 | 3 | 4 |

Gambar table longest prefix sifix table

Dengan memanfaatkan table tersebut kita akan lebih mudah menyelesaikannya, berikut ilustrasinya:




Gambar ilustrasi iterasi menggunakan algoritma KM

Seperti yang terlihat pada contoh kasus di atas, algoritma ini akan memulai proses pencocokan dari kiri ke kanan. Ketika terjadi ketidaksesuaian karakter antara teks dan pola, algoritma KMP akan memanfaatkan tabel LPS (Longest Proper Prefix which is also a Suffix) yang telah dibangun sebelumnya untuk menentukan seberapa banyak pergeseran pola yang optimal perlu dilakukan. Sebagai contoh, dalam ilustrasi di atas, pada suatu iterasi (misalnya, iterasi yang ditunjukkan pada bagian kedua, di mana terjadi pemeriksaan ke-7 pada teks), jika ketidaksesuaian terdeteksi pada indeks ke-4 pola, algoritma akan menggeser pola sejauh nilai yang ada di lps_table[j-1], di mana j adalah indeks terjadinya ketidaksesuaian di pola. Kembali ke contoh, hasil dari lps_table[j-1] adalah 2, maka pola akan bergeser ke kanan sebanyak 2 posisi, dan pencocokan dilanjutkan dari titik yang baru (lihat ilustrasi iterasi ke 3).

### 3.4.2. BM

Untuk mendapatkan pemahaman lebih lanjut mengenai algoritma ini kita dapat mengamati contoh kasus dimana kita akan mencari keberadaan sebuah pola T A T G T G pada sebuah text dengan pola seperti dibawah ini

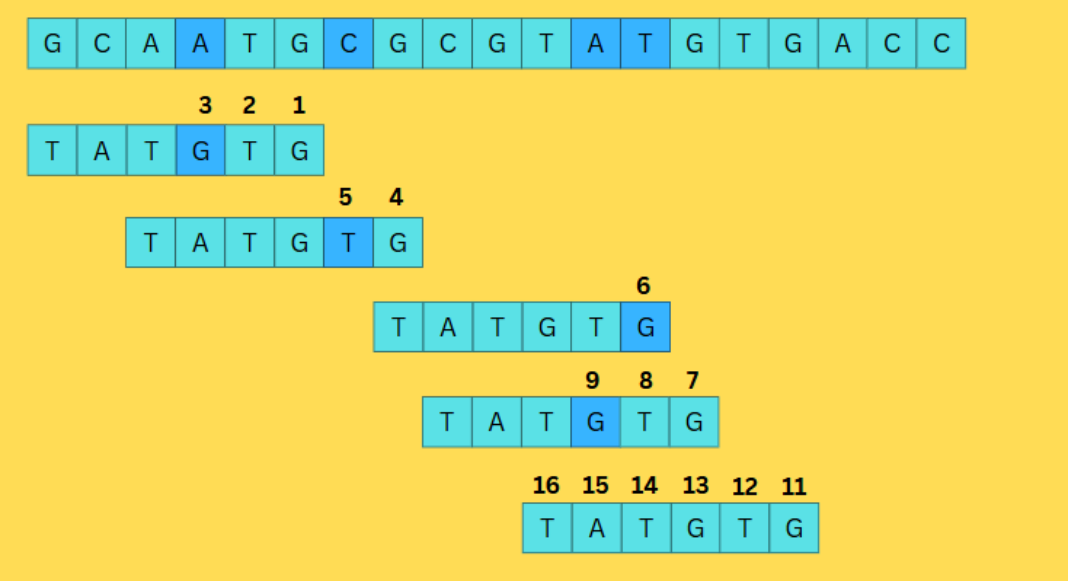
G C A A T G C G C G T A T G T G A C C

Untuk menemukan pola tersebut pada teks, algoritma ini pertama-tama akan membuat semacam tabel prekomputasi bernama *Last Occurrence Table* yang berguna untuk menentukan seberapa jauh pola dapat digeser saat terjadi ketidakcocokan karakter dalam proses pencocokan. Tabel ini mencatat posisi terakhir kemunculan setiap karakter dalam pola. Dengan menggunakan informasi dari tabel ini, algoritma dapat menghindari pemeriksaan ulang karakter yang sudah dipastikan tidak cocok, sehingga meningkatkan efisiensi pencarian. Berikut tablenya:

| T | A | G |
|---|---|---|
| 4 | 1 | 5 |

Gambar table last-occurance

Dengan memanfaatkan table tersebut kita akan lebih mudah menyelesaikannya, berikut ilustrasinya:




Gambar ilustrasi iterasi menggunakan algoritma Boyer Moore

Ketika terjadi ketidaksesuaian (mismatch), algoritma akan mendeteksi posisi indeks di mana ketidaksesuaian tersebut terjadi, kemudian melakukan pergeseran pola (shift) ke kanan sebanyak *n* karakter. Nilai *n* ini dihitung dengan menggunakan rumus berikut:

$$n=max(1,mismatch\_index-last\_occurrence[karakter])$$

Contoh nyata dapat dilihat pada ilustrasi sebelumnya, di mana pada iterasi pertama, ketidaksesuaian terjadi pada karakter ketiga yang diperiksa. Berdasarkan rumus di atas, algoritma menghitung jumlah pergeseran dan menghasilkan nilai *n = 2*, sehingga pola digeser sejauh dua posisi dari titik awal pada iterasi kedua.

Namun, apabila karakter yang menyebabkan ketidaksesuaian tidak ditemukan dalam tabel *last occurrence*, maka algoritma akan menggunakan nilai *-1* sebagai posisi terakhir kemunculan karakter tersebut. Dengan demikian, jumlah pergeseran dihitung sebagai:

$$n=max(1,mismatch\_index-(-1)$$

### 3.4.3. Levenshtein

Algoritma pencocokan levenshtein ini bekerja dengan dengan melakukan perhitungan levenshtein distance, yaitu nilai tingkatan perbedaan antara 2 kata yang didasari dengan penambahan, penghapusan, dan penggantian karakter dari kata yang dicocokkan. Dalam pencocokan string secara samar, tujuan yang ingin dicapai adalah untuk menemukan kecocokan antara sebuah string pendek dengan string lain di dalam sebuah teks yang lebih panjang; dengan anggapan hanya ada sedikit perbedaan diantara mereka.

Algoritma ini dimanfaatkan untuk asumsi kasus ketika terdapat perilaku *typo* dalam string yang akan dicocokkan.

