LAPORAN TUGAS BESAR 3 IF2211 STRATEGI ALGORITMA

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

Dosen Pengampu : Dr. Nur Ulfa Maulidevi, S.T., M.Sc.

Asisten Pembimbing : Ahmad Rafi Maliki (13522137)

Disusun oleh:

Kelompok 56 – CLCC

Nathaniel Jonathan Rusli (13523013)

Kenneth Poenadi (13523040)

Steven Owen Liauw (13523103)

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

KATA PENGANTAR

Dengan penuh rasa syukur, kami menyusun dan mempersembahkan laporan ini yang membahas topik yang sangat relevan dalam dunia rekrutmen digital dan strategi algoritma, yaitu "Pemanfaatan Pattern Matching untuk Membangun Sistem ATS (Applicant Tracking System) Berbasis CV Digital". Laporan ini disusun sebagai bagian dari pemenuhan Tugas Besar 3 IF2211 Strategi Algoritma yang mengaplikasikan pemahaman tentang algoritma *string matching* dalam memecahkan tantangan penyaringan kandidat di dunia nyata.

Laporan ini tidak hanya membahas konsep dasar dari *pattern matching*, tapi juga mengeksplorasi penerapannya dalam mengolah data yang tidak terstruktur seperti dokumen CV berformat PDF. Algoritma *exact match* yang menjadi fokus utama adalah Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), dan Aho-Corasick yang menawarkan efisiensi tinggi untuk pencarian teks pada data berukuran besar.

Melalui laporan ini, kami berharap dapat memberikan penjelasan yang komprehensif mengenai hubungan antara teori algoritma *string matching* dan implementasi praktisnya dalam sistem rekrutmen modern. Kami juga memperkenalkan penggunaan *Regular Expression* (Regex) untuk mengekstraksi informasi penting seperti keahlian dan riwayat pendidikan , serta algoritma *Levenshtein Distance* untuk menangani kemungkinan kesalahan pengetikan (*typo*) pada kata kunci melalui pendekatan *fuzzy matching*.

Kami mengucapkan terima kasih kepada dosen pengampu dan semua pihak yang telah memberikan arahan, dukungan, serta semangat selama proses penyusunan laporan ini. Akhir kata, kami berharap laporan ini dapat menjadi sumber inspirasi bagi para pembaca untuk terus mengeksplorasi peluang dan tantangan baru di bidang strategi algoritma, pemrosesan teks, dan pengembangan perangkat lunak yang inovatif.

Bandung, 15 Juni 2025

Nathaniel Jonathan Rusli (13523013)
Kenneth Poenadi (13523040)
Steven Owen Liauw (13523103)

DAFTAR ISI

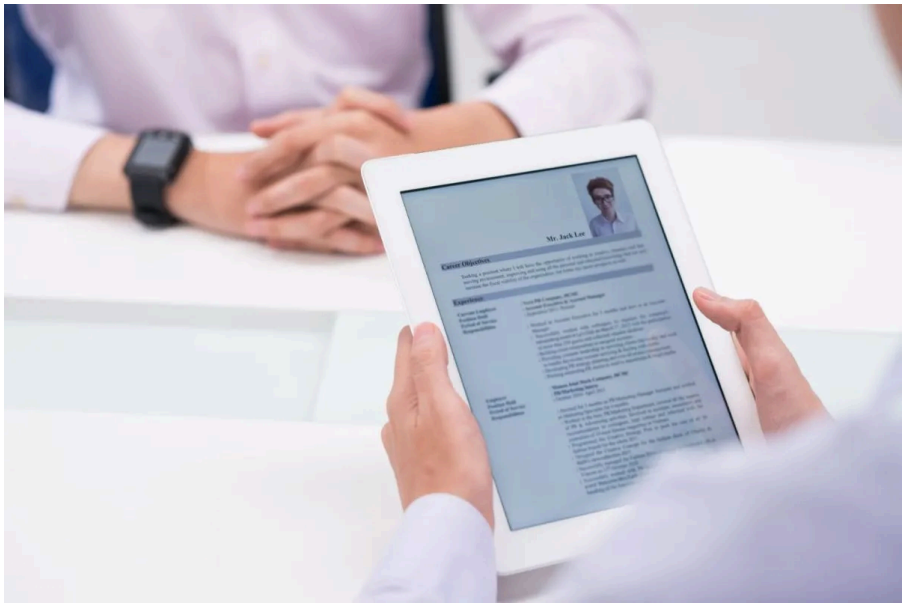
KATA PENGANTAR.....2
DAFTAR ISI.....3
DAFTAR GAMBAR.....5
BAB I DESKRIPSI TUGAS.....6
BAB II LANDASAN TEORI.....8
2.1 Knuth-Morris-Pratt (KMP).....8
2.2 Boyer-Moore (BM).....11
2.3 Aho-Corasick.....13
2.4 Fuzzy String Matching (Approximate Matching).....17
2.5 Levenshtein Distance.....18
2.6 Regular Expression (Regex).....18
BAB III ANALISIS PEMECAHAN MASALAH.....20
3.1 Cara Kerja Program.....20
3.2 Proses Pemecahan Masalah.....21
3.2.1 KMP (Knuth Morris Pratt).....21
3.2.2 Boyer-Moore.....28
3.2.3 Aho-Corasick.....30
3.2.4 Regex.....32
3.2.5 RSA Algorithm.....34
BAB IV IMPLEMENTASI DAN PENGUJIAN.....38
4.1 Spesifikasi Teknis Program.....38
4.1.1. Struktur Data.....38
4.1.2. Fungsi dan Prosedur Utama.....38
4.2 Penjelasan Tata Cara.....39
4.3 Hasil Pengujian.....41
4.3.1 Pengujian 1.....41
4.3.2 Pengujian 2.....51
4.3.3 Pengujian 3.....61
4.3.4 Pengujian 4.....71
4.4 Analisis Hasil Pengujian.....81
4.4.1 Test 1: Query = Python, React, SQL.....81
4.4.2 Test 2: Query = a, merchandise.....81
4.4.3 Test 3: Query = Local Exchange, String.....82
4.4.4 Test 4: Query = compassionatel.....82
BAB V KESIMPULAN, SARAN, DAN REFLEKSI.....83

5.1 Kesimpulan..... 83
5.2 Saran..... 84
5.3 Refleksi..... 84
LAMPIRAN.....86
DAFTAR PUSTAKA..... 87

DAFTAR GAMBAR

Gambar 1. CV ATS dalam Dunia Kerja.....6
Gambar 2. Trie Node.....15
Gambar 3. Trie extended to automaton.....15

BAB I
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 II
LANDASAN TEORI

2.1 Knuth-Morris-Pratt (KMP)

KMP (Knuth-Morris-Pratt) adalah algoritma pencocokan string yang digunakan untuk mencari sebuah pola dalam sebuah teks. Misalnya, ketika ingin mencari apakah kata data muncul dalam paragraf tertentu, dan jika ya, pada posisi ke berapa. Algoritma ini akan menemukan semua posisi kemunculan pola P di dalam teks T, melakukannya dengan lebih efisien dibandingkan algoritma brute-force (yang membandingkan dari awal setiap kali terjadi mismatch).

Perbandingan dengan Brute Force:

Dalam algoritma brute force, pencocokan dilakukan dari kiri ke kanan, tapi saat terjadi mismatch, pola langsung digeser satu posisi ke kanan dan pencocokan dimulai lagi dari awal. Hal ini menyebabkan banyak perbandingan yang berulang.

Misalkan:

```
Teks   : A B A B A B C D
Pola   : A B A B C
```

Jika kita cocokkan dari awal, kita akan cocok sampai karakter ke-4, tapi pada karakter ke-5 ($C \neq B$), brute force akan menggeser satu posisi dan mulai lagi dari awal. Padahal, sebagian pola sudah cocok dan informasi itu seharusnya bisa dimanfaatkan.

KMP menghindari perbandingan yang tidak perlu dengan memanfaatkan informasi pola yang sudah dicocokkan sebelumnya. Jika terjadi mismatch, KMP tidak menggeser pola satu per satu, tetapi menggunakan fungsi bantu bernama border function (atau failure function) untuk menentukan posisi terbaik untuk memulai ulang pencocokan. Artinya, KMP tahu bahwa sebagian pola yang sudah cocok tidak perlu diperiksa lagi karena tidak mungkin cocok ulang.

Failure function (sering dilambangkan b[] atau f[]) menyimpan informasi tentang panjang prefix terpanjang dari P[0..k] yang juga merupakan suffix dari P[1..k].Tujuannya apabila terjadi

mismatch di posisi j, failure function akan memberitahu ke posisi mana dalam pola P kita bisa lompat tanpa perlu mencocokkan ulang karakter yang sudah cocok.

Pseudocode untuk failure function untuk KMP sendiri adalah sebagai berikut:

```
function computeFailureFunction(P):
    m = length(P)
    b = array of size m
    b[0] = -1
    j = -1

    for i from 1 to m - 1:
        while j >= 0 and P[j + 1] ≠ P[i]:
            j = b[j]
        if P[j + 1] == P[i]:
            j = j + 1
        b[i] = j

    return b
```

Intinya jika terjadi mismatch pada karakter ke-j di pola dan ke-i di teks (T[i] ≠ P[j]). Maka KMP akan:

1. Tidak menggeser satu karakter (seperti brute force),
2. Tapi mencari prefix terpanjang dari P[0 .. j-1] yang juga merupakan suffix dari P[1 .. j-1], Lalu mengatur kembali indeks j ke posisi baru, sehingga karakter yang sudah dibandingkan tidak perlu dibandingkan ulang.

KMP sudah tahu mana yang pasti tidak cocok, jadi jangan diperiksa lagi.

