

# Laporan Tugas Besar III Strategi Algoritma

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

### VitaeLangX



Adinda Putri 13523071 <a href="mailto:13523071@std.stei.itb.ac.id">13523071@std.stei.itb.ac.id</a>	Muhammad Fathur Rizky 13523105 <a href="mailto:13523105@std.stei.itb.ac.id">13523105@std.stei.itb.ac.id</a>	Ahmad Wicaksono 13523121 <a href="mailto:13523121@std.stei.itb.ac.id">13523121@std.stei.itb.ac.id</a>
--	---	---

## Daftar Isi

<b>BAB I: Deskripsi Tugas</b>	<b>2</b>
<b>BAB II: Landasan Teori</b>	<b>3</b>
Knuth-Morris-Pratt (KMP)	3
Boyer-Moore (BM)	3
Aho-Corasick	3
<b>BAB III: Analisis Pemecahan Masalah</b>	<b>4</b>
Konversi CV ke Representasi Teks	4
Pencocokan Kata Kunci secara Eksak	4
Pencarian Toleran-Typo (Fuzzy Matching)	4
Penilaian dan Pengurutan Hasil	4
Ekstraksi Informasi dan Basis Data	4
Antarmuka Pengguna Interaktif	5
<b>BAB IV: Implementasi dan Pengujian</b>	<b>6</b>
Folder Structure	7
Seeding Algorithm	8
Encryption Algorithm	10
Base Search Algorithm	11
Exact Matching Algorithm	12
Knuth-Morris-Pratt Algorithm	12
Boyer-Moore Algorithm	14
String Similarity Algorithm	16
Levenshtein	16
Multi Pattern Algorithm	17
Aho-Corasick Algorithm	18
Analisis Kompleksitas dan Benchmarking	22
Pengujian Test Cases	23
<b>BAB V: Kesimpulan dan Saran</b>	<b>26</b>
Kesimpulan	26
Saran	26
<b>Lampiran</b>	<b>27</b>
<b>Daftar Pustaka</b>	<b>27</b>

## BAB I: Deskripsi Tugas

Pada era digital saat ini, proses rekrutmen tenaga kerja telah mengalami transformasi signifikan dengan memanfaatkan teknologi informasi, salah satunya melalui sistem Applicant Tracking System (ATS). Sistem ini dirancang untuk membantu perusahaan dalam menyaring dan mencocokkan informasi dari dokumen lamaran kerja, khususnya Curriculum Vitae (CV), secara otomatis dan efisien. Dengan kemampuannya mengelola ribuan dokumen, ATS memungkinkan pencarian kandidat yang relevan dilakukan dengan cepat sehingga mendukung efisiensi dalam proses perekrutan.

Namun, salah satu tantangan utama dalam pengembangan sistem ATS adalah bagaimana mengekstrak informasi penting dari dokumen CV yang umumnya berbentuk PDF dan tidak memiliki struktur tetap. Untuk menjawab tantangan ini, digunakan teknik pattern matching atau pencocokan pola teks. Dalam konteks tugas ini, algoritma Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM) digunakan sebagai metode pencarian pola utama. Kedua algoritma ini dikenal memiliki efisiensi tinggi dalam pencarian teks dan sangat sesuai untuk pengolahan dokumen berukuran besar seperti CV.

Tugas besar ini menugaskan mahasiswa untuk membangun sebuah sistem ATS berbasis CV digital yang dapat melakukan deteksi informasi pelamar secara otomatis. Sistem ini akan memanfaatkan algoritma KMP dan BM untuk melakukan pencocokan kata kunci dari pengguna terhadap isi CV yang telah dikonversi menjadi teks. Sistem juga dilengkapi dengan algoritma Levenshtein Distance untuk menangani pencarian fuzzy apabila terjadi ketidaksesuaian atau kesalahan ketik pada kata kunci yang diberikan. Selain itu, informasi penting seperti nama, kontak, pengalaman kerja, keahlian, dan pendidikan pelamar akan diekstraksi secara otomatis menggunakan teknik regular expression.

Secara keseluruhan, sistem ini tidak hanya melakukan pencarian dan pencocokan kata kunci, tetapi juga membentuk profil kandidat secara otomatis dengan mengintegrasikan data dari CV ke dalam basis data. Tujuan akhirnya adalah menciptakan sebuah sistem yang mampu menampilkan ringkasan informasi pelamar yang relevan, membantu pengguna dalam proses seleksi, dan memberikan pengalaman pencarian yang cepat, akurat, dan informatif.

## BAB II: Landasan Teori

### **Knuth-Morris-Pratt (KMP)**

Algoritma Knuth-Morris-Pratt (KMP) merupakan salah satu algoritma pencocokan string yang efisien yang dikembangkan oleh Donald Knuth, Vaughan Pratt, dan James H. Morris. KMP dirancang untuk mencari keberadaan sebuah pola dalam teks tanpa harus memeriksa karakter yang telah diketahui sebelumnya. Prinsip utama dari algoritma ini adalah penggunaan tabel prefix function (atau biasa disebut tabel "lps", longest proper prefix which is also suffix) untuk menghindari pencocokan ulang karakter yang sama. Selama proses pencarian, ketika ditemukan ketidaksesuaian karakter antara pola dan teks, algoritma KMP akan merujuk ke tabel tersebut untuk menentukan posisi pencocokan berikutnya, tanpa harus kembali ke awal pola. Dengan demikian, KMP mampu melakukan pencarian dengan kompleksitas waktu  $O(n + m)$ , di mana  $n$  adalah panjang teks dan  $m$  adalah panjang pola. Keunggulan ini menjadikan KMP sangat cocok digunakan dalam proses pencarian pola secara berulang dan masif, seperti yang diperlukan dalam sistem ATS untuk memindai teks CV.

### **Boyer-Moore (BM)**

Algoritma Boyer-Moore merupakan salah satu algoritma pencocokan string tercepat dalam praktik, terutama untuk teks berukuran besar. Diperkenalkan oleh Robert S. Boyer dan J Strother Moore, algoritma ini memanfaatkan dua heuristik utama: bad character rule dan good suffix rule. Berbeda dengan KMP yang mencocokkan karakter dari kiri ke kanan, Boyer-Moore melakukan pencocokan dari kanan ke kiri. Jika terjadi ketidaksesuaian karakter, BM akan menentukan seberapa jauh pola dapat digeser berdasarkan informasi dari kedua heuristik tersebut sehingga dalam banyak kasus pergeseran dapat dilakukan lebih jauh dibanding algoritma lainnya. Meskipun kompleksitas terburuknya adalah  $O(nm)$ , performa praktis Boyer-Moore sangat efisien, terutama saat pola yang dicari relatif panjang dan karakter yang digunakan cukup beragam. Hal ini menjadikannya pilihan ideal untuk digunakan dalam proses pencarian keyword dari dokumen CV digital pada sistem ATS.

### **Aho-Corasick**

Aho-Corasick adalah algoritma pencocokan string yang dirancang untuk menangani multi-pattern matching secara efisien. Dikembangkan oleh Alfred V. Aho dan Margaret J. Corasick, algoritma ini membentuk sebuah finite automaton (mesin hingga) dari sekumpulan pola yang ingin dicocokkan, biasanya dalam bentuk struktur trie yang kemudian diperkaya dengan fungsi fail-link dan output-link. Proses pencarian dilakukan dalam satu kali traversal teks, di mana setiap karakter dalam teks hanya dibaca sekali. Keunggulan utama dari Aho-Corasick terletak pada kemampuannya untuk menemukan semua kemunculan dari berbagai pola secara bersamaan dalam waktu linear terhadap panjang teks ditambah total panjang semua pola. Dalam konteks sistem ATS, Aho-Corasick sangat bermanfaat ketika jumlah kata kunci yang dicari cukup banyak, karena dapat menghemat waktu pencarian secara signifikan dibanding pendekatan satu per satu seperti pada KMP maupun BM. Algoritma ini menjadi solusi optimal untuk pencocokan kata kunci masif dalam skenario rekrutmen berskala besar.

## **BAB III: Analisis Pemecahan Masalah**

Permasalahan utama yang diangkat dalam tugas besar ini adalah bagaimana membangun sebuah sistem yang mampu mengekstrak serta mencocokkan informasi penting dari dokumen CV digital secara otomatis, cepat, dan akurat. Tantangan ini mencakup penanganan format CV yang tidak terstruktur serta kebutuhan pencarian kata kunci dalam skala besar yang harus tetap efisien meskipun terdapat variasi penulisan atau kesalahan ketik. Untuk mengatasi hal tersebut, pendekatan sistematis dirancang melalui langkah-langkah berikut.

### **Konversi CV ke Representasi Teks**

Langkah pertama adalah mengubah setiap dokumen CV berformat PDF menjadi satu string teks panjang. Proses ini penting untuk memudahkan analisis menggunakan algoritma pencocokan string. Semua teks yang berhasil diekstrak akan disimpan dalam bentuk linear untuk memudahkan pencarian pola.