Penghitungan levenshtein distance dapat dilakukan secara dinamis dengan menggunakan pendekatan matrix untuk mendapatkan nilai akhir distance. Pengambilan nilai minimal pada matrix yang berdekatan dengan nilai di pojok kanan matrix merepresentasikan distance akhir seberapa banyak kata harus dimodifikasi agar sesuai dengan yang dicocokkan.

Perhitungan nilai pada kolom matrix :

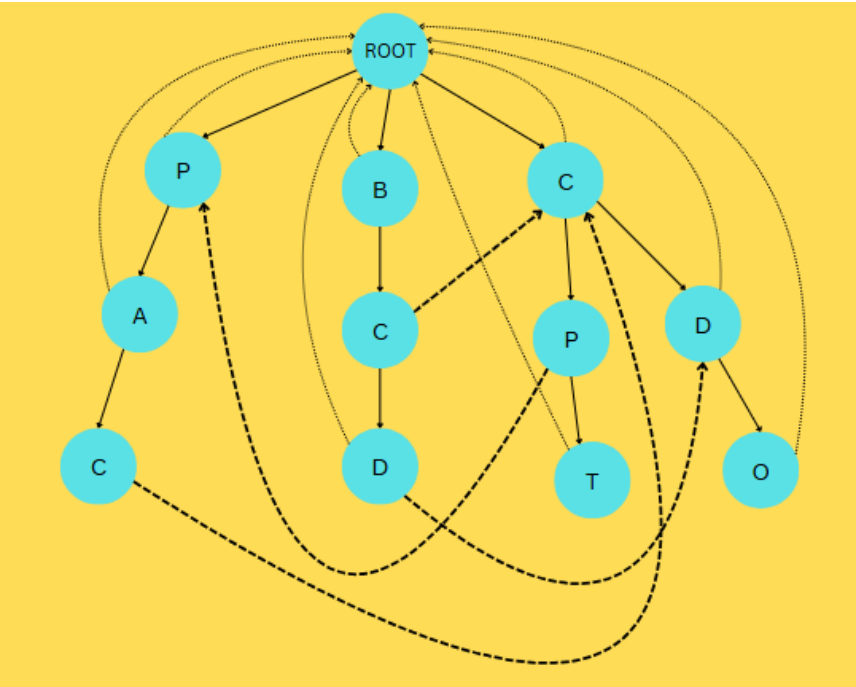
```
cost = 0 if s1[i - 1] == s2[j - 1] else 1
        # Ambil nilai minimum dari operasi (penghapusan, penyisipan, substitusi)
        dp[i][j] = min(dp[i - 1][j] + 1,        Penghapusan
                       dp[i][j - 1] + 1,        Penyisipan
                       dp[i - 1][j - 1] + cost)  Substitusi
```

| | | S | a | t | u | r | d | a | y |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **S** | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **u** | 2 | 1 | 1 | 2 | 2 | 3 | 4 | 5 | 6 |
| **n** | 3 | 2 | 2 | 2 | 3 | 3 | 4 | 5 | 6 |
| **d** | 4 | 3 | 3 | 3 | 3 | 4 | 3 | 4 | 5 |
| **a** | 5 | 4 | 3 | 4 | 4 | 4 | 4 | 3 | 4 |
| **y** | 6 | 5 | 4 | 4 | 5 | 5 | 5 | 4 | 3 |

Contoh Perhitungan Levenshtein Distance dengan Matrix

### 3.4.4. Aho Corasick

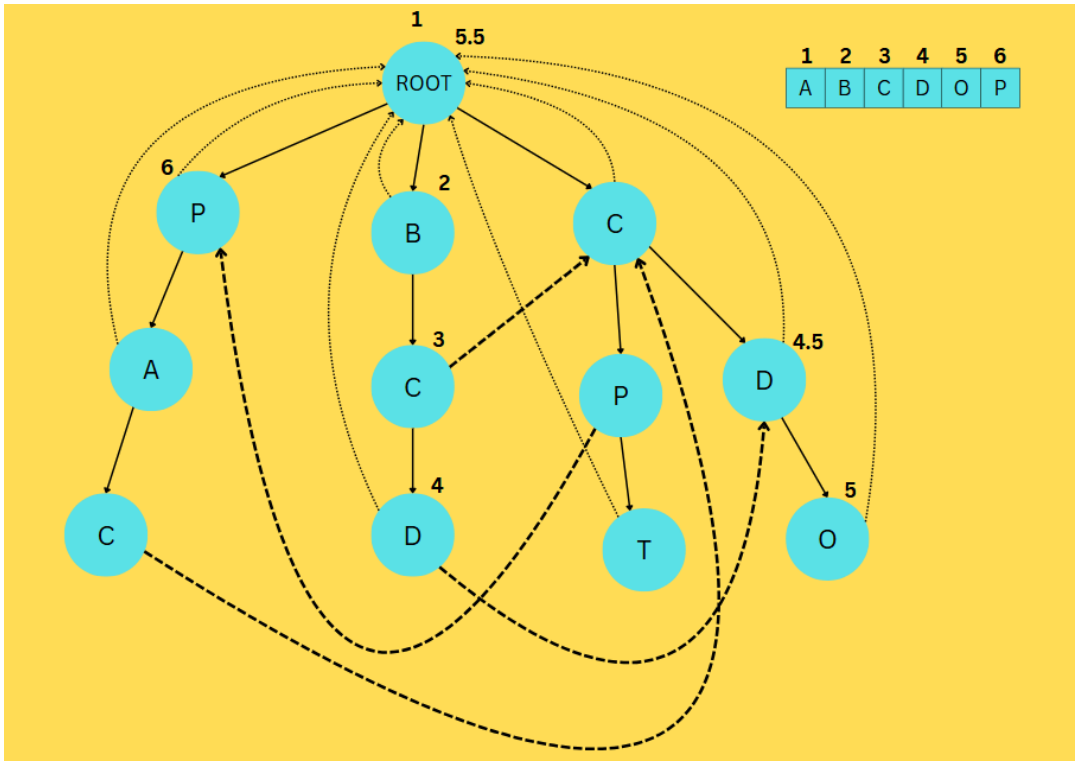
Aho Corasick merupakan salah satu algoritma pattern matching seperti KMP atau BM tetapi ia memiliki kelebihan sendiri, yaitu kemampuannya untuk mencari banyak pattern sekaligus dalam satu kali pemrosesan teks. Hal ini berbeda dengan kedua algoritma sebelumnya yang hanya dapat mencari satu pola dalam satu waktu. Untuk memahami bagaimana Aho-Corasick bekerja, kita perlu melihat struktur utama yang digunakannya, yaitu automaton yang dibentuk dari Trie (pohon prefix) ditambah dengan failure link dan output link. Berikut dibawah ini representasi Trie untuk pola seperti [“BCD”, “CPT”, “CDO”. “PAC”].