Pseudocode KMP untuk mencari pola P di teks T dengan memanfaatkan tabel LPS tadi agar pencarian jadi lebih efisien:

```
function KMP_Search(text T, pattern P)
    n ← length of T
    m ← length of P

    // Step 1: Preprocess pattern to compute LPS array
    LPS ← compute_LPS(P)

    // Step 2: Search for pattern in text
    i ← 0     // index for T
    j ← 0     // index for P

    while i < n do
        if T[i] == P[j] then
```

```
            i ← i + 1
            j ← j + 1

            if j == m then
                print("Pattern found at index", i - j)
                j ← LPS[j - 1]   // continue searching
        else
            if j != 0 then
                j ← LPS[j - 1]
            else
                i ← i + 1
```

Pseudocode Compute_LPS untuk menghitung dan buat tabel LPS (Longest Prefix Suffix) untuk pola P:

```
function compute_LPS(P)
    m ← length of P
    LPS[0] ← 0
    len ← 0         // length of previous longest prefix suffix
    i ← 1

    while i < m do
        if P[i] == P[len] then
            len ← len + 1
            LPS[i] ← len
            i ← i + 1
        else
            if len != 0 then
                len ← LPS[len - 1]
            else
                LPS[i] ← 0
                i ← i + 1

    return LPS
```

Algoritma KMP sangat efisien karena saat mencari pola dalam teks, ia hanya bergerak maju tanpa pernah mundur, sehingga cocok digunakan untuk memproses data besar seperti file besar atau data streaming dari jaringan. Kelebihan ini membuat KMP lebih cepat daripada metode pencarian biasa yang harus kembali ke awal pola setiap kali terjadi ketidakcocokan. Namun, KMP kurang efektif jika jumlah karakter yang mungkin muncul (alfabet) sangat banyak karena peluang terjadinya ketidakcocokan lebih besar, terutama jika ketidakcocokan sering muncul di awal pola. Dalam situasi seperti itu, KMP tidak bisa banyak membantu dan kecepatannya tidak jauh berbeda dengan cara pencarian biasa. Sebaliknya, KMP bekerja paling baik ketika ketidakcocokan terjadi setelah beberapa karakter pola berhasil dicocokkan, sehingga ia dapat melompat jauh dan menghindari pemeriksaan yang sia-sia.

2.2 Boyer-Moore (BM)

Boyer-Moore adalah salah satu algoritma pencocokan pola paling cepat yang digunakan untuk mencari pola (pattern) dalam teks (text). Keunggulan utamanya terletak pada cara algoritma ini melakukan perbandingan dan pergeseran pola ketika terjadi ketidakcocokan (mismatch). Boyer-Moore sendiri memanfaatkan 2 teknik utama:

1. Teknik Looking-Glass (Cermin)

Pencocokan pola dilakukan dari belakang ke depan, yaitu dimulai dari karakter terakhir dalam pola dan bergerak mundur ke karakter pertama. Ini berbeda dari pendekatan biasa yang mencocokkan dari kiri ke kanan. Teknik ini memungkinkan algoritma segera mendeteksi ketidakcocokan pada bagian yang paling akhir dari pola terlebih dahulu, sehingga bisa langsung mengambil tindakan efisien tanpa memeriksa seluruh pola jika tidak perlu.

2. Teknik Character-Jump (Lompatan Karakter)

Teknik yang digunakan saat terjadi mismatch, yaitu ketika karakter teks T[i] tidak cocok dengan karakter pola P[j]. Dalam hal ini, algoritma melihat karakter x = T[i] yang menyebabkan mismatch, lalu memeriksa apakah karakter x muncul di bagian kiri pola (sebelum posisi j). Jika x muncul, pola digeser sehingga kemunculan terakhir x di pola sejajar dengan posisi i di teks. Jika x tidak muncul di pola sama sekali, maka pola digeser sepenuhnya melewati posisi i. Kedua teknik ini bekerja sama untuk memungkinkan algoritma Boyer-Moore melompati banyak karakter dan menjadi salah satu algoritma pencocokan string tercepat dalam praktik.

Untuk mendukung efisiensi dari teknik Character-Jump, algoritma Boyer-Moore menggunakan Last Occurrence Function (L(x)), yaitu sebuah fungsi yang memetakan setiap karakter dalam alfabet ke indeks kemunculan terakhirnya di pola. Fungsi ini digunakan untuk menentukan seberapa jauh pola dapat digeser saat terjadi mismatch. Jika karakter x = T[i] muncul dalam pola, maka pola digeser sehingga karakter x di pola berada tepat di bawah karakter x di teks. Jika tidak ada, maka pola digeser melewati posisi i, karena tidak ada kemungkinan karakter tersebut bisa cocok dengan bagian manapun dari pola.

Sebagai contoh, misalkan pola P = "abac" dan teks T = "zzabazac". Pencocokan dimulai dari karakter terakhir pola, yaitu 'c', dibandingkan dengan karakter pada teks. Jika terjadi mismatch di T[i] = z, dan huruf 'z' tidak terdapat dalam pola, maka pola bisa langsung digeser melewati posisi tersebut karena tidak mungkin ada kecocokan di posisi itu. Namun, jika mismatch terjadi dengan karakter yang ada di pola, seperti 'a', maka pola hanya digeser sejauh agar 'a' terakhir dalam pola sejajar dengan 'a' di teks.

Pseudocode Last Occurence Function:

```
function buildLastOccurrenceTable(pattern):
  // Create an array to store the last occurrence of each character in the
pattern
  lastOccurrence = new array(size of the alphabet)  // e.g., 256 for ASCII

  // Initialize all elements of the array to -1 (indicating no occurrence)
  for i from 0 to size of the alphabet - 1:
    lastOccurrence[i] = -1

  // Iterate through the pattern and update the last occurrence array
  for i from 0 to length of pattern - 1:
    character = pattern[i]
    lastOccurrence[character] = i  // Store the last position of this
character

  return lastOccurrence
```

Dengan kedua teknik ini dan dukungan Last Occurrence Function, algoritma Boyer-Moore mampu menghindari banyak perbandingan yang tidak perlu, menjadikannya sangat cepat dalam praktik, terutama saat pola dan teks memiliki ukuran besar. Kecepatan ini semakin terasa saat pola jarang cocok atau hanya muncul beberapa kali di dalam teks panjang, karena Boyer-Moore bisa melompat lebih jauh dibandingkan algoritma lain seperti KMP atau brute-force.

Selain Character-Jump, algoritma Boyer-Moore juga mendukung heuristic lain yang lebih kompleks, yaitu Good Suffix Heuristic. Teknik ini digunakan saat terdapat bagian akhir dari pola (suffix) yang cocok sebelum terjadi mismatch. Dengan heuristic ini, pola bisa digeser berdasarkan kesamaan dengan bagian lain dari pola yang menyerupai suffix tersebut. Meskipun lebih kompleks untuk diimplementasikan, heuristic ini semakin meningkatkan efisiensi Boyer-Moore, khususnya saat pola memiliki banyak pengulangan internal.

Pseudocode Booyer-Moore:

```
function boyerMooreSearch(text, pattern):
    n = length(text)
    m = length(pattern)

    // Handle edge cases
    if m == 0:
        return []
    if m > n:
        return []

    // Build last occurrence table
    lastOcc = buildLastOccurrenceTable(pattern)

    // Array untuk menyimpan semua posisi yang ditemukan
    matches = []

    // Mulai pencarian
    i = 0  // posisi awal pola di teks

    while i <= n - m:
        j = m - 1  // mulai dari karakter terakhir pola

        // Cocokkan karakter dari belakang ke depan
        while j >= 0 and pattern[j] == text[i + j]:
            j = j - 1

        // Jika j < 0, berarti semua karakter cocok
        if j < 0:
            matches.append(i)  // pola ditemukan di posisi i
            i = i + 1  // geser 1 posisi untuk mencari kemunculan berikutnya
        else:
            // Terjadi mismatch di posisi j
            mismatchChar = text[i + j]

            // Hitung pergeseran menggunakan character-jump heuristic
            if lastOcc[mismatchChar] < j:
                shift = j - lastOcc[mismatchChar]
            else:
                shift = 1  // minimal geser 1 posisi

            i = i + shift

    return matches
```

2.3 Aho-Corasick

Algoritma Aho-Corasick adalah algoritma pencocokan string yang dirancang khusus untuk mencari multiple patterns sekaligus dalam satu teks. Berbeda dengan Boyer-Moore yang

hanya mencari satu pola, Aho-Corasick dapat mencari ratusan atau ribuan pola secara bersamaan dengan efisien.

Struktur Data Aho-Corasick:

1. **Trie (Prefix Tree)**

Semua pola disimpan dalam struktur trie untuk memudahkan pencarian dan berbagi prefix yang sama.

2. **Failure Function (Failure Links)**

Ketika terjadi mismatch, failure function menentukan ke node mana harus "mundur" tanpa kehilangan informasi yang sudah diproses.

3. **Output Function**

Menyimpan informasi pola mana saja yang berakhir di setiap node.

Misalkan n adalah panjang teks dan m adalah total jumlah karakter dalam semua kata, yaitu m = length(arr[0]) + length(arr[1]) + ... + length(arr[k-1]). Di sini k adalah total jumlah kata input. Jika kita menggunakan algoritma pencarian waktu linear seperti KMP, maka kita perlu mencari satu per satu semua kata dalam text[]. Ini memberikan kompleksitas waktu total sebagai O(n + length(word[0])) + O(n + length(word[1])) + O(n + length(word[2])) + ... + O(n + length(word[k-1])). Kompleksitas waktu ini dapat ditulis sebagai O(n*k + m).