### **Pencocokan Kata Kunci secara Eksak**

Setelah teks diperoleh, sistem melakukan pencocokan kata kunci menggunakan algoritma Knuth-Morris-Pratt (KMP) atau Boyer-Moore (BM).

- KMP bekerja dengan efisien dalam pencocokan berulang berkat tabel lompatan (prefix table).
- BM memanfaatkan heuristik untuk melakukan lompatan besar saat pencocokan, sangat efektif pada teks panjang.

Pengguna dapat memilih salah satu algoritma sesuai kebutuhan, dan sistem akan mengidentifikasi keyword yang muncul dalam setiap CV.

### **Pencarian Toleran-Typo (Fuzzy Matching)**

Jika tidak ditemukan kecocokan secara persis, sistem akan melakukan fuzzy matching menggunakan algoritma Levenshtein Distance. Algoritma ini menghitung jarak edit antara keyword dan kata dalam CV sehingga tetap dapat menangkap makna meskipun terjadi kesalahan ketik atau variasi penulisan. Hanya kata kunci yang gagal ditemukan secara eksak yang akan dicocokkan ulang secara fuzzy.

### **Penilaian dan Pengurutan Hasil**

Setiap CV akan diberi skor berdasarkan jumlah kecocokan kata kunci, baik dari exact match maupun fuzzy match. CV-CV tersebut kemudian diurutkan dan ditampilkan sesuai skor tertinggi. Sistem juga mencatat waktu eksekusi dari masing-masing metode pencocokan untuk memberi gambaran performa.

### **Ekstraksi Informasi dan Basis Data**

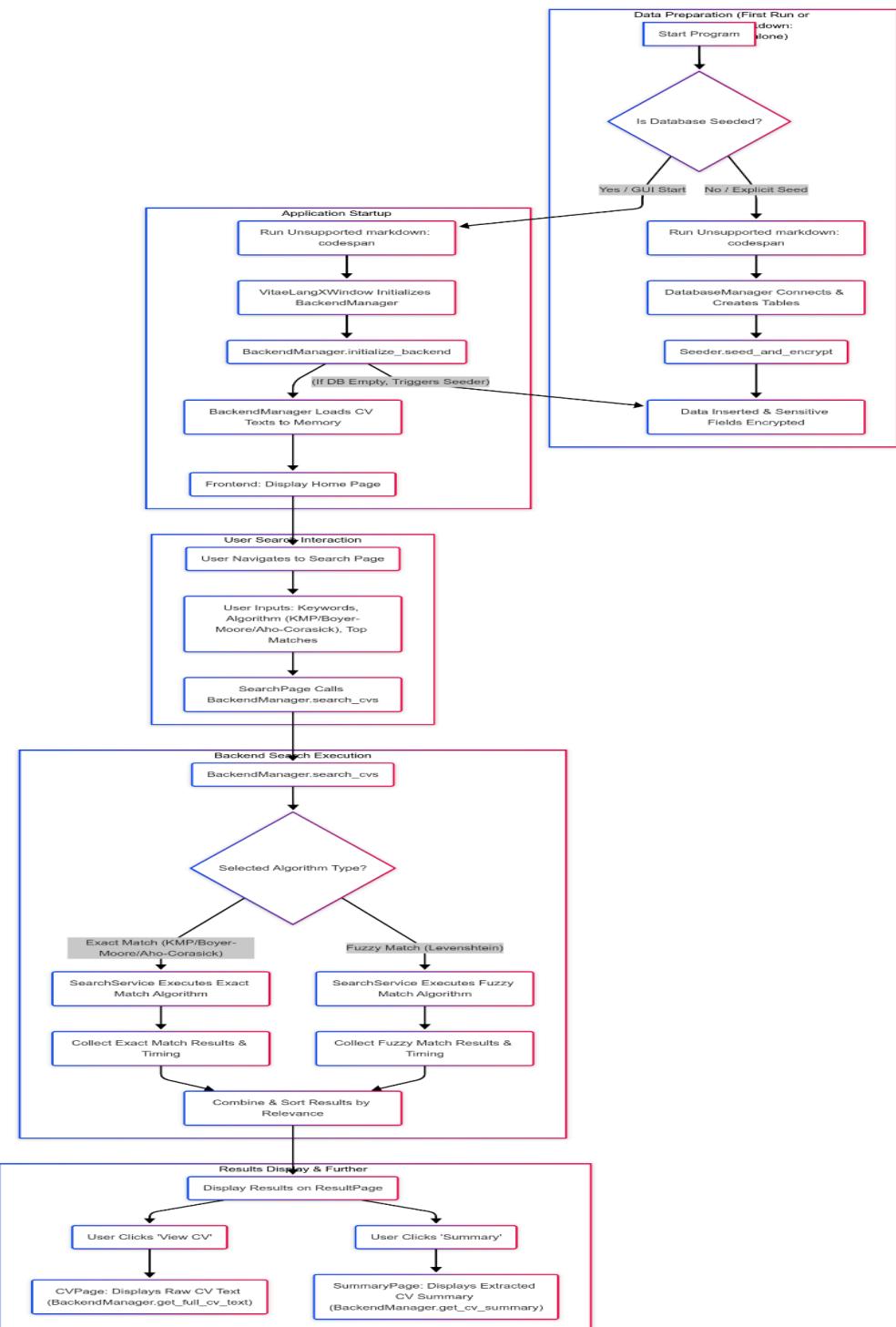
Informasi penting seperti nama, kontak, keahlian, pengalaman kerja, dan riwayat pendidikan akan diekstraksi dari teks menggunakan regular expression. Data ini disimpan dalam basis data MySQL yang terintegrasi dengan sistem sehingga pengguna dapat mengakses profil pelamar secara ringkas dan terstruktur hanya dari CV yang diunggah.

## **Antarmuka Pengguna Interaktif**

Seluruh proses diintegrasikan dalam sebuah antarmuka desktop interaktif. Pengguna dapat memasukkan kata kunci, memilih algoritma, menentukan jumlah hasil yang ingin ditampilkan, dan meninjau ringkasan hasil pencarian serta melihat dokumen CV secara langsung. Desain antarmuka disusun agar ramah pengguna dan mendukung pengalaman penggunaan yang efisien.

Melalui kombinasi teknik pattern matching, fuzzy match search, dan ekstraksi informasi otomatis, sistem yang dibangun mampu menjawab kebutuhan akan proses seleksi pelamar kerja secara digital dengan tingkat efisiensi dan akurasi yang tinggi. Pendekatan ini juga fleksibel untuk dikembangkan lebih lanjut, seperti penambahan algoritma multi-pattern seperti Aho-Corasick atau fitur keamanan tambahan seperti enkripsi data.