Gambar ilustrasi Trie Automaton Aho Corasick

Pada saat pencarian dilakukan, karakter demi karakter dari teks akan dibaca dan automaton akan berpindah dari satu simpul ke simpul lain berdasarkan transisi karakter . Namun, jika pada suatu titik tidak ditemukan transisi sesuai karakter teks saat ini, maka algoritma tidak langsung kembali ke awal, melainkan mengikuti failure link (yang digambarkan sebagai garis putus-putus tebal) ke simpul lain yang memiliki kemungkinan mencocokkan sisa pola.

Failure link ini pada dasarnya adalah jalur alternatif yang menunjukkan simpul berikutnya yang bisa dicoba ketika pencocokan gagal, tanpa harus mengulang pencarian dari akar Trie. Misalnya, jika saat membaca karakter 'D' dari teks "ABCDOP", automaton tidak menemukan transisi dari simpul saat ini, maka ia akan mengikuti failure link ke simpul terdekat yang mungkin memiliki karakter 'D'. Untuk memperjelas bagaimana iterasinya berikut ini kami sajikan proses pencarian pola untuk teks "ABCDOP".




Gambar ilustrasi pencarian pola pada string "ABCDOP"

Proses dimulai dari simpul ROOT, dan pointer teks berada pada karakter pertama, yaitu 'A'. Pada iterasi pertama, tidak ditemukan jalur dari ROOT ke 'A', sehingga terjadi mismatch dan pointer teks maju. Iterasi kedua memproses karakter 'B', dan ditemukan transisi dari ROOT ke simpul B. Karena simpul ini bukan akhir dari pola, pencarian dilanjutkan. Pada iterasi ketiga, karakter 'C' membawa kita dari simpul B ke simpul BC, lalu pada iterasi keempat, karakter 'D' mengarahkan ke simpul BCD. Simpul BCD merupakan akhir dari pola "BCD", sehingga algoritma mencatat bahwa pola tersebut ditemukan dan berakhir pada indeks 4.

Namun pencarian tidak berhenti di situ. Setelah pencocokan pola "BCD", algoritma tetap melanjutkan pencarian terhadap potensi tumpang tindih. Masih pada karakter 'D' (iterasi 4), karena karakter berikutnya dalam teks adalah 'O', algoritma menyadari bahwa 'O' tidak sesuai lanjutan dari pola "BCD". Maka digunakan failure link dari simpul BCD, yang membawa kita ke simpul CD, yakni simpul yang mewakili sufiks dari "BCD" yang juga merupakan prefiks dari pola lain, seperti "CDO".

Langkah ini disebut sebagai iterasi 4.5, di mana dari simpul CD, algoritma memeriksa apakah ada transisi untuk karakter 'O'. Ternyata ada, dan transisi ini membawa kita ke simpul CDO, yang merupakan akhir dari pola "CDO". Maka, algoritma mencatat bahwa pola "CDO" ditemukan, juga berakhir pada karakter 'O' di teks (indeks 5).

Selanjutnya, algoritma memproses karakter terakhir dari teks, yaitu 'P', dari simpul CDO. Karena tidak ada transisi 'P' dari simpul ini, maka algoritma mengikuti failure link dari CDO menuju simpul ROOT. Proses ini disebut sebagai iterasi 5.5. Dari ROOT, algoritma menemukan bahwa terdapat transisi untuk karakter 'P', sehingga berpindah ke simpul P pada iterasi 6. Karena simpul P bukan akhir dari pola, dan teks telah selesai diproses, maka pencarian berakhir.

## 3.5. Tuning Threshold pada Levenshtein Search

Sesuai dengan kebutuhan sistem, pencarian *fuzzy* diimplementasikan menggunakan algoritma Levenshtein Distance untuk menangani kemungkinan kesalahan pengetikan (*typo*) pada kata kunci. Metode ini hanya diaktifkan apabila pencarian *exact match* menggunakan algoritma Knuth-Morris-Pratt (KMP) atau Boyer-Moore (BM) tidak menemukan hasil, sehingga efisiensi proses tetap terjaga.

Pengujian awal dilakukan menggunakan *threshold* berbasis persentase kemiripan murni (misalnya, kemiripan di atas 80%). Namun, pendekatan ini menunjukkan kelemahan signifikan pada kata kunci yang pendek. Sebagai contoh, kesalahan satu huruf pada kata "Java" (menjadi "Jav") menghasilkan persentase kemiripan sebesar 75%, yang akan ditolak oleh ambang batas 80% meskipun kemungkinan besar relevan.

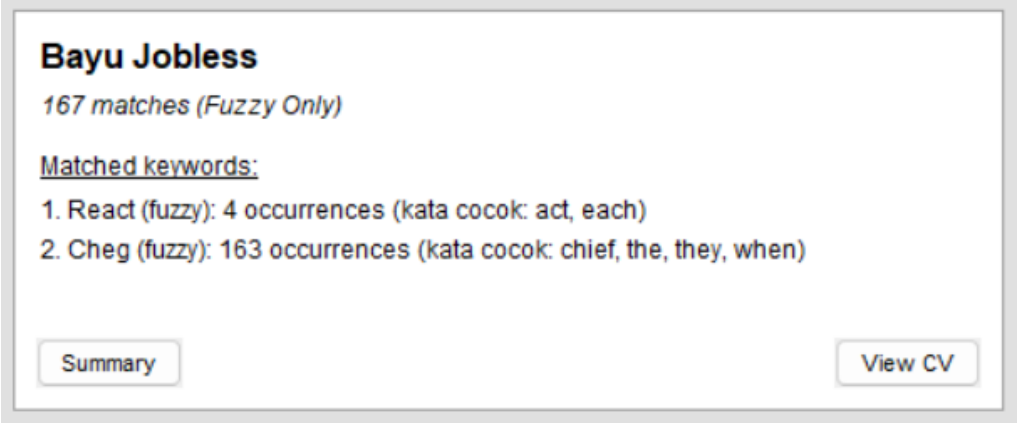
Untuk mengatasi hal tersebut, dikembangkan sebuah model threshold dinamis. Model ini tetap menggunakan persentase sebagai parameter input utama (tolerance_percent) untuk menentukan toleransi kesalahan secara proporsional, namun hasilnya dikonversi menjadi jumlah editan absolut (allowed_edits) yang diizinkan dan dibatasi oleh nilai maksimal (max_edits).