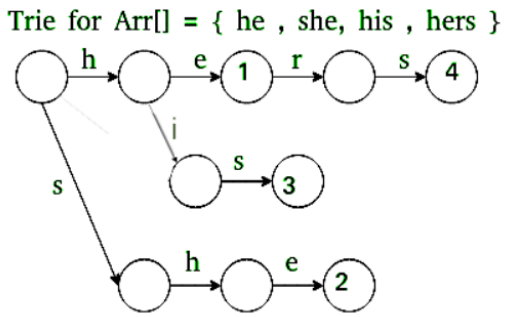
Algoritma Aho-Corasick menemukan semua kata dalam waktu O(n + m + z) di mana z adalah total jumlah kemunculan kata dalam teks. Algoritma pencocokan string Aho-Corasick membentuk dasar dari perintah Unix asli fgrep.

Gambaran Umum Algoritma:

Proses Konstruksi Detail

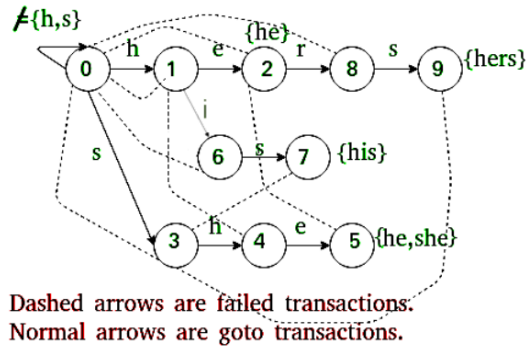
Fase 1: Membangun Trie (Fungsi Go To)

- Pertama kita membangun **Trie** (atau Keyword Tree) dari semua kata.



Gambar 2. *Trie Node*
(Sumber: https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/)

- Bagian ini mengisi entri dalam goto g[][] dan output o[].
- Untuk semua karakter yang tidak memiliki edge di root, kita menambahkan edge kembali ke root



Gambar 3. Trie extended to automaton
(Sumber: https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/)

Fase 2: Membangun Fungsi Failure

- Untuk state s, kita menemukan suffix proper terpanjang yang juga merupakan prefix proper dari beberapa pola.
- Ini dilakukan menggunakan Breadth First Traversal dari Trie.
- Bagian ini mengisi entri dalam failure f[]

Fase 3: Membangun Fungsi Output

- Untuk state s, indeks dari semua kata yang berakhir di s disimpan.
- Indeks ini disimpan sebagai bitwise map (dengan melakukan bitwise OR dari nilai-nilai).
- Ini dihitung menggunakan Breadth First Traversal dengan fungsi Failure.
- Ini memperluas array output o[].

Pseudocode Trie:

```
function BuildTrie(patterns):
    root ← new TrieNode()
    for each pattern in patterns:
        currentNode ← root
        for each character c in pattern:
            if currentNode.child[c] does not exist:
                currentNode.child[c] ← new TrieNode()
            currentNode ← currentNode.child[c]
        currentNode.output.add(pattern)  // Tandai akhir pola
    return root
```

Pseudocode Fungsi Failure:

```
function BuildFailureLinks(root):
    queue ← empty queue
    for each childNode of root:
        childNode.failure ← root
        enqueue(queue, childNode)

    while queue not empty:
        currentNode ← dequeue(queue)
        for each character c in alphabet:
            child ← currentNode.child[c]
            if child exists:
                failNode ← currentNode.failure
                while failNode != null and failNode.child[c] == null:
                    failNode ← failNode.failure
                if failNode == null:
                    child.failure ← root
                else:
                    child.failure ← failNode.child[c]
                    child.output ← child.output ∪ child.failure.output
                enqueue(queue, child)
```

Pseudocode Main Search:

```
function AhoCorasickSearch(root, text):
    currentNode ← root
    for i in 0 to length(text)-1:
        c ← text[i]
        while currentNode != root and currentNode.child[c] == null:
            currentNode ← currentNode.failure
```

```
        if currentNode.child[c] != null:
            currentNode ← currentNode.child[c]
        // Jika ada output pada node ini, berarti pola ditemukan
        if currentNode.output not empty:
            for pattern in currentNode.output:
                print("Pola ditemukan:", pattern, "di posisi", i -
length(pattern) + 1)
```

2.4 Fuzzy String Matching (Approximate Matching)

Fuzzy matching bertujuan untuk menemukan kecocokan pola dalam teks dengan memperbolehkan sejumlah kesalahan (insertions, deletions, substitutions). Hal ini berguna misalnya untuk pencarian teks dengan typo atau variasi kecil.

```
Misal kita punya:

Teks: T dengan panjang n

Pola: P dengan panjang m

Maksimal toleransi kesalahan: k

Tujuan: cari semua posisi di T dimana substring sepanjang m memiliki jarak
edit (edit distance) ≤ k terhadap P.
```

Pseudocode Fuzzy Matching:

```
function FuzzySearch(T, P, k):
    n ← length(T)
    m ← length(P)

    // Buat matriks DP dengan ukuran (m+1) x (n+1)
    // dp[i][j] = edit distance antara P[0..i-1] dan T[j-i..j-1]
    // untuk i ≤ j ≤ n dan i ≤ m

    for j in 0 to n:
        dp[0][j] ← 0  // Kosongkan pola cocok di posisi mana saja

    for i in 1 to m:
        dp[i][0] ← i  // Insert i karakter jika teks kosong

    // Hitung edit distance untuk setiap substring teks sepanjang m
    for j in 1 to n:
        for i in 1 to min(m, j):
            if P[i-1] == T[j-1]:
                cost ← 0
            else:
                cost ← 1
```

```
            dp[i][j] ← minimum(
                dp[i-1][j] + 1,      // Deletion
                dp[i][j-1] + 1,      // Insertion
                dp[i-1][j-1] + cost  // Substitution or match
            )

        // Setelah isi dp kolom j, cek apakah substring P cocok di posisi (j -
m)
        if dp[m][j] ≤ k:
            print("Approximate match found at index", j - m)
```

2.5 Levenshtein Distance

Levenshtein Distance adalah algoritma untuk menghitung jumlah minimum operasi edit yang dibutuhkan untuk mengubah satu string menjadi string lain.
Operasi yang diperbolehkan:

- Insert (menyisipkan karakter)
- Delete (menghapus karakter)
- Substitute (mengganti karakter)

Misal:

- String A: "kitten"
- String B: "sitting”

Langkah transformasi:

- kitten → sitten (substitusi 'k' → 's')
- sitten → sittin (substitusi 'e' → 'i')
- sittin → sitting (tambah 'g')

Maka Levenshtein Distance = 3

2.6 Regular Expression (Regex)

Regular Expression (Regex) adalah pola (pattern) yang digunakan untuk mencari dan mengekstrak bagian teks tertentu berdasarkan aturan tertentu. Regex sering dipakai dalam pemrosesan string, validasi input, dan ekstraksi informasi.

Contoh penggunaan regex dalam kehidupan sehari-hari:

1. Pencari semua email dalam teks.
2. Pencarian tanggal kelulusan.
3. Ekstrak semua nama universitas.

```
function isMatchDP(text, pattern):
    m = length(text)
    n = length(pattern)

    // dp[i][j] = true jika text[0..i-1] cocok dengan pattern[0..j-1]
    dp = array[m+1][n+1] initialized to false

    // Base case: empty pattern cocok dengan empty text
    dp[0][0] = true

    // Handle patterns seperti a*, a*b*, a*b*c* yang bisa match empty string
    for j from 2 to n:
        if pattern[j-1] == '*':
            dp[0][j] = dp[0][j-2]

    // Fill DP table
    for i from 1 to m:
        for j from 1 to n:
            if pattern[j-1] == '*':
                // '*' bisa match 0 atau lebih karakter sebelumnya
                dp[i][j] = dp[i][j-2]  // match 0 karakter

                if pattern[j-2] == text[i-1] or pattern[j-2] == '.':
                    dp[i][j] = dp[i][j] or dp[i-1][j]  // match 1+ karakter

            else if pattern[j-1] == '.' or pattern[j-1] == text[i-1]:
                dp[i][j] = dp[i-1][j-1]

    return dp[m][n]
```

BAB III
ANALISIS PEMECAHAN MASALAH

3.1 Cara Kerja Program

Saat pertama kali membuka aplikasi CV Scanner pada layar utama, kita dapat mengetikkan kata kunci yang ingin kita cari untuk dilakukan *string matching* (misalnya "Python, Excel, marketing") di area input. Kita juga dapat memilih algoritma pencarian mana yang ingin digunakan baik KMP, Boyer-Moore yang lebih cepat untuk pattern yang panjang, atau Aho-Corasick yang bagus untuk mencari banyak kata kunci sekaligus. Selain itu, kita dapat menentukan berapa banyak hasil teratas yang ingin ditampilkan, mulai dari hanya beberapa hingga puluhan CV.

Ketika kita menekan tombol "Search CVs", proses pencarian dimulai. Pertama-tama, aplikasi memastikan kita sudah memasukkan kata kunci. Kemudian, tombol pencarian berubah menjadi "Searching..." agar kita tahu sistem sedang bekerja. Di balik layar, aplikasi memanggil fungsi pencarian utama dengan parameter berupa jumlah hasil yang kita inginkan, algoritma yang kita pilih, dan kata kunci yang kita masukkan.

```
PS C:\Users\CLCC\TubesSTIMA3\src> python main.py


Searching with: algorithm=kmp, keywords=semogastimadapeta,
n=1000
```

Proses pencarian di backend cukup kompleks karena database terenkripsi dan harus di dekripsi. Pertama, aplikasi terhubung ke database MySQL untuk mengambil data semua pelamar dan detail CV mereka. Data pribadi pelamar dienkripsi dengan algoritma RSA untuk keamanan, jadi sistem perlu mendekripsi informasi ini terlebih dahulu. Kemudian, kata kunci yang kita masukkan dipisahkan menjadi daftar kata kunci individual, semuanya diubah menjadi huruf kecil agar pencarian tidak dibedakan berdasarkan kapitalisasi.