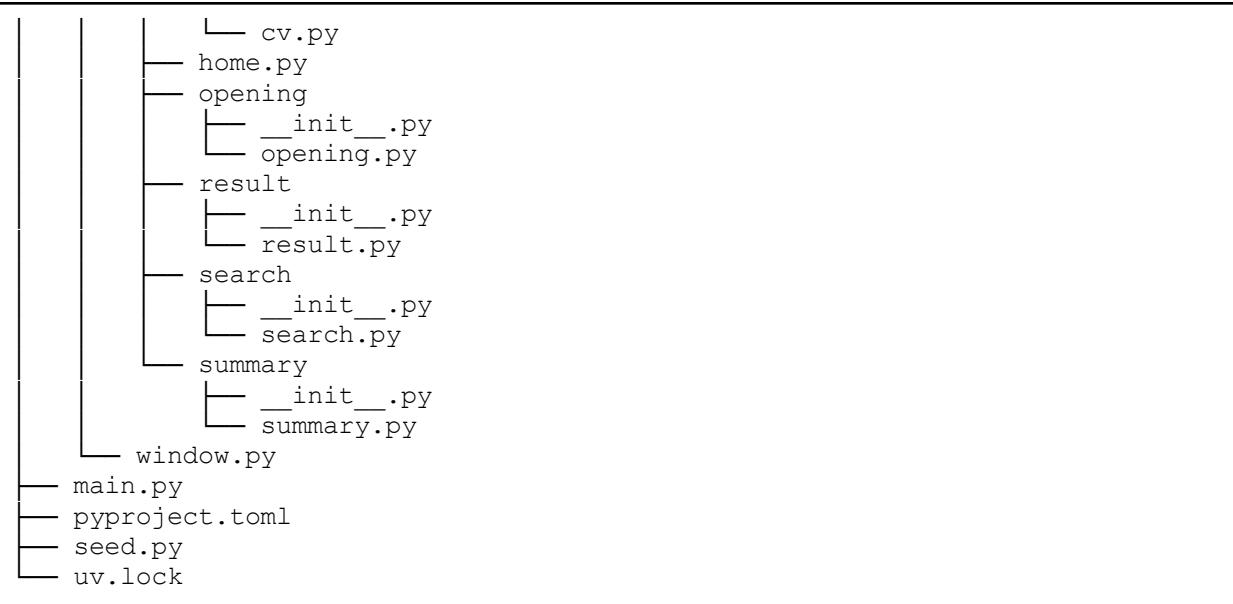
## BAB IV: Implementasi dan Pengujian



Gambar 1. Diagram Alir Program

## Folder Structure

```
src
└── backend
    ├── __init__.py
    └── algorithms
        ├── __init__.py
        ├── base_search_algorithm.py
        ├── exact_match
        │   ├── __init__.py
        │   ├── boyer_moore.py
        │   ├── exact_string_matching_algorithm.py
        │   └── kmp.py
        ├── fuzzy_match
        │   ├── __init__.py
        │   ├── levenshtein.py
        │   └── string_similarity_algorithm.py
        └── multi_pattern
            ├── __init__.py
            └── aho_corasick.py
            └── multi_pattern_string_matching.py
    └── backend_manager.py
    └── common
        ├── __init__.py
        └── settings.py
    └── db
        ├── __init__.py
        └── database_manager.py
    └── encryption
        ├── __init__.py
        └── vigenere_cipher.py
    └── models
        ├── __init__.py
        └── models.py
    └── preprocessor
        ├── __init__.py
        ├── cv_processor.py
        └── regex_extractor.py
    └── seeder.py
    └── services
        ├── __init__.py
        └── search_service.py
    └── utils
        └── utils.py
    └── frontend
        ├── __init__.py
        └── components
            ├── __init__.py
            └── sidebar.py
        └── page
            ├── __init__.py
            ├── about.py
            └── creator.py
        └── cv
            └── __init__.py
```



## Seeding Algorithm

```

class Seeder:
    def __init__(self, db_manager, encryption_cipher):
        self.db_manager = db_manager
        self.encryption = encryption_cipher

    def seed_and_encrypt(self, sql_file: str):
        try:
            self._execute_sql_file(sql_file)
            self._encrypt_existing_data()
            self.db_manager.connection.commit()
        except Exception as e:
            if self.db_manager.connection:
                self.db_manager.connection.rollback()
            raise

    def _execute_sql_file(self, sql_file: str):
        with open(sql_file, "r", encoding="utf-8") as file:
            sql_content = file.read()

            sql_statements = [
                stmt.strip() for stmt in sql_content.split(";") if stmt.strip()
            ]

            for statement in sql_statements:
                try:
                    self.db_manager._execute_query(statement, commit=True)
                except Exception:
                    continue

    def _encrypt_existing_data(self):
        query = "SELECT * FROM ApplicantProfile"
        records = self.db_manager._execute_query(query, fetch_all=True)

```

```

    if not records:
        return

    for record_dict in records:
        encrypted_data = self._process_record_fields(record_dict)
        self._update_record(encrypted_data, record_dict["applicant_id"])

    def _process_record_fields(self, record_dict: dict) -> dict:
        encrypted_data = {}
        for column, value in record_dict.items():
            if self._is_sensitive_field(column) and value:
                encrypted_data[column] = self.encryption.encrypt(str(value))
            else:
                encrypted_data[column] = value
        return encrypted_data

    def _is_sensitive_field(self, column_name: str) -> bool:
        sensitive_fields = {"first_name",
                            "last_name", "address", "phone_number"}
        return column_name.lower() in sensitive_fields

    def _update_record(self, data: dict, record_id):
        set_clauses = []
        values = []
        id_column = "applicant_id"

        for column, value in data.items():
            if column != id_column:
                set_clauses.append(f"{column} = %s")
                values.append(value)

        if set_clauses:
            values.append(record_id)
            update_query = f"UPDATE ApplicantProfile SET {',
'.join(set_clauses)} WHERE {id_column} = %s"
            self.db_manager._execute_query(
                update_query, tuple(values), commit=True)

```

Kelas Seeder dirancang untuk menginisialisasi database dengan data awal dan mengenkripsi informasi sensitif. Ini berinteraksi dengan db\_manager untuk operasi database dan encryption\_cipher untuk enkripsi data.

Fungsi intinya adalah seed\_and\_encrypt. Metode ini pertama-tama menjalankan kueri SQL dari file yang disediakan untuk mengisi database. Setelah data diisi, ia melanjutkan untuk mengenkripsi bidang sensitif yang sudah ada di database. Jika ada langkah yang gagal, ia akan mengembalikan transaksi database untuk memastikan integritas data.

Metode \_execute\_sql\_file membaca file SQL, memisahkannya menjadi pernyataan individual, dan menjalankannya satu per satu. Ia dirancang untuk melanjutkan pemrosesan bahkan jika pernyataan tertentu gagal, mengisolasi kegagalan dan mencegahnya menghentikan seluruh operasi.

Metode `_encrypt_existing_data` mengambil semua catatan dari tabel `ApplicantProfile`. Untuk setiap catatan, ia memanggil `_process_record_fields` untuk mengidentifikasi dan mengenkripsi bidang sensitif. Setelah bidang dienkripsi, `_update_record` kemudian memperbarui catatan di database dengan data terenkripsi.

Dua metode pembantu, `_process_record_fields` dan `_is_sensitive_field`, bekerja sama untuk mengidentifikasi bidang mana yang memerlukan enkripsi. `_is_sensitive_field` memiliki daftar bidang yang telah ditentukan sebelumnya yang dianggap sensitif, seperti nama, alamat, dan nomor telepon. `_process_record_fields` kemudian menggunakan daftar ini untuk secara selektif mengenkripsi nilai dalam catatan sebelum mengembalikannya.

Akhirnya, metode `_update_record` membangun kueri pembaruan SQL secara dinamis berdasarkan data terenkripsi yang disediakan dan ID catatan. Ini memastikan bahwa hanya bidang yang relevan yang diperbarui, menjaga kolom lain tetap utuh, dan kemudian mengeksekusi kueri pembaruan terhadap database.

## Encryption Algorithm

```
class VigenereCipher:
    PRINTABLE_ASCII_START = 32
    PRINTABLE_ASCII_END = 126
    PRINTABLE_ASCII_RANGE_SIZE = PRINTABLE_ASCII_END - PRINTABLE_ASCII_START + 1
    def _process_text(self, input_text: str, key: str, encrypt_mode: bool) -> str:
        if not key: return input_text

        processed_chars = []
        key_length = len(key)
        for i, text_char_code in enumerate(input_text):
            key_char = key[i % key_length]
            text_ord = ord(text_char_code)
            key_ord = ord(key_char)

            if (self.PRINTABLE_ASCII_START <= text_ord <=
                self.PRINTABLE_ASCII_END and
                self.PRINTABLE_ASCII_START <= key_ord <=
                self.PRINTABLE_ASCII_END):
                shift_amount = key_ord - self.PRINTABLE_ASCII_START
                normalized_text_ord = text_ord - self.PRINTABLE_ASCII_START
                if encrypt_mode:
                    processed_val = (normalized_text_ord + shift_amount) %
                        self.PRINTABLE_ASCII_RANGE_SIZE
                else:
                    processed_val = (normalized_text_ord - shift_amount +
                        self.PRINTABLE_ASCII_RANGE_SIZE) % self.PRINTABLE_ASCII_RANGE_SIZE
                final_processed_ord = processed_val +
                self.PRINTABLE_ASCII_START
                processed_chars.append(chr(final_processed_ord))
            else:
                processed_chars.append(text_char_code)
        return "".join(processed_chars)
```

```

def encrypt(self, plain_text: str, key: str) -> str:
    encrypted_text = self._process_text(plain_text, key, True)
    return encrypted_text

def decrypt(self, cipher_text: str, key: str) -> str:
    decrypted_text = self._process_text(cipher_text, key, False)
    return decrypted_text

```

Class VigenereCipher mengimplementasikan cipher Vigenère yang disederhanakan yang dirancang untuk enkripsi dan dekripsi teks. Penting untuk dicatat bahwa implementasi ini bersifat edukasi dan tidak aman secara kriptografis untuk penggunaan nyata. Cipher ini beroperasi pada karakter ASCII yang dapat dicetak, dengan rentang yang didefinisikan oleh PRINTABLE\_ASCII\_START (32) hingga PRINTABLE\_ASCII\_END (126).

Inti dari operasi enkripsi dan dekripsi terletak pada metode pembantu internal \_process\_text(self, input\_text: str, key: str, encrypt\_mode: bool). Metode ini mengambil teks masukan (plain\_text untuk enkripsi, cipher\_text untuk dekripsi) dan sebuah key string. Jika key kosong, teks dikembalikan tanpa perubahan. Prosesnya mengiterasi setiap karakter dalam input\_text. Untuk setiap karakter, ia menentukan karakter kunci yang relevan dengan menggunakan key secara berulang (misalnya, key[i % key\_length]).