## Tuning Parameter

Pada fungsi dynamicLevenshteinSearch(text, pattern, tolerance_percent=0.15, max_edits=3), pemilihan nilai default dilakukan setelah mempertimbangkan berbagai jenis kesalahan pengetikan umum, dengan pengorbanan transposisi karakter (huruf tertukar) seperti html -> hmtl. Observasi menunjukkan bahwa konfigurasi yang terlalu bebas memberikan terlalu banyak kata yang bahkan tidak relevan.

1. tolerance_percent: float = 0.15 (15%)

   Tingkat toleransi ini dipilih untuk mempersempit cakupan pencarian fuzzy, dengan fokus untuk menangkap kesalahan pengetikan yang sangat minor dan paling umum terjadi (misalnya, satu huruf tertukar atau salah ketik). Pendekatan ini secara sadar mengurangi risiko sistem mencocokkan kata yang artinya sudah berbeda, dengan konsekuensi bahwa beberapa typo yang lebih kompleks seperti adanya karakter tertukar pada jumlah kata 4 html -> hmtl mungkin tidak akan ditemukan.




Contoh hasil pencarian ketika toleransi 25%, maka dikurangi ke 15%

- Kata 1-6 huruf: ceil(0.15 * 6) = 1. Hanya mengizinkan satu kesalahan minor.
- Kata 7-13 huruf: ceil(0.15 * 13) = 2 . Mengizinkan maksimal dua kesalahan.
- Kata >13 huruf = 3 Mengizinkan maksimal tiga kesalahan.

2. Max_edits = 3

Aturan ini didasarkan pada prinsip bahwa kesalahan yang melibatkan lebih dari tigaeditan (kombinasi penghapusan, penyisipan, atau substitusi) memiliki probabilitas yang sangat tinggi untuk menghasilkan kata dengan makna atau maksud yang sama sekali berbeda. Batasan ini krusial untuk menjaga integritas hasil pencarian pada frasa atau kata kunci yang panjang.

Dari hasil beberapa kali percobaan pencarian, yang menghasilkan hasil paling logis dan masuk akal adalah kombinasi ini. Jika dibuat batasan levenshtein distance yang lebih longgar, hasil seringkali memberikan kata dengan makna yang sangat berbeda dengan yang dimaksudkan.

# Bab 4

# Implementasi dan Pengujian

## 4.1. Spesifikasi Teknis Program

### 4.1.1. Struktur Data

Terdapat beberapa struktur data yang digunakan untuk membantu pembangunan aplikasi ini. Struktur data ini ada baik di bagian GUI maupun logika pencarian.

Di bagian GUI, aplikasi ini menggunakan kelas CVAnalyzerApp dan kelas SummaryWindow. CVAnalyzerApp adalah kelas yang merepresentasikan window utama dari aplikasi ini. Sementar itu, SummaryWindow adalah kelas yang merepresentasikan *popup window* yang muncul ketika tombol "view summary" ditekan.

Pencarian Aho-Corasick memanfaatkan dua kelas untuk membantu pencarian. Kelas pertama adalah node yang merepresentasikan sebuah state atau posisi dalam mesin pencarian. Kelas ini memiliki tiga atribut, yaitu children, output, dan failure_link. Atribut children berfungsi untuk merepresentasikan transisi dari satu state ke state lain. Atribut failure_link berfungsi untuk berpindah ke state lain saat terjadi *mismatch.* Atribut output berfungsi untuk menandai akhir dari sebuah kata kunci yang valid. Kelas kedua adalah AhoCorasick yang merepresentasikan struktur keseluruhan. Struktur queue juga digunakan dalam pembuatan failure link.

### 4.1.2. Fungsi dan Prosedur

Tabel 4.1.2.1 Fungsi dan Prosedur Program

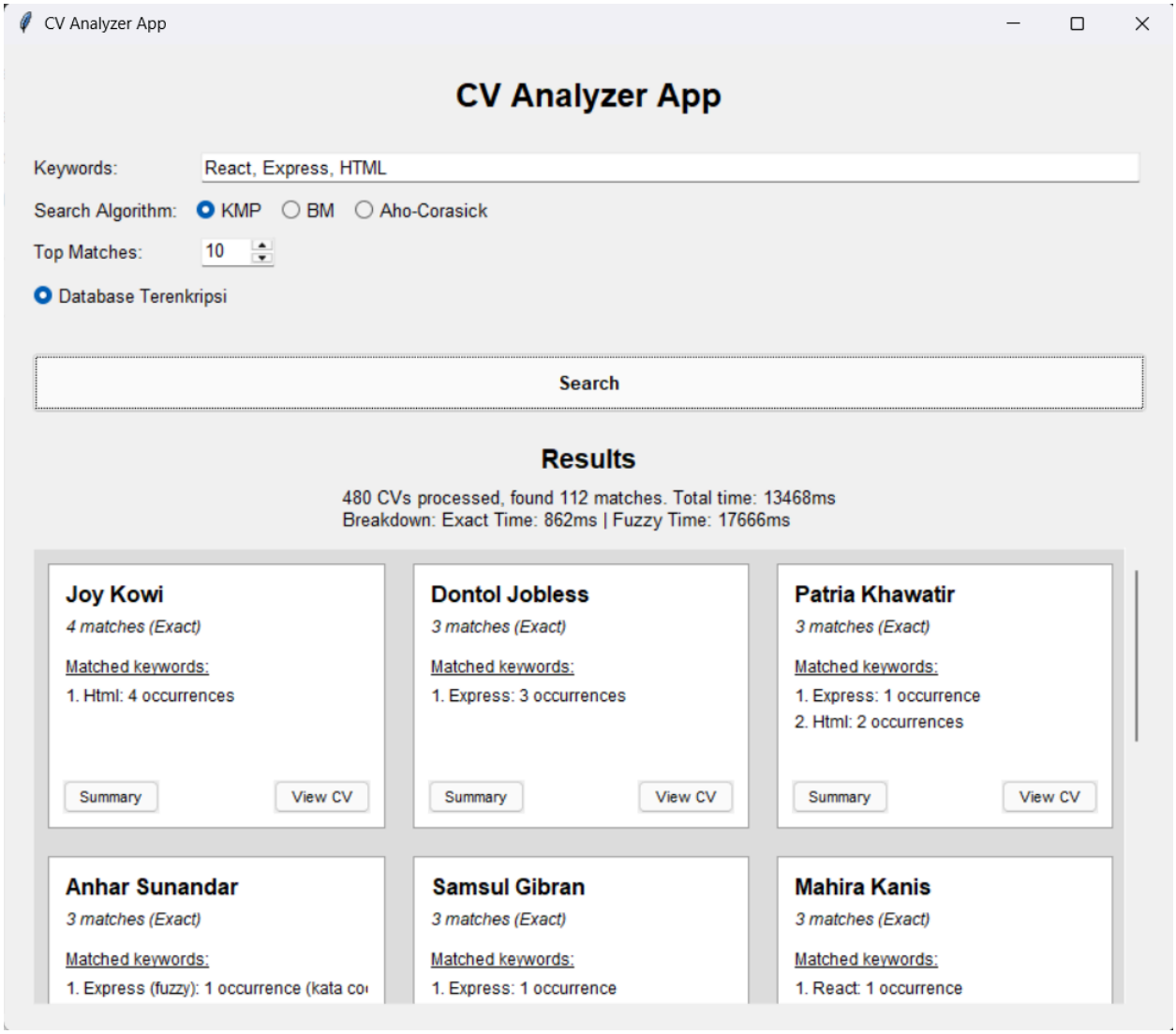
| Fungsi / Prosedur | Deskripsi |
|---|---|
| extract_text_from_pdf() | Fungsi untuk membaca isi file pdf |
| create_flat_text() | Fungsi untuk mengubah isi file pdf menjadi sebuah flat text panjang |
| find_section_starts(text) | Fungsi untuk mencari awal dari section-section yang ada di file pdf |
| split_into_sections(text) | Fungsi untuk membagi file pdf berdasarkan section-section yang ada |
| is_potential_entry_start(line_text, next_line_text, year_pattern) | Fungsi yang memutuskan apakah sebuah line merupakan awal suatu entry |
| parse_experience_section() | Fungsi untuk mengambil isi section Experience |