Aplikasi menggunakan teknik parallel processing untuk meningkatkan kecepatan pencarian. Untuk setiap CV dalam database, sistem mengekstrak teks dari file PDF, lalu menerapkan algoritma pencarian yang kita pilih (KMP, Boyer-Moore, atau Aho-Corasick) untuk mencari kata kunci kita. Sistem mencatat berapa kali setiap kata kunci muncul dalam setiap CV.

CV Scanner tidak hanya mencari kecocokan kata yang persis sama, tetapi juga menerapkan pencarian fuzzy menggunakan algoritma Levenshtein Distance. Artinya, kalau ada typo kecil atau variasi penulisan, sistem masih bisa mengenali kata tersebut. Misalnya, jika kita mencari "JavaScript" tapi di CV tertulis "Javascrpt" atau "JavaScrip", sistem masih bisa mendeteksinya sebagai kecocokan potensial. Namun, algoritma ini hanya diterapkan apabila terdapat kata kunci yang tidak memiliki *match,* algoritma ini adalah algoritma *fallback.*

Setelah selesai mencari, sistem mengurutkan CV berdasarkan jumlah total kemunculan kata kunci yang kita cari. Hanya CV yang memiliki minimal satu kecocokan yang akan ditampilkan. Untuk CV teratas sesuai dengan jumlah yang kita tentukan, aplikasi melakukan ekstraksi informasi tambahan untuk membuat ringkasan CV yang lebih komprehensif. Hasil pencarian kemudian dikembalikan ke antarmuka pengguna dalam bentuk daftar CV teratas, lengkap dengan informasi waktu eksekusi pencarian dan total jumlah CV yang telah dipindai.

3.2 Proses Pemecahan Masalah

3.2.1 KMP (Knuth Morris Pratt)

Untuk algoritma KMP sendiri, memiliki 3 fungsi yang memiliki kepentingannya yang berbeda-beda yaitu sebagai berikut:

Fungsi utama KMP akan melakukan inisialisasi result dengan langkah berikut:

```
results = []
    # Iterasi setiap kata kunci (mulai dari depan)dan terapkan
algoritma KMP
    for keyword in keywords:
        count = kmp_search(text, keyword)
        results.append(count)  # Simpan jumlah kemunculan setiap
kata kunci
```

Fungsi ini berfungsi sebagai entry point untuk algoritma KMP:

- Menerima teks CV dan daftar kata kunci yang ingin dicari
- Mengubah setiap kata kunci menjadi huruf kecil untuk pencarian yang tidak sensitif terhadap kapitalisasi
- Memanggil fungsi kmp_search untuk setiap kata kunci untuk mendapatkan jumlah kemunculan
- Mengembalikan list berisi jumlah kemunculan untuk setiap kata kunci

```
def kmp_search(text: str, pattern: str) -> int:

"""Menghitung kemunculan pattern dalam text menggunakan algoritma
KMP"""
    n = len(text)
    m = len(pattern)

    # Kasus khusus: pattern kosong
    if m == 0:
        return 0

    # Dapatkan tabel border
    b = compute_border(pattern)
    count = 0
    i = j = 0  # i untuk indeks text, j untuk indeks pattern
```

Fungsi ini berfungsi sebagai entry point untuk algoritma KMP:

- Menerima teks CV dan daftar kata kunci yang ingin dicari
- Mengubah setiap kata kunci menjadi huruf kecil untuk pencarian yang tidak sensitif terhadap kapitalisasi
- Memanggil fungsi kmp_search untuk setiap kata kunci untuk mendapatkan jumlah kemunculan
- Mengembalikan list berisi jumlah kemunculan untuk setiap kata kunci

2. Fungsi KMP Search kmp_search()

Fungsi kmp_search adalah fungsi algortima ini yang akan melakukan pencarian pattern dalam text menggunakan algoritma KMP:

- Pertama memperoleh tabel border dengan memanggil compute_border()
- Menelusuri teks secara linear sekali jalan (O(n))
- Menggunakan indeks i untuk text dan j untuk pattern
- Mencocokkan karakter dan memajukan kedua pointer saat karakter cocok
- Saat pattern ditemukan lengkap (j == m), menambah counter dan melakukan lompatan cerdas

- Saat terjadi ketidakcocokan, menggunakan tabel border untuk melompat ke posisi yang tepat alih-alih mundur ke awal pattern
- Menghitung total kemunculan pattern dalam text

Terdapat 3 kasus utama dalam kasus string matching sendiri yaitu:

1. Kasus 1: Karakter cocok

```
if text[i] == pattern[j]:
    i += 1
    j += 1
```

Kasus ini akan mengecek kondisi if text[i] == pattern[j] yang artinya membandingkan karakter saat ini dari teks (yang ditunjuk oleh indeks i) dengan karakter dari pola/pattern (yang ditunjuk oleh indeks j). Lalu akan memajukan pointer teks jika karakternya cocok, maka i ditambah 1 untuk bergerak ke karakter berikutnya dalam teks dan memajukan pointer pattern. Bersamaan dengan itu, j juga ditambah 1 untuk bergerak ke karakter berikutnya dalam pattern.

2. Kasus 2 ketika pattern ditemukan sudah lengkap:

- Kondisi if j == m mengecek apakah kita telah mencapai akhir pattern (indeks j sama dengan panjang pattern m) Jika ya, berarti kita telah menemukan satu kecocokan lengkap pattern dalam teks.

- count += 1 menambah penghitung kecocokan, yang nantinya akan menjadi hasil akhir dari algoritma.
- Lompatan dalam KMP:

  Alih-alih kembali ke awal pattern dan memulai pencarian baru dari karakter berikutnya di teks KMP melakukan lompatan. j = b[j - 1] mengatur ulang pointer pattern (j) menggunakan nilai dari tabel border. Nilai ini menunjukkan posisi dalam pattern dimana kita bisa langsung melanjutkan pencarian tanpa harus mengulang perbandingan karakter yang sudah kita ketahui cocok.

  Contoh Konkret:

  Text: "ABAB**ABAB**"
  Pattern: "**ABAB**"
  Tabel border: [0,0,1,2]

Saat j = 4 (pattern "ABAB" cocok), kita tidak mulai dari j = 0 lagi. Kita lihat b[3] = 2, berarti kita lanjut dari indeks ke-2 di pattern ("ABAB"), karena dua karakter terakhir "AB" juga merupakan dua karakter awal pattern.

3. Kasus 3: Ketidakcocokan

```
elif i < n and text[i] != pattern[j]:
    # KUNCI KMP: Hindari mengulang perbandingan
    if j != 0:
        j = b[j - 1]  # Lompat ke state sebelumnya
    else:
        i += 1  # Move to next character in text
```

Kasus 3 adalah saat terjadi ketidakcocokan antara karakter di teks dan di pattern. Ini adalah bagian yang paling krusial dalam algoritma KMP yang membedakannya dengan algoritma brute force dan lainnya. elif i < n and text[i] != pattern[j] memeriksa apakah kita belum mencapai akhir teks DAN karakter saat ini di teks tidak cocok dengan karakter saat ini di pattern.

Penanganan Ketidakcocokan:

- Jika j != 0 (kita sudah mencocokkan setidaknya satu karakter sebelumnya), j = b[j - 1] adalah inti kecerdasan KMP. Kita tidak kembali ke awal pattern (j = 0), tetapi "melompat" ke posisi yang ditunjukkan oleh tabel border. Lompatan ini memungkinkan kita melewati perbandingan yang pasti gagal berdasarkan informasi yang kita miliki dari kecocokan sebelumnya
- Jika j == 0 (kita di awal pattern), i += 1 kita hanya bergerak ke karakter berikutnya di teks, karena kita sudah di awal pattern, tidak ada lompatan yang bisa dilakukan

Contoh Konkret:

```
Misalkan:
Teks: "ABACABABC"
Pattern: "ABAC"
Tabel border: [0,0,1,0]
```

Proses:

```
Mulai: i=0, j=0
"A" cocok dengan "A": i=1, j=1
"B" cocok dengan "B": i=2, j=2
"A" cocok dengan "A": i=3, j=3
"C" cocok dengan "C": i=4, j=4 -> Pattern lengkap ditemukan!
count=1, j=b[3]=0
Mulai lagi: i=4, j=0
"A" cocok dengan "A": i=5, j=1
"B" cocok dengan "B": i=6, j=2
"A" cocok dengan "A": i=7, j=3
"B" tidak cocok dengan "C" -> Kasus 3 terjadi!
j != 0, jadi j = b[j-1] = b[2] = 1
Artinya kita loncat ke karakter kedua pattern ("B")
"B" tidak cocok dengan "A" -> Kasus 3 terjadi lagi!
j != 0, jadi j = b[j-1] = b[0] = 0
"B" tidak cocok dengan "A" -> Kasus 3, dengan j = 0
j == 0, jadi i += 1 (bergerak ke karakter berikutnya di teks)
Lanjutkan proses...
```

Keunggulan KMP terlihat jelas pada langkah 10-11. Alih-alih kembali ke awal pattern dan mulai membandingkan dari karakter berikutnya di teks, KMP loncat ke posisi yang tepat

dalam pattern berdasarkan kecocokan sebelumnya. Ini menghemat banyak operasi perbandingan karakter, terutama untuk teks dan pattern yang panjang.

Terakhir fungsi compute_border yang merupakan inti dari kecerdasan algoritma KMP. Fungsi ini membangun tabel border function yang memungkinkan algoritma melakukan lompatan cerdas ketika terjadi ketidakcocokan. Border function, atau sering disebut "failure function", adalah tabel yang menyimpan informasi untuk setiap posisi dalam pattern:

- Nilai b[i]: panjang maksimum dari prefix pattern[0...i] yang juga merupakan suffix dari pattern[0...i].
- Dengan kata lain, jika sudah mencocokkan i+1 karakter dan kemudian terjadi ketidakcocokan, berapa banyak karakter dari awal pattern yang sudah pasti cocok dengan akhir dari segmen yang sudah dicocokkan.