Untuk karakter teks dan karakter kunci yang berada dalam rentang ASCII yang dapat dicetak, algoritma melakukan pergeseran modular:

- Enkripsi: Karakter teks digeser maju berdasarkan nilai karakter kunci. Pergeseran ini dilakukan dalam rentang PRINTABLE\_ASCII\_RANGE\_SIZE (yaitu, 95 karakter) dan hasilnya kemudian disesuaikan kembali ke offset ASCII awal.
- Dekripsi: Prosesnya dibalik; karakter teks digeser mundur untuk mengembalikan nilai aslinya, juga menggunakan aritmetika modular dalam rentang yang sama.

Karakter apa pun dalam input\_text atau key yang berada di luar rentang ASCII yang dapat dicetak akan dilewatkan tanpa perubahan. Hasilnya adalah string dari karakter yang telah diproses.

Metode publik encrypt(self, plain\_text: str, key: str) dan decrypt(self, cipher\_text: str, key: str) berfungsi sebagai wrapper sederhana untuk metode \_process\_text. Metode encrypt memanggil \_process\_text dengan encrypt\_mode diatur ke True, sementara decrypt memanggilnya dengan encrypt\_mode diatur ke False sehingga memberikan antarmuka yang jelas dan mudah digunakan untuk operasi cipher.

## Base Search Algorithm

```

class BaseSearchAlgorithm(ABC):
    @abstractmethod
    def search(self, text: str, pattern: any) -> any:
        pass

```

Class BaseSearchAlgorithm adalah kelas abstrak dasar yang berfungsi sebagai fondasi untuk semua algoritma pencarian di sistem. Kelas ini didefinisikan sebagai abstrak (ABC) yang berarti tidak dapat diinstansiasi secara langsung. Tujuannya adalah untuk mendefinisikan antarmuka standar yang harus dipatuhi oleh semua algoritma pencarian konkret yang diturunkan darinya.

Antarmuka ini diwujudkan melalui metode abstrak search. Metode search ini memerlukan implementasi spesifik di setiap kelas turunan, di mana mereka akan mendefinisikan bagaimana pencarian dilakukan pada text dan patterns tertentu, serta format nilai kembalinya. Dengan demikian, BaseSearchAlgorithm memastikan konsistensi dalam cara berbagai algoritma pencarian digunakan dalam aplikasi, sambil memungkinkan fleksibilitas dalam detail implementasi masing-masing algoritma.

## Exact Matching Algorithm

```
class ExactStringMatchingAlgorithm(BaseSearchAlgorithm):
    @abstractmethod
    def search(self, text: str, pattern: str) -> list[int]:
        pass
```

Class ExactStringMatchingAlgorithm adalah sebuah kelas abstrak dasar yang khusus dirancang untuk algoritma pencarian string eksak. Kelas ini merupakan turunan dari BaseSearchAlgorithm yang berarti ia mewarisi sifat-sifat dasar dari kelas pencarian umum, namun dengan fokus yang lebih spesifik.

Tujuan utama dari ExactStringMatchingAlgorithm adalah untuk menyediakan antarmuka standar bagi algoritma yang mencari semua kemunculan satu pola string (pattern) di dalam sebuah teks string (text) secara tepat. Metode inti dari kelas ini adalah search yang merupakan metode abstrak. Implementasi dari metode search ini di kelas-kelas turunan (seperti KMP atau Boyer-Moore) diharapkan untuk mengembalikan daftar indeks awal (berbasis 0) di mana pattern ditemukan di dalam text. Jika tidak ada kemunculan yang ditemukan, metode ini harus mengembalikan daftar kosong. Hal ini memastikan bahwa semua algoritma pencarian string eksak yang berbeda akan memiliki cara penggunaan dan keluaran yang konsisten.

## Knuth-Morris-Pratt Algorithm

```
class KMP(ExactStringMatchingAlgorithm):
    def _compute_lps_array(self, pattern: str) -> list[int]:
        m = len(pattern)
        lps = [0] * m
        length = 0
        i = 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
    return lps

def search(self, text: str, pattern: str) -> list[int]:
    n = len(text)
    m = len(pattern)
    if m == 0:
        return []
    if n == 0:
        return []
    if m > n:
        return []

    lps = self._compute_lps_array(pattern)
    i = 0
    j = 0
    occurrences = []

    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1
        if j == m:
            occurrences.append(i - j)
            j = lps[j - 1]
        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return occurrences

```

Class KMP adalah implementasi dari algoritma Knuth-Morris-Pratt (KMP) yang sangat efisien untuk pencarian string eksak. Artinya, algoritma ini dirancang untuk menemukan semua kemunculan yang persis sama dari sebuah pattern (pola string tunggal) di dalam sebuah text (teks string yang lebih besar). Karena merupakan turunan dari ExactStringMatchingAlgorithm, KMP mematuhi antarmuka standar untuk algoritma pencarian eksak.

Inti dari efisiensi algoritma KMP terletak pada kemampuannya untuk menghindari backtracking yang tidak perlu pada text ketika terjadi ketidakcocokan antara pattern dan text. Untuk mencapai ini, KMP menggunakan sebuah teknik pra-pemrosesan pada pattern itu sendiri yang menghasilkan apa yang disebut "LPS (Longest Proper Prefix which is also Suffix) array".

Metode `_compute_lps_array(self, pattern: str)` bertanggung jawab untuk membangun array LPS ini. Array LPS yang memiliki ukuran yang sama dengan pattern, menyimpan informasi penting untuk setiap posisi  $i$  dalam pattern: panjang prefiks terpanjang dari  $\text{pattern}[0 \dots i]$  yang juga merupakan sufiks dari  $\text{pattern}[0 \dots i]$ . Misalnya, jika pattern adalah "ABABCABAB", array LPS-nya adalah [0, 0, 1, 2, 0, 1, 2, 3, 4]. Nilai 0 pada indeks pertama menunjukkan bahwa prefiks "A" tidak memiliki proper sufiks yang juga prefiks. Nilai 2 pada indeks "ABABC" menunjukkan bahwa "AB" adalah prefiks terpanjang yang juga sufiks dari "ABABC". Informasi ini sangat

penting karena, ketika terjadi ketidakcocokan, array LPS memberitahu algoritma berapa banyak karakter yang dapat digeser pattern ke kanan tanpa harus memeriksa kembali karakter yang sudah diketahui cocok.

Setelah array LPS dihitung, metode search(self, text: str, pattern: str) mulai bekerja. Metode ini menginisialisasi dua pointer: i untuk iterasi melalui text dan j untuk iterasi melalui pattern. Algoritma membandingkan text[i] dengan pattern[j].

- Jika karakter-karakter cocok, kedua pointer (i dan j) dimajukan.
- Jika semua karakter dalam pattern cocok (yaitu, j mencapai panjang pattern), maka pola telah ditemukan, dan indeks awal kecocokan ( $i - j$ ) ditambahkan ke daftar occurrences. Setelah itu, j diatur ulang ke  $\text{lps}[j - 1]$  untuk mencari kecocokan berikutnya tanpa perlu memulai pattern dari awal.
- Jika terjadi ketidakcocokan ( $\text{pattern}[j] \neq \text{text}[i]$ ), algoritma tidak perlu memundurkan pointer i pada text. Sebaliknya, ia menggunakan array LPS: jika j bukan 0, j dipindahkan ke  $\text{lps}[j - 1]$  yang secara efektif menggeser pattern ke kanan berdasarkan informasi prefiks-sufiks yang cocok. Jika j adalah 0 (yaitu, ketidakcocokan terjadi pada karakter pertama pattern), maka hanya pointer i yang dimajukan untuk memeriksa karakter text berikutnya.

Proses ini terus berlanjut hingga seluruh text telah diperiksa. Hasil akhir adalah daftar semua indeks awal di mana pattern ditemukan di dalam text. Keunggulan utama KMP adalah kompleksitas waktu terbaiknya yang linear ( $O(n+m)$ ), di mana n adalah panjang teks dan m adalah panjang pola.

## Boyer-Moore Algorithm

```
class BoyerMoore(ExactStringMatchingAlgorithm):  
    def __init__(self, pattern: str) -> dict[str, int]:  
        m = len(pattern)  
        bad_char = {}  
  
        for i in range(m):  
            bad_char[pattern[i]] = i  
  
        return bad_char  
  
    def search(self, text: str, pattern: str) -> list[int]:  
        n = len(text)  
        m = len(pattern)  
        if m == 0:  
            return []  
        if n == 0:  
            return []  
        if m > n:  
            return []  
  
        bad_char = self.__init__(pattern)  
        occurrences = []  
        s = 0
```

```

while s <= (n - m):
    j = m - 1
    while j >= 0 and pattern[j] == text[s + j]:
        j -= 1

    if j < 0:
        occurrences.append(s)
        if s + m < n:
            next_char = text[s + m]
            if next_char in bad_char:
                shift = m - bad_char[next_char] - 1
            else:
                shift = m
            s += max(1, shift)
        else:
            s += 1
    else:
        mismatched_char = text[s + j]
        if mismatched_char in bad_char:
            shift = j - bad_char[mismatched_char]
        else:
            shift = j + 1
        s += max(1, shift)

return occurrences

```

Class BoyerMoore mengimplementasikan algoritma pencarian string Boyer-Moore yang dirancang untuk menemukan semua kemunculan pola (pattern) secara eksak dalam sebuah teks (text) dengan efisien. Sebagai turunan dari ExactStringMatchingAlgorithm, kelas ini menyediakan metode search yang konsisten untuk pencarian string tunggal.