| | |
|---|---|
| parse_educations_section() | Fungsi untuk mengambil isi section Education |
| parse_skills_section() | Fungsi untuk mengambil isi section Skill |
| xor_encrypt_data(str, str) | Fungsi untuk mengenkripsi data |
| xor_decrypt_data(str, str) | Fungsi untuk mendekripsi data |
| get_db_connection() | Fungsi untuk menyambungkan program dengan database |
| get_profile_by_id(id, decrypted) | Fungsi untuk mengambil data kandidat berdasarkan idnya |
| hitungLPS(pola) | Fungsi untuk menghitung prefiks dan sufiks terbesar dari pola |
| KMP(teks,pola) | Fungsi untuk melakukan pencarian KMP |
| buildtrie(self) | Fungsi untuk membuat trie dari daftar kata kunci |
| buildfailurelink(self) | Fungsi untuk membuat failure link |
| search(self,text) | Fungsi untuk melakukan pencarian menggunakan Aho-Corasick |
| generatePreProcessTable(pattern) | Fungsi untuk membuat tabel *last occurrence* dari tiap karakter |
| BM(text, pattern) | Fungsi untuk melakukan pencarian dengan algoritma Boyer-Moore |
| // tambahin fungsi fuzzy di sini | |
| process_cv_worker(data, keyword, algo, automaton) | Fungsi untuk memproses satu file CV |
| perform_search(self) | Fungsi untuk handle ketika tombol search ditekan, melakukan pencarian sesuai input |
| display_results(self,data) | Fungsi untuk menampilkan hasil |
| view_summary(self,data) | Fungsi untuk menampilkan window ringkasan CV dan biodata |
| view_cv(self,data) | Fungsi untuk membuka file asli CV |

## 4.2. Tata Cara Penggunaan Program

### 4.2.1. Interface

Berikut adalah interface dari aplikasi kami.




### 4.2.2. Cara Menjalankan Program

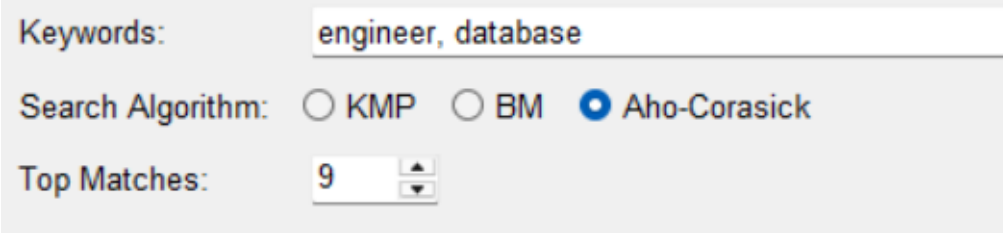
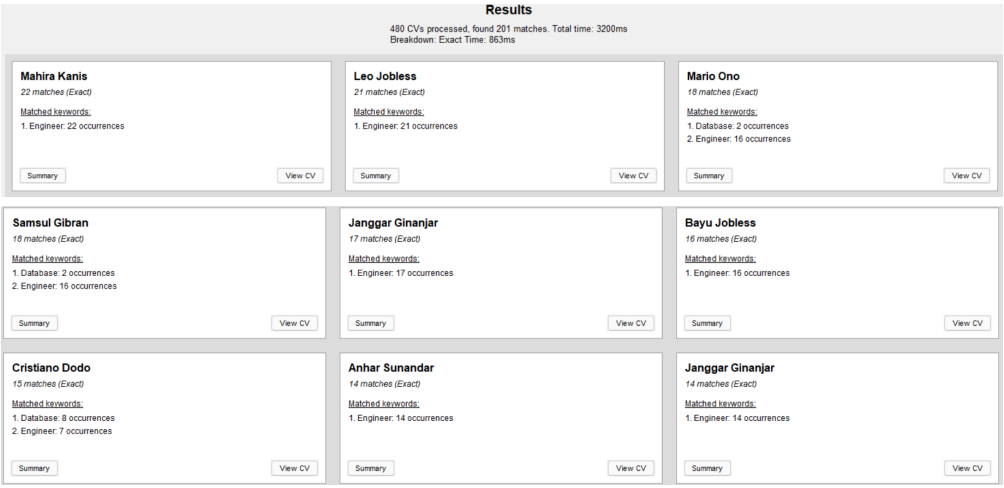
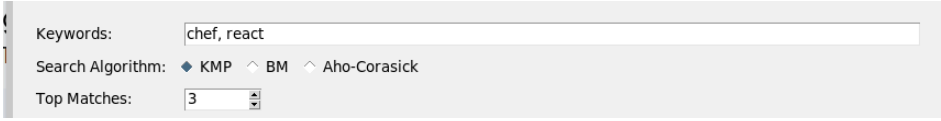
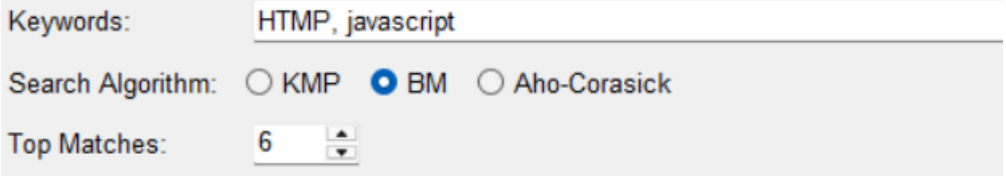
- Siapkan file-file CV dan database
- Pindah ke direktori src
- Jalankan command “py gui.py”