Langkah-langkah Fungsi:

1. Inisialisasi:

- m = len(pattern) - Mendapatkan panjang pattern
- b = [0 for _ in range(m)] - Membuat array b dengan panjang m, semua nilai awal 0.
- length = 0 - Variabel untuk melacak panjang border saat ini.
- i = 1 - Mulai dari karakter kedua (indeks 1), karena b[0] selalu 0

2. Iterasi melalui pattern: Untuk setiap karakter mulai dari indeks 1:

   Kasus 1: Karakter saat ini cocok dengan karakter di posisi 'length':

```
if pattern[i] == pattern[length]:
    length += 1  # Perpanjang border saat ini
    b[i] = length  # Simpan panjang border
    i += 1  # Lanjut ke karakter berikutnya
```

   Kasus 2: Ketidakcocokan terjadi:

```
else:
    if length != 0:
        length = b[length - 1]  # Lompat ke border sebelumnya
```

```
    else:
        b[i] = 0  # Tidak ada border yang cocok
        i += 1  # Lanjut ke karakter berikutnya
```

Contoh Konkret Pembentukan Tabel Border:

```
Pattern: A B A B C A B A B
Indeks:  0 1 2 3 4 5 6 7 8
```

```
Langkah-langkah:

i=1, length=0: 'B' ≠ 'A', length=0, b[1]=0, i=2
i=2, length=0: 'A' = 'A', length=1, b[2]=1, i=3
i=3, length=1: 'B' = 'B', length=2, b[3]=2, i=4
i=4, length=2: 'C' ≠ 'A', length=b[1]=0
i=4, length=0: 'C' ≠ 'A', length=0, b[4]=0, i=5
i=5, length=0: 'A' = 'A', length=1, b[5]=1, i=6
i=6, length=1: 'B' = 'B', length=2, b[6]=2, i=7
i=7, length=2: 'A' = 'A', length=3, b[7]=3, i=8
i=8, length=3: 'B' = 'B', length=4, b[8]=4
Tabel border akhir: [0,0,1,2,0,1,2,3,4]
```

```
def compute_border(pattern: str) -> List[int]:
    """Menghitung tabel border function (panjang prefix yang juga
suffix)"""
    m = len(pattern)
    b = [0] * m  # Inisialisasi tabel border

    length = 0  # Panjang border saat ini
    i = 1       # Mulai dari karakter kedua

    while i < m:
        if pattern[i] == pattern[length]:
            # Kasus 1: Karakter cocok - perpanjang border
            length += 1
            b[i] = length
            i += 1
        else:
            # Kasus 2: Ketidakcocokan
            if length != 0:
                # Backtracking ke border sebelumnya
                length = b[length - 1]
            else:
                # Tidak ada border yang cocok
```

```
                b[i] = 0
                i += 1
    return b
```

3.2.2 Boyer-Moore

Pertama, fungsi utama boyer_moore akan memproses setiap keyword di dalam daftar. Untuk setiap keyword, fungsi akan mengubahnya menjadi huruf kecil. Tujuannya agar pencarian tidak membedakan huruf besar atau kecil.

Selanjutnya, proses pencocokan dimulai. Pengecekan dilakukan dari karakter paling kanan pada pola (pattern). Jika terjadi ketidakcocokan, algoritma akan melakukan pergeseran berdasarkan blok kode pada fungsi bm_single_match.

```
# Blok kode utama untuk menangani semua kasus ketidakcocokan
else:
   lo = last[ord(text[i])]

   i = i + m - min(j, 1 + lo)
   j = m - 1
```

Berikut adalah penanganan untuk tiga kasus utama saat terjadi ketidakcocokan:

1. Jika karakter ada di tabel last_occurrence: Algoritma akan melihat di mana posisi terakhir karakter tersebut di dalam pola. Kemudian, pola digeser ke kanan agar karakter yang sama di dalam pola tersebut menjadi sejajar dengan karakter yang tidak cocok di text. Tujuannya adalah untuk menempatkan kandidat karakter yang berpotensi cocok ke posisi yang benar secepat mungkin.
    - Kode: Penanganan ini terjadi saat lo = last[ord(text[i])] berhasil menemukan indeks karakter (lo >= 0), dan i = i + m - min(j, 1 + lo) menggunakan nilai lo tersebut untuk menghitung pergeseran yang tepat.

2. Jika karakter tidak ada di tabel last_occurrence: Jika karakter dari text yang tidak cocok itu sama sekali tidak ada di dalam pola, maka algoritma akan menggeser seluruh panjang pola melewati posisi saat ini. Hal ini dilakukan karena jika karakter tersebut saja tidak ada, maka tidak ada gunanya melakukan pergeseran kecil.

- Kode: Kasus ini terjadi ketika lo mendapatkan nilai -1 (nilai default saat inisialisasi tabel). Rumus pergeseran i = i + m - min(j, 1 + lo) secara otomatis akan menjadi i = i + m, menyebabkan pergeseran penuh.

3. Jika posisi terakhir karakter sudah terlewat: Ini terjadi jika posisi terakhir sebuah karakter di dalam pola (lo) berada di sebelah kanan dari titik pengecekan saat ini (j). Untuk mencegah pola bergeser mundur, algoritma memastikan bahwa pergeseran selalu bergerak maju setidaknya satu langkah. Ini menjamin pencarian tidak pernah mundur atau terjebak di tempat yang sama.

    - Kode: Fungsi min() dalam i = i + m - min(j, 1 + lo) adalah bagian yang menangani ini. Dengan memilih nilai terkecil antara j dan 1 + lo, ia menjamin bahwa pergeseran tidak akan menyebabkan pola mundur.

Berikut adalah contoh konkret untuk algoritma Boyer-Moore

Skenario :

```
Teks: ABABD

Pola: ABD
```

Tahap 0 : Membuat tabel last_occurence

Karakter	Indeks terakhir
A	0
B	1
D	2

Tahap 1: Penjajaran Awal & Ketidakcocokan

```
Teks: A B A B D
Pola: A B D
          ^
```

- Bandingkan: Karakter di Teks A (indeks 2) dengan karakter di Pola D (indeks 2).
- Hasil: Tidak Cocok

Tahap 2 : Menghitung Lompatan

1. Identifikasi Bad Character: Karakter di Teks yang menyebabkan kegagalan adalah 'A'.
2. Cek Tabel last: Di mana posisi terakhir 'A' di dalam pola ABD? Tabel menunjukkan posisinya di indeks 0.
3. Hitung Pergeseran: Kita ingin menyelaraskan 'A' di teks dengan 'A' di pola. Untuk melakukan itu, kita perlu menggeser pola ke kanan.

```
Posisi 'A' di Teks: 2
Posisi 'A' di Pola: 0
Pergeseran yang dibutuhkan: 2 - 0 = 2 langkah
```

4. Tindakan: Geser pola sejauh 2 langkah ke kanan.

Tahap 3: Penjajaran Baru & Kecocokan

```
Teks: A B A B D
Pola:       A B D
            ^ ^ ^
```

1. Bandingkan Teks[4] ('D') vs Pola[2] ('D'): Cocok. Mundur ke kiri.
2. Bandingkan Teks[3] ('B') vs Pola[1] ('B'): Cocok. Mundur ke kiri.
3. Bandingkan Teks[2] ('A') vs Pola[0] ('A'): Cocok.

   Kecocokan Ditemukan

3.2.3 Aho-Corasick

Menjalankan Aho-Corasick dibagi menjadi 3 tahap utama yaitu :

1. Membangun Trie (build_trie)

   Pertama, semua kata kunci digabungkan ke dalam sebuah struktur data yang disebut Trie, atau pohon prefiks. Setiap simpul (node) di pohon ini merepresentasikan sebuah awalan (prefiks) dari satu atau lebih kata kunci. Ketika sebuah kata kunci selesai dimasukkan,

simpul terakhirnya akan ditandai sebagai output untuk menandakan akhir dari sebuah pola.

Kode : Fungsi build_trie melakukan tahap ini. Perulangan di dalamnya memasukkan setiap karakter dari setiap pattern ke dalam root Trie.

```
def build_trie(patterns: List[str]) -> TrieNode:
    root = TrieNode()
    for i, pattern in enumerate(patterns):
        node = root
        # Memasukkan setiap karakter dari pola ke dalam Trie
        for char in pattern:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        # Menandai akhir dari sebuah pola
        node.output.append(i)
    return root
```

2. Membangun Tautan Kegagalan / Failure Links (build_failure_links)

   Setiap simpul di dalam Trie diberikan sebuah tautan kegagalan (failure link). Tautan ini adalah sebuah "jalan pintas" yang akan digunakan saat terjadi ketidakcocokan. Tautan ini menunjuk ke simpul lain di dalam Trie yang merepresentasikan sufiks terpanjang dari jalur saat ini, yang juga merupakan prefiks dari kata kunci lain. Sederhananya, jika kita mengikuti "ABC" dan gagal di karakter berikutnya, tautan kegagalan mungkin akan membawa kita ke simpul "C", karena "C" bisa jadi merupakan awal dari kata kunci lain (misalnya, "CAT").

Kode: Fungsi build_failure_links mengatur semua jalan pintas ini. Ia menggunakan Breadth-First Search (BFS) untuk melintasi Trie dan menentukan tautan kegagalan untuk setiap simpul.

```
# Di dalam build_failure_links
while failure and char not in failure.children:
    failure = failure.failure # Mengikuti rantai kegagalan

# Menetapkan tautan kegagalan
if failure and char in failure.children:
    child.failure = failure.children[char]
else:
```

```
    child.failure = root
```

3. Mencari di dalam Teks (search)

   Dengan Trie dan tautan kegagalan yang sudah siap, algoritma akan melintasi text karakter per karakter sambil bergerak di dalam Trie.