Salah satu ciri khas algoritma Boyer-Moore adalah cara membandingkan pola: tidak seperti algoritma lain yang membandingkan dari kiri ke kanan, Boyer-Moore memulai perbandingan dari kanan ke kiri pola. Efisiensi utamanya berasal dari heuristik pergeseran yang cerdas saat terjadi ketidakcocokan. Implementasi ini menggunakan "bad character heuristic" yang dihitung oleh metode pembantu \_bad\_char\_heuristic(self, pattern: str). Metode ini membangun tabel pergeseran (bad\_char) yang menyimpan indeks kemunculan terkaitan dari setiap karakter dalam pola.

Dalam metode search(self, text: str, pattern: str), setelah kasus dasar (pola atau teks kosong, pola lebih panjang dari teks) ditangani, algoritma memulai proses pencarian. Sebuah jendela s (shift) bergerak sepanjang text. Di setiap posisi jendela, perbandingan dimulai dari karakter terakhir pattern ( $j = m - 1$ ) dan bergerak mundur ke kiri.

- Jika semua karakter dalam pola cocok (yaitu,  $j$  menjadi kurang dari 0), maka kecocokan ditemukan, dan indeks awal  $s$  ditambahkan ke daftar occurrences. Pola kemudian digeser ke kanan.
- Jika terjadi ketidakcocokan pada  $\text{text}[s + j]$  dengan  $\text{pattern}[j]$ , algoritma menggunakan "bad character heuristic". Ia melihat karakter yang tidak cocok pada text (mismatched\_char).

- Jika mismatched\_char ada di tabel bad\_char, pola digeser sedemikian rupa sehingga mismatched\_char dalam text sejajar dengan kemunculan terkanannya di pattern yang berada di sebelah kiri posisi j. Pergeseran dihitung sebagai  $j - \text{bad\_char}[mismatched\_char]$ .
- Jika mismatched\_char tidak ada di pattern, pola digeser melampaui mismatched\_char sepenuhnya ( $j + 1$ ). Pergeseran selalu diambil nilai maksimum 1 atau nilai yang dihitung untuk memastikan kemajuan. Proses ini berlanjut hingga seluruh teks telah diperiksa, dan daftar semua indeks awal kecocokan dikembalikan. Algoritma Boyer-Moore seringkali sangat cepat karena kemampuannya untuk melakukan "lompatan" besar dalam teks.

## String Similarity Algorithm

```
class StringSimilarityAlgorithm(ABC):
    @abstractmethod
    def calculate_distance(self, s1: str, s2: str) -> int | float:
        pass

    @abstractmethod
    def calculate_similarity_percentage(self, s1: str, s2: str) -> float:
        pass
```

Class StringSimilarityAlgorithm adalah sebuah kelas abstrak dasar yang berfungsi sebagai fondasi untuk semua algoritma yang mengukur kemiripan antar string. Didefinisikan sebagai kelas abstrak (ABC), ia tidak dapat diinstansiasi secara langsung, melainkan berfungsi sebagai template atau kontrak yang harus dipatuhi oleh kelas-kelas konkret turunannya. Tujuannya adalah untuk memastikan konsistensi dalam cara berbagai algoritma kemiripan string diimplementasikan dan digunakan dalam sistem.

Kelas ini mendefinisikan dua metode abstrak yang harus diimplementasikan oleh setiap algoritma kemiripan string konkret:

1. calculate\_distance(self, s1: str, s2: str) -> int | float: Metode ini bertanggung jawab untuk menghitung "jarak" atau ketidakmiripan antara dua string, s1 dan s2. Secara umum, nilai jarak yang lebih rendah menunjukkan bahwa kedua string tersebut lebih mirip. Tipe nilai kembalinya bisa berupa integer atau float, tergantung pada implementasi spesifik algoritma jarak yang digunakan.
2. calculate\_similarity\_percentage(self, s1: str, s2: str) -> float: Metode ini bertugas menghitung persentase kemiripan antara dua string, s1 dan s2. Nilai kembalinya adalah float yang berkisar antara 0.0 hingga 100.0, di mana 100.0% menunjukkan string yang identik dan 0.0% menunjukkan tidak ada kemiripan sama sekali.

## Levenshtein

```
class Levenshtein(StringSimilarityAlgorithm):
    def calculate_similarity_percentage(self, s1: str, s2: str) -> float:
        max_len = max(len(s1), len(s2))
        if max_len == 0:
            return 100.0
        distance = self.calculate_distance(s1, s2)
```

```

        return (1 - (distance / max_len)) * 100

    def calculate_distance(self, s1: str, s2: str) -> int:
        if len(s1) < len(s2):
            return self.calculate_distance(s2, s1)

        if len(s2) == 0:
            return len(s1)

        previous_row = range(len(s2) + 1)
        for i, c1 in enumerate(s1):
            current_row = [i + 1]
            for j, c2 in enumerate(s2):
                insertions = previous_row[j + 1] + 1
                deletions = current_row[j] + 1
                substitutions = previous_row[j] + (c1 != c2)
                current_row.append(min(insertions, deletions, substitutions))
            previous_row = current_row

    return previous_row[-1]

```

Class Levenshtein mengimplementasikan algoritma jarak Levenshtein yang merupakan metode untuk mengukur kemiripan antara dua string. Kelas ini merupakan turunan dari StringSimilarityAlgorithm sehingga ia mematuhi antarmuka standar untuk algoritma kemiripan string yang mengharuskan implementasi metode untuk menghitung jarak dan persentase kemiripan.

Metode inti dari kelas ini adalah calculate\_distance(self, s1: str, s2: str) yang menghitung jarak Levenshtein antara dua string yang diberikan. Jarak Levenshtein didefinisikan sebagai jumlah minimum operasi edit karakter tunggal (penyisipan, penghapusan, atau substitusi) yang diperlukan untuk mengubah satu string menjadi string lainnya. Implementasi ini menggunakan pendekatan pemrograman dinamis, membangun matriks (diwakili oleh baris sebelumnya dan baris saat ini) untuk menghitung biaya edit minimum secara iteratif. Semakin kecil nilai jarak yang dihasilkan, semakin mirip kedua string tersebut.

Selain itu, kelas Levenshtein juga menyediakan metode calculate\_similarity\_percentage(self, s1: str, s2: str). Metode ini mengubah nilai jarak Levenshtein menjadi persentase kemiripan yang lebih intuitif, berkisar antara 0.0 hingga 100.0. Persentase ini dihitung berdasarkan jarak yang diperoleh relatif terhadap panjang string maksimum; 100% berarti string identik (jarak 0), sementara persentase yang lebih rendah menunjukkan tingkat ketidakmiripan yang lebih tinggi. Hal ini membuat kelas Levenshtein sangat berguna untuk aplikasi yang membutuhkan pencocokan "fuzzy", seperti koreksi ejaan atau pencarian teks yang toleran terhadap kesalahan.

## Multi Pattern Algorithm

```

class MultiPatternStringMatchingAlgorithm(BaseSearchAlgorithm):
    @abstractmethod
    def search(self, text:str, patterns: list[str]) -> dict[str, list[int]]:
        pass

```

Class MultiPatternStringMatchingAlgorithm adalah kelas abstrak dasar yang khusus dirancang untuk algoritma pencocokan multi-pola string. Ini merupakan turunan dari BaseSearchAlgorithm, menunjukkan bahwa ia adalah bentuk khusus dari algoritma pencarian. Tujuannya adalah untuk mendefinisikan antarmuka standar yang harus dipatuhi oleh algoritma apa pun yang mampu mencari beberapa pola string sekaligus di dalam sebuah teks string tunggal.

Metode intinya adalah search yang merupakan metode abstrak. Metode ini menerima dua argumen: text (teks tempat pencarian dilakukan) dan patterns (sebuah daftar string yang merupakan pola-pola yang akan dicari). Berbeda dengan algoritma pencarian eksak tunggal, metode search pada kelas ini diharapkan mengembalikan sebuah dictionary. Di dalam dictionary ini, setiap key adalah pola string yang ditemukan dalam teks, dan value terkait adalah daftar indeks awal (0-based) di mana pola tersebut muncul dalam teks.