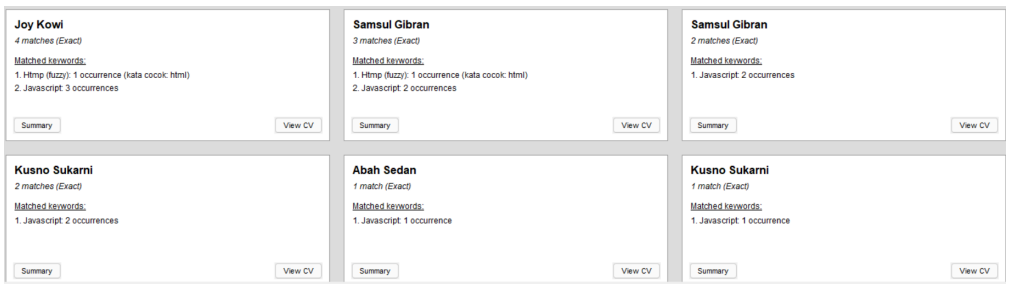
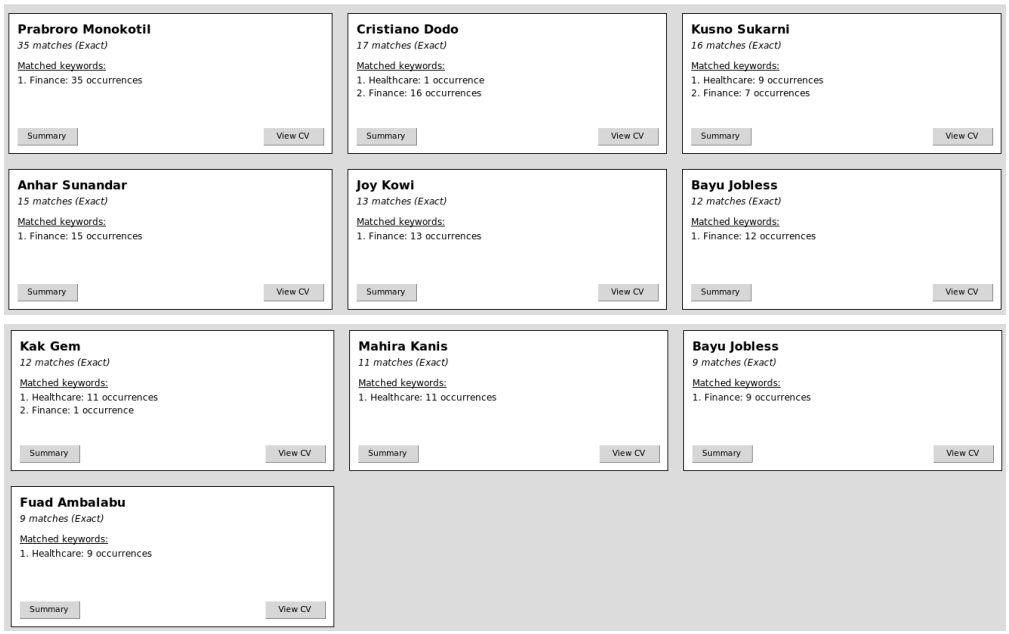
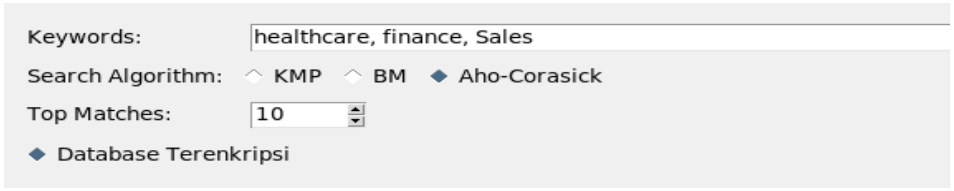
### 4.2.3. Cara Penggunaan Program