- Jika karakter cocok: Ia akan maju ke simpul anak berikutnya. Jika simpul baru tersebut adalah output (akhir dari sebuah kata kunci), maka sebuah kecocokan akan dicatat. Berkat tautan kegagalan, satu simpul bisa mencatat kecocokan untuk beberapa pola sekaligus (misalnya, menemukan "hers" juga berarti menemukan "he").
- Jika karakter tidak cocok: Alih-alih memulai dari awal, ia akan mengikuti tautan kegagalan ke simpul lain dan mencoba mencocokkan karakter yang sama lagi dari keadaan baru tersebut. Ini memungkinkan algoritma untuk tidak kehilangan informasi dan melanjutkan pencarian dari posisi yang paling relevan.

Kode: Fungsi search melakukan proses ini. Blok while di dalamnya adalah inti dari mekanisme kegagalan, yang membedakannya dari pencarian Trie biasa.

```
# Di dalam fungsi search
# Jika terjadi ketidakcocokan, ikuti tautan kegagalan
while node and char not in node.children:
    node = node.failure

# Jika ada transisi yang valid (setelah mungkin mengikuti tautan kegagalan)
if node and char in node.children:
    node = node.children[char]
else:
    # Jika tidak ada jalan sama sekali, kembali ke root
    node = root
```

3.2.4 Regex

Proses ekstraksi menggunakan regex dapat dipecah menjadi beberapa tahap :

1. Ekstraksi Teks dari PDF

Sebelum analisis Regex dapat dilakukan, seluruh konten teks dari setiap halaman file PDF diambil dan digabungkan menjadi satu string tunggal untuk dianalisis. Hal ini, dilakukan menggunakan Fungsi extract_text_from_pdf

2. Analisis Teks dengan Pola Regex
   - Mencari Judul Pekerjaan

     Pola ini mengasumsikan judul pekerjaan utama (misalnya, "SOFTWARE ENGINEER") ditulis di bagian atas CV dengan huruf kapital semua.

```
job_title_pattern = r'^([A-Z][A-Z\s]+)(?=\n)'
job_title_match = re.search(job_title_pattern, text, re.MULTILINE)
```

   - Mengekstrak Bagian Umum (Skills, Summary, Highlights, dll.)

     Skrip menggunakan pola yang sangat mirip untuk menemukan judul bagian (seperti "Skills") dan kemudian menangkap semua teks setelahnya hingga menemukan judul bagian berikutnya (misalnya, "Experience") atau akhir dokumen

```
# Pola umum yang diadaptasi untuk Skills, Summary, dll.
skills_pattern = r'Skills\s*\n+(.*?)(?=\n\s*[A-Z][a-z]+\s*\n|$)'
skills_match = re.search(skills_pattern, text, re.DOTALL | re.IGNORECASE)
```

   - Mengekstrak Pengalaman Kerja

     Pertama, ia mengambil seluruh blok "Experience". Kedua, ia mencari entri pekerjaan yang terstruktur di dalam blok tersebut, dengan asumsi formatnya adalah Tanggal, Nama Perusahaan, dan Posisi.

```
experience_pattern = r'Experience\s*\n+(.*?)(?=\n\s*Education|$)'
# ...
work_entries = re.findall(r'(\d{2}/\d{4}\s*-\s*\d{2}/\d{4})\s*\n([^\n,]+)\s+City\s*,\s*State\s+([^\n]+)',
experience_text)
```

   - Mengekstrak Pendidikan

Pertama, seluruh bagian "Education" diekstrak. Kemudian, pola-pola Regex yang lebih spesifik dijalankan pada blok tersebut untuk menemukan nama sekolah, gelar, dan tahun kelulusan.

```
education_pattern = r'Education\s*\n+(.*?)(?=\n\s*Certifications|$)'
# ...
school_pattern = r'([A-Za-z\s]+(?:College|University))'
degree_pattern = r'(Associate|Bachelor|Master|PhD).*?:\s*([A-Za-z\s]+)'
year_pattern = r'\b(19|20)\d{2}\b'
```

3. Penyusunan Hasil Akhir

Setelah semua informasi diekstrak dan disimpan dalam sebuah dictionary extracted_info

```
# Contoh bagian penyusunan hasil
if 'Job_Title' in extracted_info:
    result += "JOB TITLE:\n"
    result += f"  {extracted_info['Job_Title']}\n\n"
```

3.2.5 RSA *Algorithm*

Untuk mengenkripsi data profil pelamar, digunakan RSA algorithm yang merupakan sistem kriptografi asimetris. Sistem ini menggunakan sepasang kunci: kunci publik untuk proses enkripsi dan kunci privat untuk proses dekripsi. Keamanannya didasarkan pada kesulitan matematis untuk memfaktorkan bilangan besar yang merupakan hasil perkalian dua bilangan prima.

Algoritma RSA diimplementasikan dalam sebuah *class* RSA yang kemudian akan digunakan oleh fungsi-fungsi lain dalam mengenkripsi data profil pelamar. *Class* ini bertanggung jawab untuk keseluruhan proses kriptografi, mulai dari validasi bilangan prima, pembuatan pasangan kunci publik dan privat secara otomatis berdasarkan nilai p dan q yang diberikan, hingga menyediakan metode inti untuk enkripsi teks (mengubah karakter menjadi angka terenkripsi) dan dekripsi (mengubah angka terenkripsi kembali menjadi teks)

```
class RSA:
    def __init__(self, p: int, q: int, e: int = None) -> None:
        """Initialize RSA with two prime numbers, p and q."""
```

```
        if not (self.is_prime(p) and self.is_prime(q)):
            raise ValueError("p and q must be prime numbers.")
        if p == q:
            raise ValueError("p and q cannot be the same.")

        self.p = p
        self.q = q
        self.n = p * q
        self.phi_n = (p - 1) * (q - 1)

        # Generate public and private keys
        if e is not None:
            self.public_key, self.private_key = self.generate_keys_with_e(e)
        else:
            self.public_key, self.private_key = self.generate_keys()
```

Bagian ini adalah konstruktor kelas RSA. Saat sebuah objek RSA dibuat, kode ini:

1. Memvalidasi Input: Memastikan bahwa p dan q adalah bilangan prima yang berbeda.
2. Menghitung Komponen Kunci: Menghitung dua nilai fundamental untuk RSA:
    - n = p * q: Modulus yang digunakan untuk enkripsi dan dekripsi.
    - phi_n = (p - 1) * (q - 1): *Euler's Totient Function*, yang penting untuk menghitung *private key*.
3. Membuat Kunci: Memanggil fungsi internal untuk menghasilkan pasangan *public* dan *private key*.

Selain itu, terdapat berbagai *helper functions* matematis yang diperlukan untuk RSA:

```
def is_prime(self, num: int) -> bool: ...
def gcd(self, a: int, b: int) -> int: ...
def extended_gcd(self, a: int, b: int) -> Tuple[int, int, int]: ...
def modular_inverse(self, a: int, m: int) -> int: ...
```

- is_prime(...) : Memeriksa apakah sebuah angka adalah bilangan prima.
- gcd(...) : Menghitung Faktor Persekutuan Terbesar (GCD) untuk menemukan e.
- extended_gcd(...) dan modular_inverse(...) : Dua fungsi yang bekerja sama untuk menghitung d (eksponen kunci privat) dengan mencari invers modular dari e.

Pembuatan kunci dari algoritma RSA memiliki implementasi dan fungsinya sendiri:

```
def generate_keys(self) -> Tuple[Tuple[int, int], Tuple[int, int]]:
        """Generate public and private key pairs."""
```

```
        e = random.randrange(2, self.phi_n)
        while self.gcd(e, self.phi_n) != 1:
            e = random.randrange(2, self.phi_n)
        d = self.modular_inverse(e, self.phi_n)
        return ((self.n, e), (self.n, d))

def generate_keys_with_e(self, e: int):
    if self.gcd(e, self.phi_n) != 1:
        raise ValueError("e must be coprime with phi_n")
    d = self.modular_inverse(e, self.phi_n)
    return ((self.n, e), (self.n, d))
```

- generate_keys(...) : Memilih eksponen kunci publik (e) secara acak, yang harus relatif prima terhadap phi_n. Kemudian, ia menghitung eksponen kunci privat (d).
- generate_keys_with_e(...) : Varian yang memungkinkan pengguna menentukan nilai e secara manual, yang berguna untuk konsistensi dan pengujian.

Proses dan fungsi untuk mengenkripsi dan mendekripsi teks:

```
def encrypt_text(self, text: str, public_key: Tuple[int, int]) ->
List[int]:
        n, e = public_key
        encrypted_chars = [pow(ord(char), e, n) for char in text]
        return encrypted_chars

def decrypt_text(self, encrypted_chars: List[int], private_key:
Tuple[int, int]) -> str:
       n, d = private_key
       decrypted_chars = [chr(pow(char, d, n)) for char in
encrypted_chars]
       return "".join(decrypted_chars)
```

- encrypt_text(...) : Mengambil sebuah string, mengubah setiap karakter menjadi nilai numeriknya (ASCII/Unicode ord()), lalu menerapkan rumus enkripsi RSA (ciphertext=plainnume(modn)) pada setiap angka. Hasilnya adalah sebuah list berisi angka-angka terenkripsi.
- decrypt_text(...) : Mengambil list angka terenkripsi, menerapkan rumus dekripsi RSA (plainnum=ciphertextd(modn)) pada setiap angka, lalu mengubah hasilnya kembali menjadi karakter (chr()) dan menggabungkannya menjadi string utuh.