## Aho-Corasick Algorithm

```
class AhoCorasick(MultiPatternStringMatchingAlgorithm):
    def __init__(self):
        super().__init__()
        self._nodes = []
        self._pattern_map = {}
        self._automaton_built_for_patterns = None
        self._initialize_automaton()

    def _initialize_automaton(self):
        self._nodes = [{'children': {}, 'parent': None,
                       'char': None, 'failure_link': 0, 'output': set()}]
        self._pattern_map = {}
        self._automaton_built_for_patterns = None

    def _build_automaton(self, patterns: list[str]):
        self._initialize_automaton()

        valid_patterns = [p for p in patterns if p]
        if not valid_patterns:
            self._automaton_built_for_patterns = tuple(
                sorted(patterns))
            return

        self._pattern_map = {i: pattern_str for i,
                            pattern_str in enumerate(valid_patterns)}

        for pattern_idx, pattern_str in self._pattern_map.items():
            node_idx = 0
            for char_in_pattern in pattern_str:
                if char_in_pattern not in self._nodes[node_idx]['children']:
                    new_node_idx = len(self._nodes)
                    self._nodes[node_idx]['children'][char_in_pattern] =
new_node_idx
                    self._nodes.append({
                        'children': {},
                        'parent': node_idx,
```

```

        'char': char_in_pattern,
        'failure_link': 0,
        'output': set()
    })
    node_idx = self._nodes[node_idx]['children'][char_in_pattern]

    self._nodes[node_idx]['output'].add(pattern_idx)

queue = deque()

for child_node_idx in self._nodes[0]['children'].values():
    queue.append(child_node_idx)

while queue:
    current_node_idx = queue.popleft()
    for char, next_node_idx in
self._nodes[current_node_idx]['children'].items():
        queue.append(next_node_idx)
        failure_candidate_idx =
self._nodes[current_node_idx]['failure_link']
        while char not in
self._nodes[failure_candidate_idx]['children'] and failure_candidate_idx != 0:
            failure_candidate_idx =
self._nodes[failure_candidate_idx]['failure_link']

            if char in self._nodes[failure_candidate_idx]['children']:
                self._nodes[next_node_idx]['failure_link'] =
self._nodes[failure_candidate_idx]['children'][char]
            else:
                pass

            output_from_failure = self._nodes[self._nodes[next_node_idx]
['failure_link']]['output']
            if output_from_failure:
                self._nodes[next_node_idx]['output'].update(
                    output_from_failure)

self._automaton_built_for_patterns = tuple(sorted(patterns))

```

Class AhoCorasick mengimplementasikan algoritma Aho-Corasick, sebuah metode yang sangat efisien untuk pencocokan multi-pola string. Ini merupakan turunan dari MultiPatternStringMatchingAlgorithm, menegaskan perannya dalam mencari beberapa pola sekaligus dalam sebuah teks.

Saat diinisialisasi, kelas ini menyiapkan beberapa struktur data internal yang penting: `_nodes`, sebuah daftar yang mewakili node-node dalam automaton Aho-Corasick; `_pattern_map`, sebuah dictionary untuk memetakan indeks internal pola ke string pola yang sebenarnya; dan `_automaton_built_for_patterns` yang melacak set pola yang terakhir digunakan untuk membangun automaton guna menghindari pembangunan ulang yang tidak perlu. Metode `_initialize_automaton` dipanggil saat inisialisasi dan juga oleh `_build_automaton` untuk mereset struktur ini, memastikan automaton selalu dibangun dari keadaan bersih. Node awal (root)

automaton dibuat dengan children, parent, char, failure\_link (mengarah ke dirinya sendiri untuk root), dan output (untuk menyimpan indeks pola yang berakhir di node tersebut).

Bagian inti dari persiapan algoritma ini adalah metode \_build\_automaton(self, patterns: list[str]). Metode ini bertanggung jawab untuk membangun struktur data utama yang digunakan untuk pencarian multi-pola: automaton Aho-Corasick. Proses ini dibagi menjadi dua fase:

1. Membangun Trie (Goto Function): Setiap pola dalam daftar patterns disisipkan ke dalam struktur data yang mirip pohon yang disebut Trie. Setiap node baru ditambahkan ke daftar \_nodes, dan children dari node induk diperbarui untuk menunjuk ke node baru. Setelah seluruh pola disisipkan, node terakhir dari pola tersebut ditandai dengan menambahkan indeks polanya ke set output node.
2. Membangun Fungsi Failure dan Output Lanjutan: Setelah Trie dasar terbentuk, "failure links" dan "output links" untuk setiap node dihitung menggunakan algoritma Breadth-First Search (BFS) yang memanfaatkan collections.deque sebagai antrean. Failure link untuk sebuah node u menunjuk ke node v terpanjang yang merupakan sufiks dari string yang diwakili oleh u dan juga merupakan prefiks dari salah satu pola. Ini memungkinkan algoritma untuk melakukan transisi secara efisien setelah terjadi ketidakcocokan tanpa harus memundurkan teks. Selain itu, set output dari sebuah node diperbarui untuk mencakup semua pola yang berakhir di node itu sendiri ditambah semua pola yang berakhir di node yang ditunjuk oleh failure link-nya. Setelah semua failure links dan output diperbarui, \_automaton\_built\_for\_patterns disetel untuk mencerminkan pola-pola yang telah diproses yang kemudian digunakan oleh metode search untuk menentukan apakah automaton perlu dibangun ulang.

```
def search(self, text: str, patterns: list[str]) -> dict[str, list[int]]:  
    if not text or not patterns:  
        return {}  
  
    current_patterns_tuple = tuple(  
        sorted(p for p in patterns if p))  
  
    if not current_patterns_tuple:  
        return {}  
  
    if self._automaton_built_for_patterns != current_patterns_tuple:  
        self._build_automaton([p for p in patterns if p])  
  
    if len(self._nodes) <= 1 and not self._nodes[0]['children']:  
        return {}  
  
    results = {self._pattern_map[idx]: [] for idx in self._pattern_map}  
    current_node_idx = 0  
  
    for i, char_in_text in enumerate(text):  
        while char_in_text not in self._nodes[current_node_idx]['children']  
and current_node_idx != 0:  
            current_node_idx = self._nodes[current_node_idx]['failure_link']  
  
        if char_in_text in self._nodes[current_node_idx]['children']:  
            results[i].append(current_node_idx)
```

```

        current_node_idx =
self._nodes[current_node_idx]['children'][char_in_text]
    else:
        pass

    if self._nodes[current_node_idx]['output']:
        for pattern_idx in self._nodes[current_node_idx]['output']:
            matched_pattern_str = self._pattern_map[pattern_idx]

            start_index = i - len(matched_pattern_str) + 1
            results[matched_pattern_str].append(start_index)

final_results = {}
for pattern_str, indices in results.items():
    if indices:
        final_results[pattern_str] = sorted(list(set(indices)))

return final_results

```

Setelah membangkitkan automaton (yang melibatkan pembangunan Trie dan komputasi failure serta output links), metode search pada kelas AhoCorasick siap untuk memproses teks input dan menemukan semua kemunculan pola-pola yang telah didefinisikan.

Langkah pertama dalam metode search adalah validasi input. Ini mencakup pemeriksaan untuk memastikan bahwa baik text maupun daftar patterns tidak kosong. Selain itu, metode ini secara cerdas memeriksa apakah set pola yang saat ini akan dicari (current\_patterns\_tuple) berbeda dari set pola yang digunakan untuk membangun automaton terakhir kali (\_automaton\_built\_for\_patterns). Jika ada perbedaan, ini menandakan bahwa automaton perlu dibangun ulang dengan pola-pola baru, dan metode \_build\_automaton akan dipanggil untuk memastikan automaton selalu relevan dengan pencarian saat ini.

Setelah automaton dipastikan siap, proses pencarian dimulai dengan inisialisasi sebuah dictionary results untuk menyimpan semua kemunculan pola yang akan ditemukan, dengan kunci berupa string pola dan nilai berupa daftar indeks awal kemunculan. Sebuah pointer current\_node\_idx diatur ke node root automaton (indeks 0) yang merupakan titik awal traversal.

Kemudian, algoritma mengiterasi melalui setiap karakter (char\_in\_text) dalam text input menggunakan indeks i. Untuk setiap karakter, ada dua mekanisme utama yang bekerja:

1. Transisi State: Algoritma mencoba menemukan transisi dari current\_node\_idx yang cocok dengan char\_in\_text.
  - o Jika transisi langsung ditemukan, current\_node\_idx bergerak ke node anak yang sesuai.
  - o Namun, jika tidak ada transisi langsung yang cocok dari current\_node\_idx, algoritma tidak berhenti atau mundur di text. Sebaliknya, ia secara rekursif mengikuti "failure links" dari current\_node\_idx hingga menemukan node di mana char\_in\_text memiliki transisi yang cocok, atau hingga kembali ke node root (jika tidak ada kecokongan transisi yang ditemukan di jalur failure link). Pergerakan cerdas melalui failure links ini memastikan bahwa algoritma terus maju di dalam teks dan tidak pernah "kembali" ke karakter yang sudah diproses, menjadikannya sangat efisien.