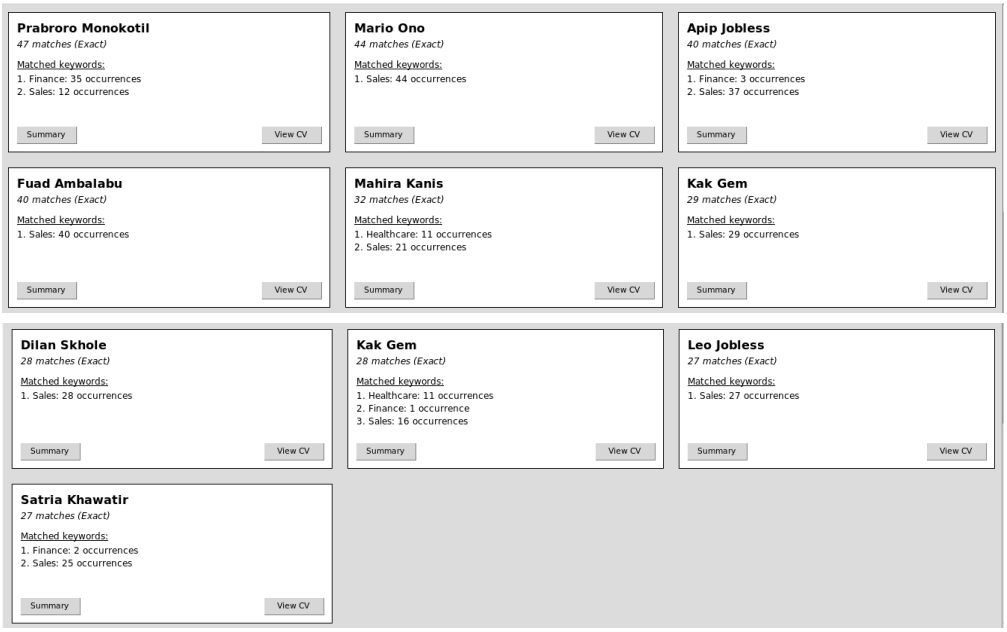
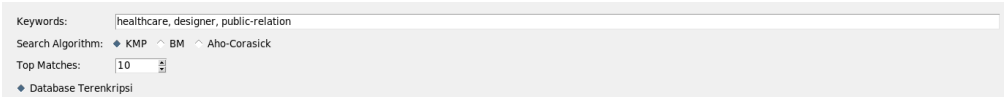
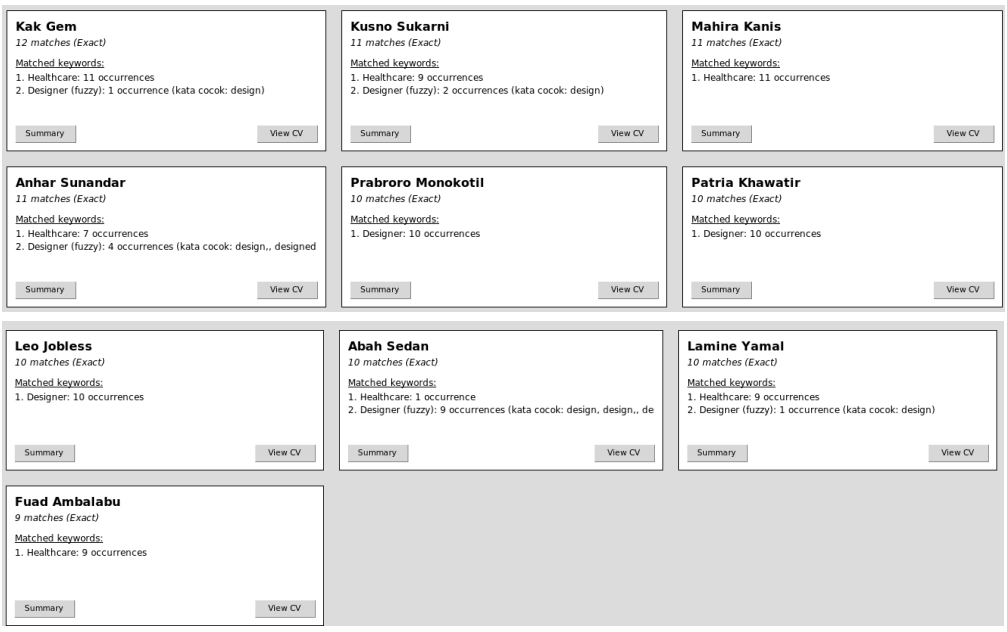
- Masukkan kata kunci yang diinginkan, jika lebih dari satu pisahkan dengan koma
- Masukkan jumlah hasil yang ingin ditampilkan
- Pilih algoritma yang ingin digunakan
- Tekan tombol “search”

- Jika ingin melihat ringkasan CV dan biodata kandidat tekan tombol “view summary”
- Jika ingin melihat file asli CV tekan tombol “view CV”

## 4.3. Hasil Pengujian

<table>
<tr><th>No</th><th>Label</th><th>Isi</th></tr>
<tr><td rowspan="2">1</td><td>Query dan algoritma</td><td> 
</td></tr>
<tr><td>Hasil Pencarian</td><td> 
</td></tr>
<tr><td rowspan="2">2</td><td>Query dan algoritma</td><td> 
</td></tr>
<tr><td>Hasil Pencarian</td><td> 
</td></tr>
<tr><td>3.</td><td>Query dan algoritma</td><td> 
</td></tr>
</table>

| | | |
|---|---|---|
| | Hasil Pencarian |  |
| 4. | Query dan algoritma |  |
| | Hasil Pencarian |  |
| 5 | Query dan algoritma |  |

| | | |
|---|---|---|
| | Hasil Pencarian |  |
| 6. | Query dan Algoritma |  |
| | Hasil Pencarian |  |

## 4.4. Analisis Hasil Pengujian

### 4.4.1 Hasil Pengujian Fungsionalitas

| No | Skenario | Kata Kunci | Teks pada CV | Hasil diharapkan | Hasil Aktual | Status |
|---|---|---|---|---|---|---|
| **1** | Exact Match | Engineer, Database | Engineer, Database | Ditemukan (exact) | Ditemukan (exact) | Berhasil |
| **2** | Multi, Exact Match | chef, react | chef,react | Ditemukan (exact) | Ditemukan (fuzzy-exact) | Gagal |
| **3** | Multi, typo minor + exact | HTMP, Javascript | HTML, Javascript | Ditemukan (fuzzy - exact) | Ditemukan (fuzzy - exact) | Berhasil |
| **4** | Multi, exact match | Healthcare, Finance | Healthcare, Finance | Ditemukan (exact) | Ditemukan (exact) | Berhasil |
| **5** | Multi, exact match | Healthcare, Finance, Sales | Healthcare, Finance, Sales | Ditemukan (exact) | Ditemukan (exact) | Berhasil |
| **6** | Multi, exact-fuzzy match | Healthcare, designer, public-relation | Healthcare, designer, public-relation,... | Ditemukan (exact,fuzzy) | Ditemukan (exact,fuzzy) | Berhasil |

### 4.4.2 Hasil Kinerja

| Jumlah CV | Algoritma | Jumlah Kata Kunci | Total Waktu Proses | Total CPU Work - Exact | Total CPU Work - Fuzzy |
|---|---|---|---|---|---|
| 480 | Aho-Corasick | 2 | 3800 ms | 465 ms | - |
| | KMP | 2 | 23474 ms | 1347 ms | 16170 ms |
| | BM | 2 | 11535 ms | 10 ms | 1699 ms |
| | BM | 2 | 14941 ms | 1333 ms | 21797 ms |
| | Aho-Corasick | 3 | 3274 ms | 434 ms | - |
| | KMP | 3 | 40517 ms | 5435 ms | 150249 ms |

Berdasarkan hasil pengujian fungsionalitas dan kinerja ketiga algoritma, dapat disimpulkan bahwa sistem ini secara umum mampu mendeteksi pola atau kata kunci yang ada pada kumpulan CV. Sebagian besar skenario pengujian fungsionalitas menunjukkan keberhasilan, baik untuk pencocokan exact match maupun fuzzy match (dengan toleransi kesalahan seperti typo minor). Fitur fuzzy match ini krusial karena seringkali pola pada CV memiliki variasi penulisan.