Data yang dienkripsi akan disimpan pada sebuah *dictionary* (berisi *applicant profile*) sehingga dibentuk sebuah lapisan abstraksi untuk mengenkripsi dan mendekripsi data dalam *dictionary*:

```
def encrypt_data(data_dict: ..., rsa_instance: RSA) -> ...:
    # ... (code implementation) ...

def decrypt_data(encrypted_dict: ..., rsa_instance: RSA) -> ...:
    # ... (code implementation) ...
```

- encrypt_data(...) : Menerima sebuah dictionary, lalu mengiterasi setiap pasangan kunci-nilai. Jika nilainya adalah string, fungsi ini akan memanggil rsa_instance.encrypt_text untuk mengenkripsinya. Jika nilainya bukan string (misalnya applicant_id atau tanggal), nilai tersebut akan dibiarkan apa adanya.
- decrypt_data(...) : Melakukan hal sebaliknya. Fungsi ini mengasumsikan bahwa setiap nilai yang berupa list adalah data terenkripsi dan akan memanggil rsa_instance.decrypt_text(...) untuk mengembalikannya ke bentuk string asli.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1 Spesifikasi Teknis Program

Program *CV Scanner ATS* dirancang untuk melakukan pencarian kata kunci pada dokumen CV digital menggunakan berbagai algoritma pencocokan string. Sistem dibangun dalam bahasa Python 3.11 dengan struktur modular dan dukungan beberapa pustaka eksternal.

4.1.1. Struktur Data

- **TrieNode**: digunakan dalam implementasi Aho-Corasick untuk menyimpan pohon prefix dari kata kunci.
- **List dan Dictionary (HashMap)**: digunakan untuk menyimpan kata kunci, hasil pencarian, serta representasi data hasil ekstraksi.
- **Queue (deque)**: digunakan dalam traversal BFS saat membangun tautan kegagalan Aho-Corasick.
- **Tuple dan Custom Class (RSA)**: untuk manajemen enkripsi dan dekripsi data pelamar.

4.1.2. Fungsi dan Prosedur Utama

1. **String Matching Algorithms**
    - kmp_search(text, pattern) dan compute_border(pattern): untuk algoritma KMP.
    - bm_search_all_occurrences(text, pattern): untuk Boyer-Moore.
    - aho_corasick(text, keywords): untuk Aho-Corasick multi-pattern matching.
2. **Fuzzy Matching**
    - levenshtein_distance(s1, s2): menghitung jarak edit.
    - find_fuzzy_matches_hybrid(text, keyword, max_dist): pencocokan fuzzy gabungan berbasis kata dan sliding window.
3. **Regex Extraction**

- extract_skills(text), extract_experience(text), extract_education(text): menggunakan ekspresi reguler untuk mengidentifikasi bagian penting dari CV.

4. **Enkripsi Data**

   - RSA.encrypt_text(), RSA.decrypt_text(): enkripsi profil pelamar menggunakan algoritma RSA.
   - encrypt_data(), decrypt_data(): memproses dictionary yang berisi data sensitif.

5. **Antarmuka dan Backend**

   - main.py: sebagai entry point program.
   - database.py: koneksi dan pengambilan data MySQL.
   - gui.py: membangun antarmuka pengguna dengan tkinter.
   - multiprocessing.Pool: digunakan untuk mempercepat pemrosesan banyak CV secara paralel.

4.2 Penjelasan Tata Cara

Langkah-langkah penggunaan aplikasi CV Scanner adalah sebagai berikut:

1. Input Kata Kunci
   Pengguna memasukkan daftar kata kunci yang ingin dicocokkan dari CV kandidat (misalnya: Python, Excel, Data).

2. Pemilihan Algoritma
   Pengguna memilih algoritma string matching yang akan digunakan:

   KMP (Knuth-Morris-Pratt)

   BM (Boyer-Moore)

   Aho-Corasick (untuk multi-pattern sekaligus)

3. Menentukan Jumlah Hasil Teratas
   Pengguna menentukan jumlah maksimum CV yang ingin ditampilkan.

4. Klik Tombol “Search CVs”

   Aplikasi akan memproses semua dokumen CV dan mencocokkan kata kunci menggunakan algoritma yang dipilih.

5. Eksekusi *Backend*

   - Program membaca data dari database MySQL.
   - Ekstrak teks dari PDF CV pelamar.
   - Terapkan algoritma pencocokan string.
   - Fallback fuzzy matching menggunakan Levenshtein Distance jika exact match tidak ditemukan.