2. Identifikasi Output/Kecocokan: Setelah `current_node_idx` diperbarui (baik melalui transisi langsung maupun melalui `failure link`), algoritma memeriksa set output dari node `current_node_idx`. Set output ini berisi indeks dari semua pola yang berakhir di node tersebut atau yang berakhir di node yang dapat dijangkau melalui `failure link` dari node tersebut. Untuk setiap `pattern_idx` dalam set output tersebut, string pola yang sesuai (`matched_pattern_str`) diambil dari `_pattern_map`. Indeks awal kemunculan pola di dalam teks dihitung sebagai  $i - \text{len}(\text{matched\_pattern\_str}) + 1$  dan ditambahkan ke daftar `results` untuk pola yang bersangkutan.

Setelah seluruh teks selesai diiterasi, `dictionary results` akan berisi semua pola yang ditemukan beserta daftar indeks kemunculannya. Sebagai langkah final, `results` diproses untuk memastikan setiap daftar indeks kemunculan diurutkan dan hanya berisi nilai unik (menghilangkan duplikat yang mungkin terjadi karena sifat tumpang tindih dari kecocokan atau mekanisme `failure link`), sebelum dikembalikan sebagai `final_results`.

### **Analisis Kompleksitas dan Benchmarking**

Dalam membangun sistem ATS yang efisien, pemilihan algoritma pencocokan string memainkan peran krusial, terutama ketika sistem harus memindai puluhan hingga ratusan dokumen CV digital dalam waktu singkat. Oleh karena itu, dilakukan analisis kompleksitas teoritis terhadap algoritma-algoritma utama yang digunakan, yakni Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), dan Aho-Corasick (AC), diikuti dengan pengujian benchmarking untuk mengevaluasi performanya di lingkungan sistem.

Secara kompleksitas waktu, algoritma KMP memiliki performa teoretis  $O(n + m)$  di mana  $n$  adalah panjang teks dan  $m$  panjang pola, dengan keunggulan menghindari pencocokan ulang melalui tabel LPS. Boyer-Moore dalam praktik sering kali lebih cepat berkat strategi lompatan besar dari kanan ke kiri berdasarkan heuristik bad character, meskipun kompleksitas terburuknya  $O(nm)$ . Sementara itu, Aho-Corasick unggul untuk pencarian banyak pola secara serentak, dengan kompleksitas  $O(n + k + z)$  (dimana  $k$  total panjang semua pola dan  $z$  jumlah kecocokan), menjadikannya ideal dalam skenario multi-keyword.

Untuk mengukur performa aktual, dilakukan pengujian terhadap 2484 CV dengan berbagai skenario pencarian. Hasil benchmarking menunjukkan perbedaan yang signifikan antar algoritma tergantung pada jumlah keyword dan jumlah hasil yang diminta:

Algoritma	Keyword	Top Matches	Waktu Eksekusi (ms)
KMP	finance (1 keyword)	50	1105.2
	science, mathematics (2 keyword)	5	2335.3
Boyer-Moore	finance (1 keyword)	50	443.2
	science, mathematics	5	745.8

	(2 keyword)		
Aho-Corasick	finance (1 keyword)	50	1473.9
	science, mathematics (2 keyword)	5	1564.8

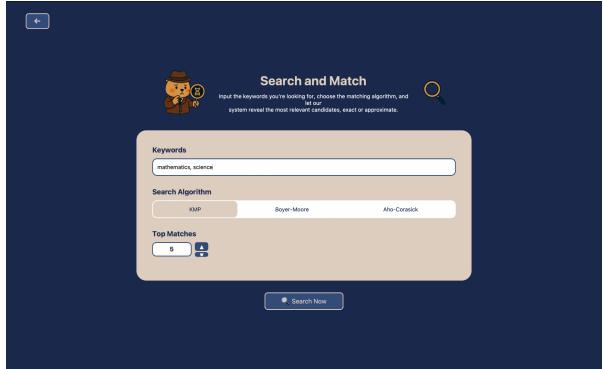
Dari tabel di atas, Boyer-Moore terbukti menjadi algoritma tercepat dalam kasus pencarian satu keyword, bahkan mengalahkan KMP hampir 2.5 kali lebih cepat. Namun, saat menangani banyak keyword seperti pada kasus science dan mathematics, performa Aho-Corasick menjadi lebih unggul dibanding KMP, karena mampu mencari seluruh keyword dalam satu kali traversal teks.

Secara umum, hasil benchmark memperkuat analisis teoritis: KMP cocok untuk pencocokan eksak tunggal dengan struktur data ringan, Boyer-Moore efisien untuk keyword panjang dan teks besar, sedangkan Aho-Corasick unggul pada pencarian multi-pattern secara simultan. Selain itu, variasi jumlah hasil yang ditampilkan (top match) juga memengaruhi waktu eksekusi secara linear, karena sistem perlu menyortir dan mengurutkan hasil sesuai skor kemunculan keyword.

Dengan adanya data ini, sistem ATS dapat mengadopsi pendekatan adaptif: menggunakan Boyer-Moore untuk pencarian tunggal cepat, dan Aho-Corasick ketika jumlah kata kunci melebihi satu. Pendekatan berbasis performa ini mendukung efisiensi sistem secara keseluruhan dan memberikan fleksibilitas pemilihan algoritma berdasarkan kebutuhan pengguna.

## Pengujian Test Cases

Input	Output



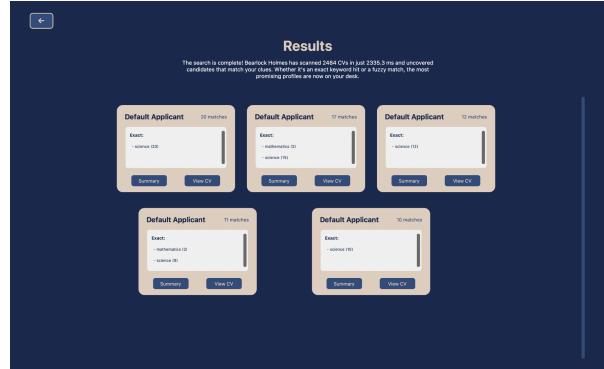
**Search and Match**  
Input the keywords you're looking for, choose the matching algorithm, and let our system reveal the most relevant candidates, exact or approximate.

Keywords: mathematics, science

Search Algorithm: KMP (selected), Boyer-Moore, Aho-Corasick

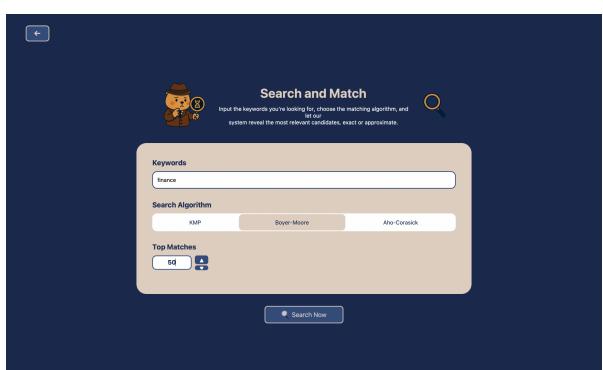
Top Matches: 5

Search Now



**Results**  
The search is complete! BeardoK Holmes has scanned 244 CVs in just 2315.5 ms and uncovered 6 candidates that match your query. Whether it's an exact keyword hit or a fuzzy match, the most promising profiles are now on your desk.

Default Applicant	Exact	Approximate
Default Applicant	25 matches	
Default Applicant	17 matches	
Default Applicant	12 matches	
Default Applicant	11 matches	
Default Applicant	10 matches	



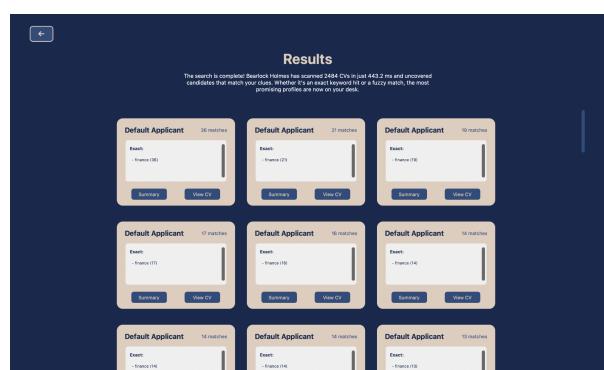
**Search and Match**  
Input the keywords you're looking for, choose the matching algorithm, and let our system reveal the most relevant candidates, exact or approximate.

Keywords: france

Search Algorithm: KMP (selected), Boyer-Moore, Aho-Corasick

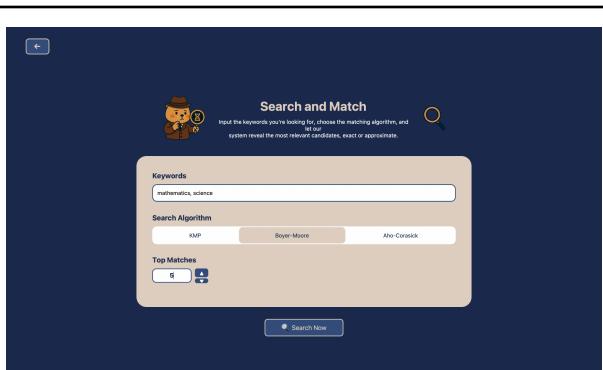
Top Matches: 50

Search Now



**Results**  
The search is complete! BeardoK Holmes has scanned 244 CVs in just 442.5 ms and uncovered 10 candidates that match your query. Whether it's an exact keyword hit or a fuzzy match, the most promising profiles are now on your desk.