Pada percobaan kali ini, dapat dilihat bahwa semakin banyak kata kunci yang dipakai, performa algoritma tertentu akan sangat berbeda. Notasi Big O digunakan untuk menggambarkan bagaimana waktu eksekusi algoritma bertumbuh seiring dengan peningkatan ukuran input.

Pada percobaan yang menggunakan algoritma Aho-Corasick, didapatkan waktu eksekusi yang cukup cepat (Tabel 4.4.2) . Hal ini sesuai dengan teori dimana algoritma ini memang memiliki kelebihan yaitu dapat melakukan pencocokan multi pola dengan sebuah string secara efisien. Jika ditinjau dari kompleksitas waktunya, performa dari algoritma ini memang sesuai dengan harapan dimana untuk notasi Big O nya sendiri adalah  O(m+N+K), dimana m adalah panjang teks CV, N adalah total panjang kata kunci yang ingin dicari dan K adalah jumlah kecocokan yang ditemukan.

Sementara disisi lain algoritma String matching seperti BM dan KMP memiliki total waktu yang naik secara signifikan disaat kita mulai mencari multiple kata kunci. Jika dirujuk dari tabel 4.4.2 diantara kedua algoritma tersebut yang memiliki waktu eksekusi lebih buruk adalah algoritma KMP. Hal ini bisa saja terjadi karena terdapat banyak string yang memiliki prefix/suffix yang sama dengan yang ada pada tabel longest prefix-suffix nya yang menyebabkan ia tidak dapat melakukan lompatan yang jauh. Secara garis besar kedua algoritma ini memiliki kompleksitas rata-rata O(m+n) dan pada kasus terburuk akan mencapai O(mn).

Pada Tabel 4.4.2, kami juga menyadari bahwa exact match tidak selalu dapat diandalkan, terutama jika CV partisipan mengandung banyak typo. Banyaknya typo tersebut mengakibatkan perlunya fuzzy matching, yang jika diterapkan secara tidak efisien, dapat menciptakan overhead komputasi yang besar saat mencari pola yang relevan. Pada kasus ini, proses fuzzy matching menjadi bottleneck yang cukup signifikan, terutama jika hasil dari exact match tidak dapat diandalkan dan sistem terpaksa mengandalkan fuzzy matching untuk menemukan kecocokan. Hal ini adalah salah satu faktor utama mengapa waktu yang dibutuhkan untuk mencari dua kata kunci atau lebih menjadi sangat signifikan, karena beban tambahan dari fuzzy matching tersebut.

# Bab 5
# Kesimpulan

## 5.1. Kesimpulan

Pada pengerjaan tugas besar ini, telah berhasil dibangun aplikasi desktop yang mampu menampilkan CV kandidat yang memiliki kecocokan terbesar dengan kata kunci yang dimasukkan pengguna. Pencarian kecocokan *exact matching* menggunakan algoritma KMP, BM, dan Aho-Corasick, sementara untuk *fuzzy matching* memanfaatkan Levenshtein Distance. Algoritma yang dibuat bisa dibilang efektif karena bisa secara tepat menghitung kecocokan CV dalam pengujian yang sudah dilakukan.

## 5.2. Saran

Saran untuk pengembangan aplikasi ini adalah peningkatan kinerja threading program. Saat ini aplikasi memang berhasil mencari kecocokan, tetapi waktu yang diperlukan cukup lama. Selain itu, aplikasi mungkin juga bisa menambahkan algoritma lain yang lebih *advance* untuk pengukuran kecocokan.

Antarmuka aplikasi ini juga bisa ditingkatkan. Antarmuka sekarang masih sederhana. Antarmuka bisa dibuat lebih kaya dan interaktif. Selain itu, juga bisa dikembangkan agar bisa mendukung multiplatform.

## 5.3. Refleksi

Pengerjaan tugas besar ini memberikan pengalaman baru terhadap teknik *pattern matching*. Tugas besar ini juga menambah pengalaman dalam pengembangan aplikasi desktop. Selain hard skill, pengerjaan tugas besar ini juga memberikan pelajaran tentang pembagian tugas dan kerja sama.

Pengerjaan tugas besar ini juga memberikan pelajaran agar tidak menunda-nunda pekerjaan. Pekerjaan yang mungkin terlihat mudah akan tetap sulit jika kita kehabisan waktu. Selain itu, banyaknya tugas lain juga membuat kita bisa meningkatkan kemampuan manajemen waktu dalam pengerjaan tugas besar ini.

# Lampiran

## Tautan Repository Github

orvin14/Tubes3_MeACavemanDoesntHaveReligion

## Tautan Video

https://youtu.be/CL-XUOgnb2E?si=nIzmNdrtneaqVgMF

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

# Daftar Pustaka


GeeksforGeeks. (2024, Mar11). *Aho-Corasick Algorithm for Pattern Searching*. GeeksforGeeks. Diakses pada 8 Juni 2025, dari https://www.geeksforgeeks.org/dsa/aho-corasick-algorithm-pattern-searching/

GeeksforGeeks. (2025, Apr11). *Boyer Moore Algorithm for Pattern Searching*. GeeksforGeeks. Diakses pada 7 Juni 2025, dari https://www.geeksforgeeks.org/dsa/boyer-moore-algorithm-for-pattern-searching/

GeeksforGeeks. (2024, Jan31). *Introduction to Levenshtein Distance*. GeeksforGeeks. Diakses pada 15 Juni 2025, dari https://www.geeksforgeeks.org/dsa/introduction-to-levenshtein-distance/

GeeksforGeeks. (2025, Feb 25). *KMP Algorithm for Pattern Searching*. GeeksforGeeks. Diakses pada 7 Juni 2025, dari https://www.geeksforgeeks.org/dsa/kmp-algorithm-for-pattern-searching/

Munir, R. (2021). *Pencocokan string* [Materi Kuliah]. Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. Diakses dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf pada 7 Juni 2025.