Default Applicant	Exact	Approximate
Default Applicant	36 matches	
Default Applicant	21 matches	
Default Applicant	19 matches	
Default Applicant	17 matches	
Default Applicant	16 matches	
Default Applicant	14 matches	
Default Applicant	14 matches	
Default Applicant	13 matches	



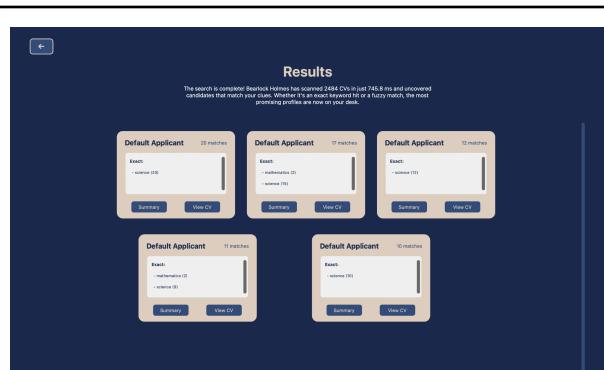
**Search and Match**  
Input the keywords you're looking for, choose the matching algorithm, and let our system reveal the most relevant candidates, exact or approximate.

Keywords: mathematics, science

Search Algorithm: KMP (selected), Boyer-Moore, Aho-Corasick

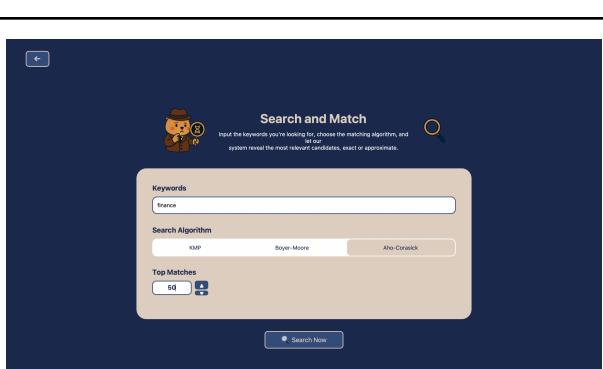
Top Matches: 5

Search Now



**Results**  
The search is complete! BeardoK Holmes has scanned 244 CVs in just 246.5 ms and uncovered 6 candidates that match your query. Whether it's an exact keyword hit or a fuzzy match, the most promising profiles are now on your desk.

Default Applicant	Exact	Approximate
Default Applicant	25 matches	
Default Applicant	17 matches	
Default Applicant	12 matches	
Default Applicant	11 matches	
Default Applicant	10 matches	



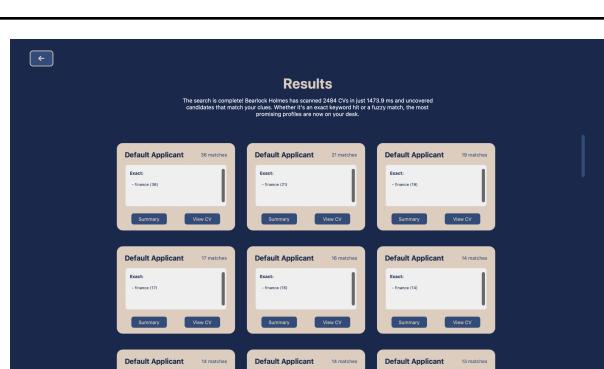
**Search and Match**  
Input the keywords you're looking for, choose the matching algorithm, and let our system reveal the most relevant candidates, exact or approximate.

Keywords: france

Search Algorithm: KMP (selected), Boyer-Moore, Aho-Corasick

Top Matches: 50

Search Now



**Results**  
The search is complete! BeardoK Holmes has scanned 244 CVs in just 147.5 ms and uncovered 10 candidates that match your query. Whether it's an exact keyword hit or a fuzzy match, the most promising profiles are now on your desk.

Default Applicant	Exact	Approximate
Default Applicant	36 matches	
Default Applicant	21 matches	
Default Applicant	19 matches	
Default Applicant	17 matches	
Default Applicant	16 matches	
Default Applicant	14 matches	
Default Applicant	14 matches	
Default Applicant	13 matches	

The image displays two screenshots of a web-based search and matching application. The left screenshot shows the 'Search and Match' interface with a search bar containing 'sales, marketing, director'. The right screenshot shows the 'Results' page displaying multiple search results for different applicants.

**Search and Match**

Input the keywords you're looking for, choose the matching algorithm, and let our system reveal the most relevant candidates, exact or approximate.

**Keywords**  
sales, marketing, director

**Search Algorithm**  
KMP      Boyer-Moore      Aho-Corasick

**Top Matches**  
15

**Results**

The search is complete! Headlock Haven has scanned 3484 CVs in just 164.4 ms and uncovered 6 databases that match your search criteria. Based on our Kullback-Leibler or Jaccard metric, the most promising profiles are now on your desk.

**Default Applicant** 12 matches  
Exact:  
- sales (18)  
- marketing (16)

**Default Applicant** 62 matches  
Exact:  
- sales (28)  
- marketing (26)

**Default Applicant** 64 matches  
Exact:  
- sales (14)  
- marketing (16)

**Default Applicant** 62 matches  
Exact:  
- sales (16)  
- marketing (17)

**Default Applicant** 55 matches  
Exact:  
- sales (22)

**Default Applicant** 53 matches  
Exact:  
- sales (17)

**Default Applicant** 50 matches  
Exact:  
- sales (16)

## **BAB V: Kesimpulan dan Saran**

### **Kesimpulan**

Sistem ATS yang dikembangkan terbukti efektif dalam mengotomatisasi pencarian dan ekstraksi informasi dari dokumen CV. Integrasi algoritma pencocokan string seperti KMP, Boyer-Moore, dan Aho-Corasick, serta algoritma fuzzy matching Levenshtein, mampu menangani berbagai skenario pencarian keyword baik secara eksak maupun toleran terhadap kesalahan ketik.

Implementasi sistem mendukung proses seleksi kandidat yang lebih efisien dengan skor dan pengurutan otomatis berdasarkan relevansi keyword. Seluruh data penting pelamar juga berhasil diekstraksi ke dalam basis data SQL yang terstruktur. Antarmuka pengguna interaktif mempermudah proses pencarian dan peninjauan hasil.

Secara keseluruhan, proyek ini berhasil menjawab tantangan dalam pengolahan CV digital dengan pendekatan algoritmik yang efisien dan modular.

### **Saran**

Saat ini, sistem menggunakan Levenshtein Distance untuk fuzzy matching, yang efektif dalam mengukur jumlah operasi edit karakter yang dibutuhkan. Namun, untuk skenario yang lebih kompleks, di mana ada variasi penulisan yang signifikan atau penggunaan singkatan, pertimbangkan untuk mengimplementasikan atau mengintegrasikan algoritma fuzzy matching lainnya. Misalnya, Jaro-Winkler Distance dapat lebih robust dalam menangani kesalahan tipografi yang umum dan lebih cocok untuk pencocokan nama atau istilah teknis yang mungkin memiliki variasi minor. Alternatif lain adalah Cosine Similarity yang dikombinasikan dengan representasi word embedding. Pendekatan ini memungkinkan perbandingan semantik, bukan hanya karakter, sehingga kata kunci seperti "pengembang perangkat lunak" dan "developer" dapat dikenali sebagai frasa yang sangat mirip.

## Lampiran

Repositori Github: [https://github.com/fathurwithyou/Tubes3\\_VitaeLangX](https://github.com/fathurwithyou/Tubes3_VitaeLangX)

Video Bonus Penggerjaan:

<https://www.youtube.com/watch?v=qvQ1hmGUuQ&si=R9wXceZIWHh5-uAS>

Tabel I. Checklist Penggerjaan

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

## Daftar Pustaka

CP-Algorithms. (n.d.). Aho-Corasick algorithm. Diakses pada 23 Mei 2025, dari [https://cp-algorithms.com/string/aho\\_corasick.html](https://cp-algorithms.com/string/aho_corasick.html)

Wikipedia contributors. (n.d.). Boyer–Moore string-search algorithm. Wikipedia, The Free Encyclopedia. Diakses pada 23 Mei 2025, dari [https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore\\_string-search\\_algorithm](https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm)

CP-Algorithms. (n.d.). Prefix function – Knuth–Morris–Pratt. Diakses pada 23 Mei 2025, dari <https://cp-algorithms.com/string/prefix-function.html>