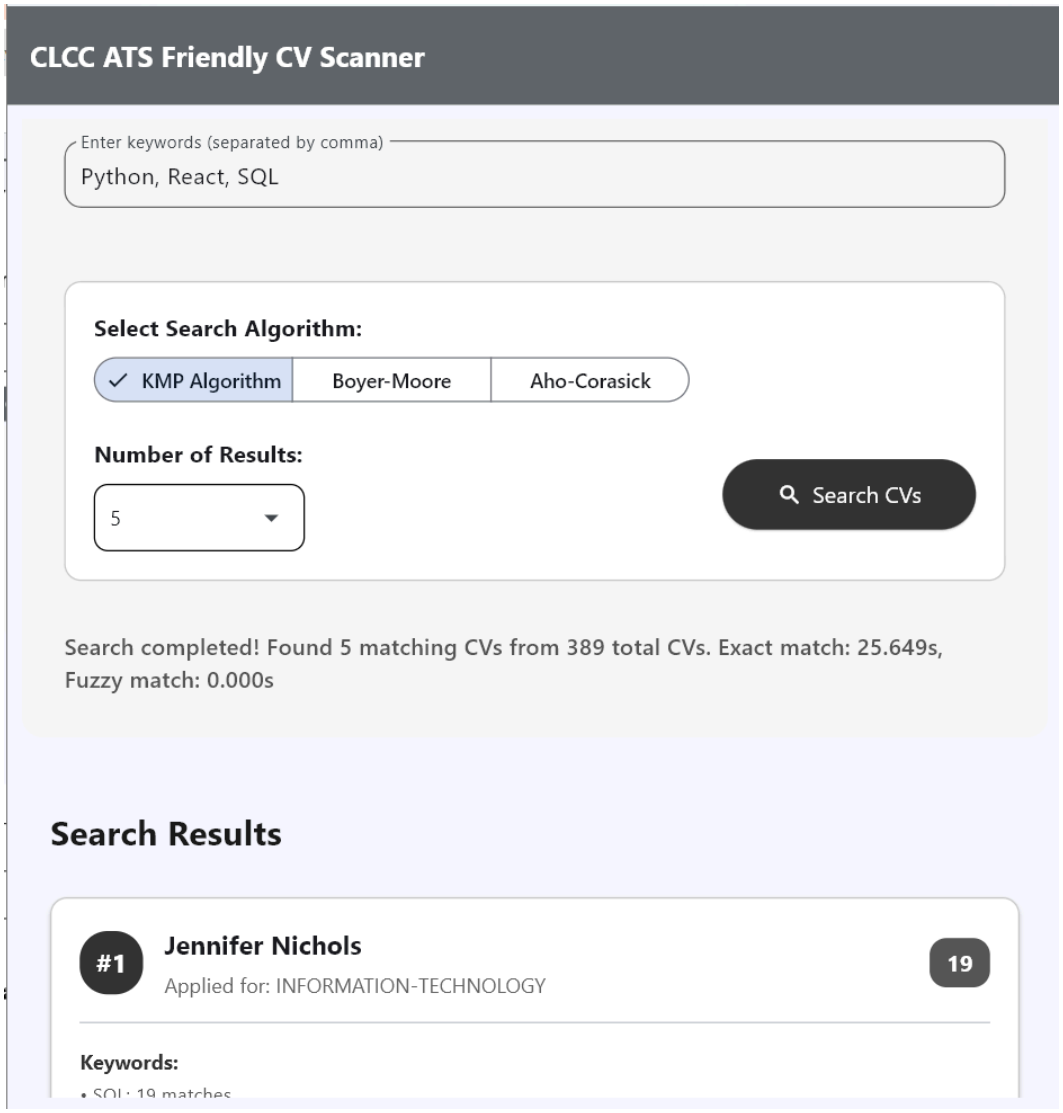
6. *Output*

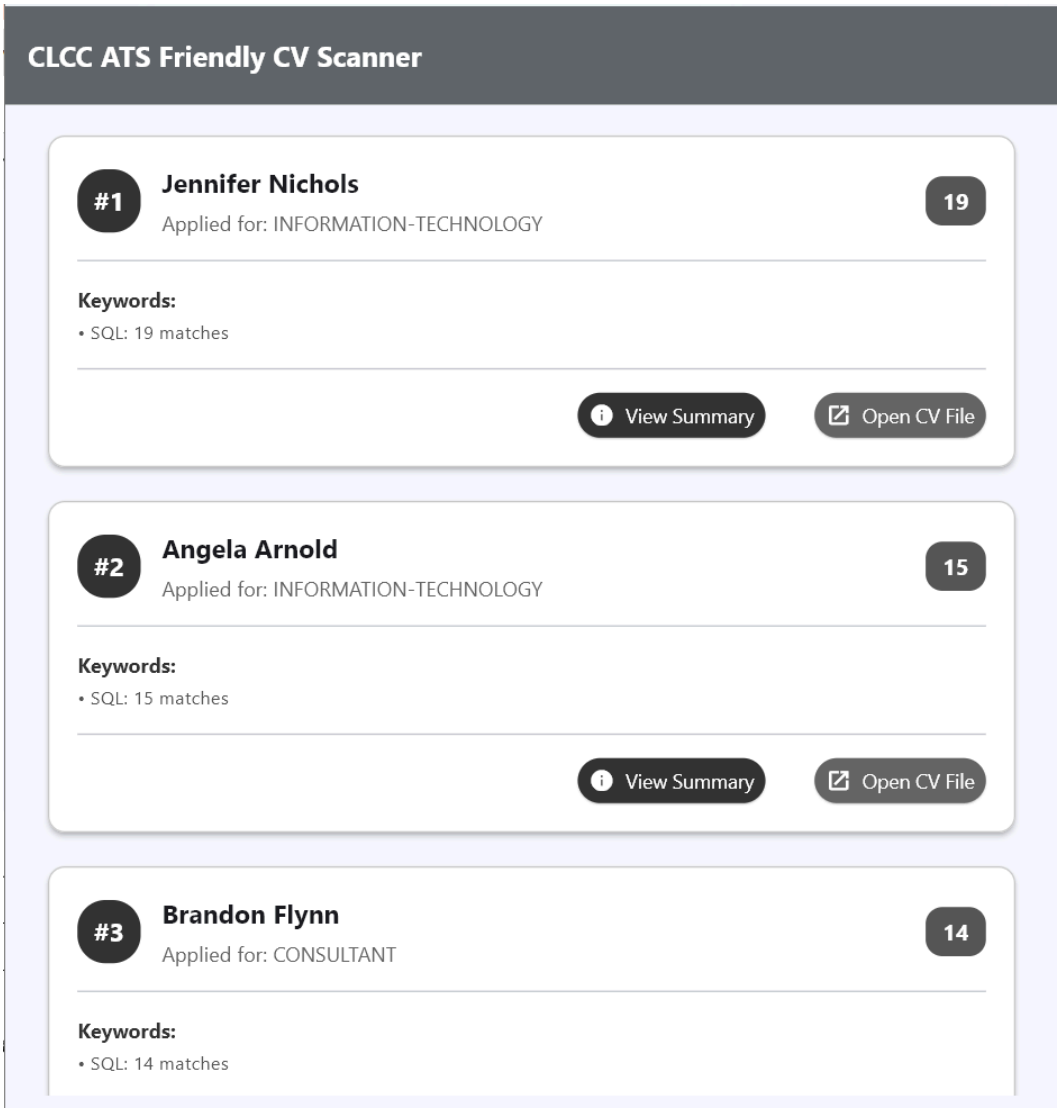
   Aplikasi akan menampilkan daftar ringkasan CV pelamar dengan skor kecocokan tertinggi. Skor dihitung berdasarkan jumlah kemunculan kata kunci.

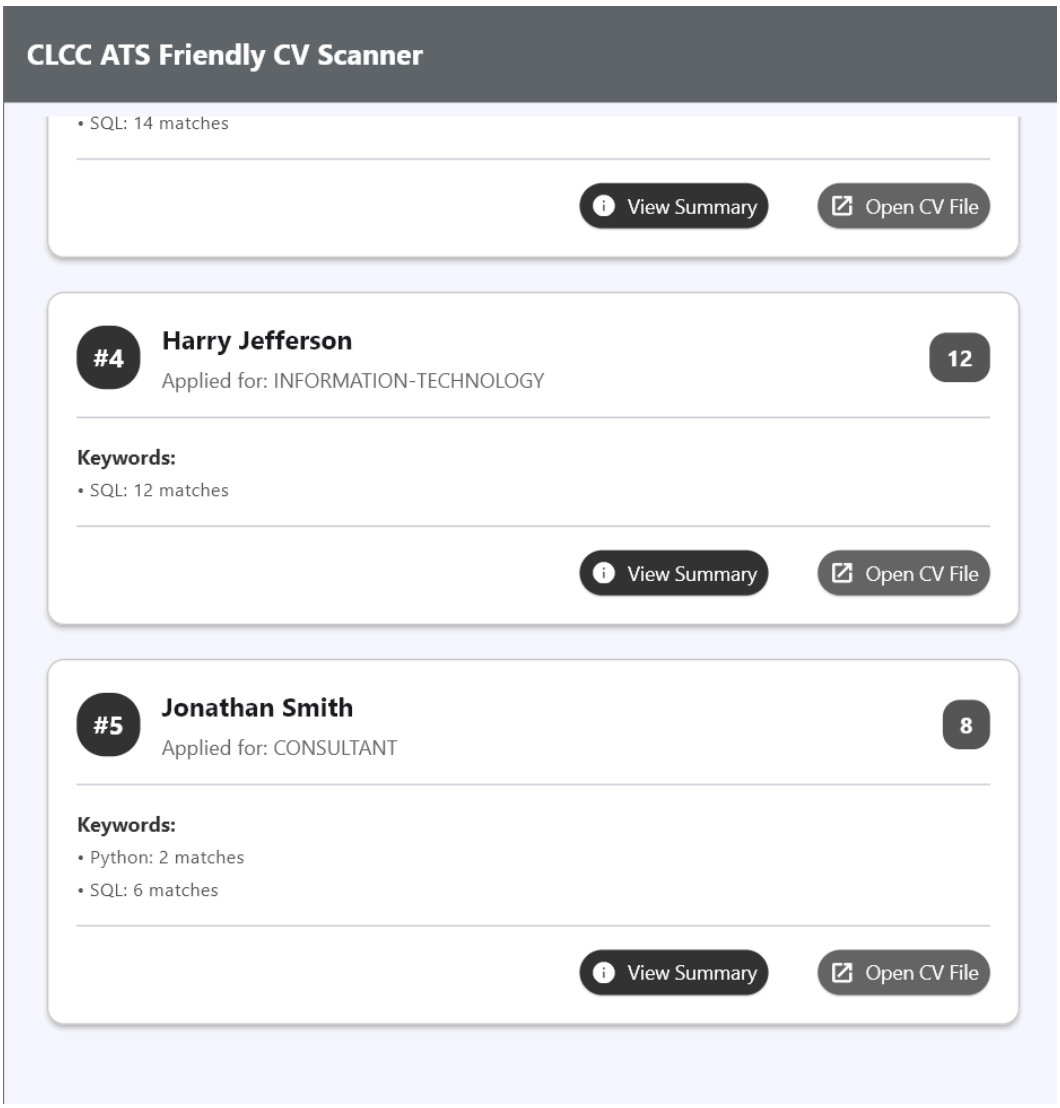
4.3 Hasil Pengujian

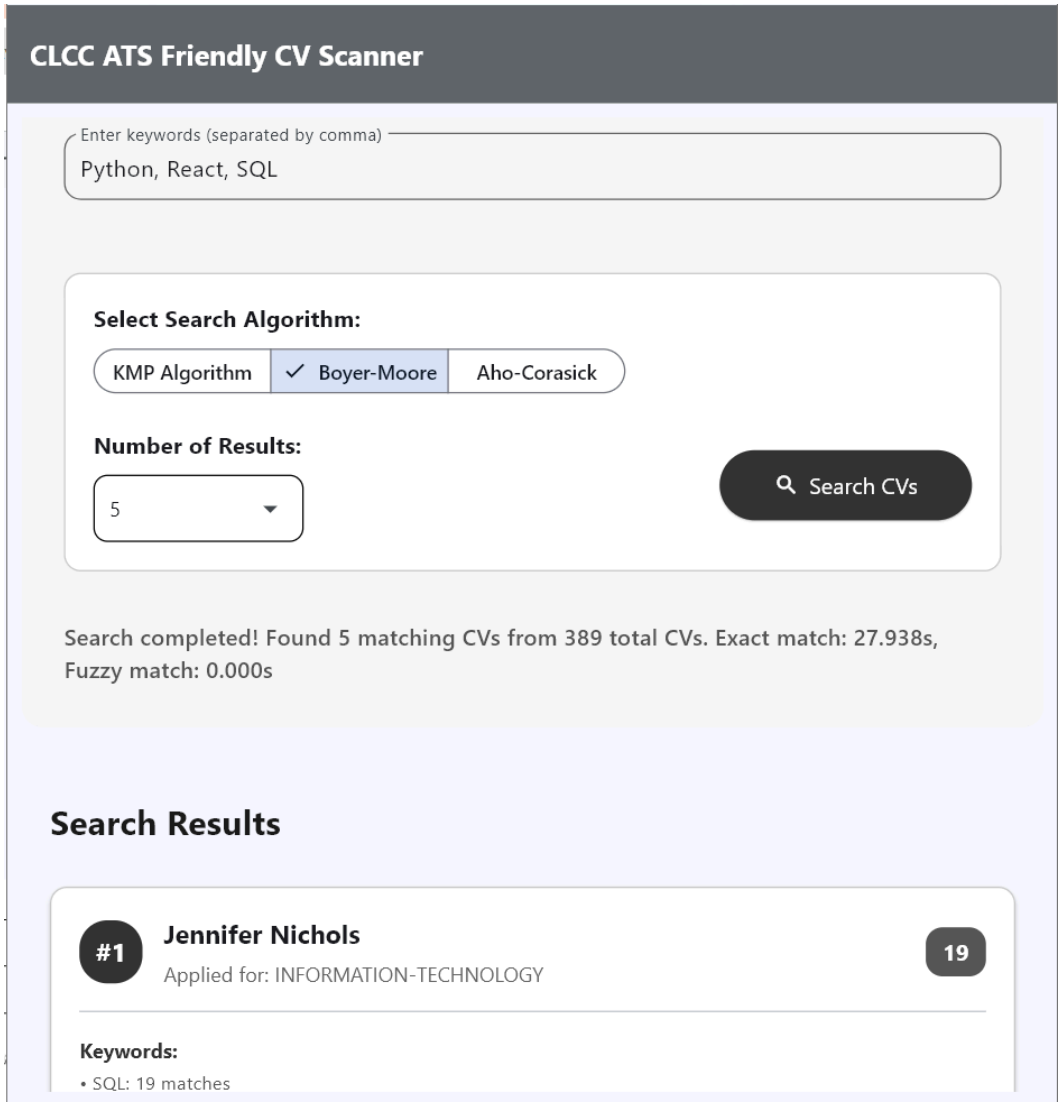
4.3.1 Pengujian 1

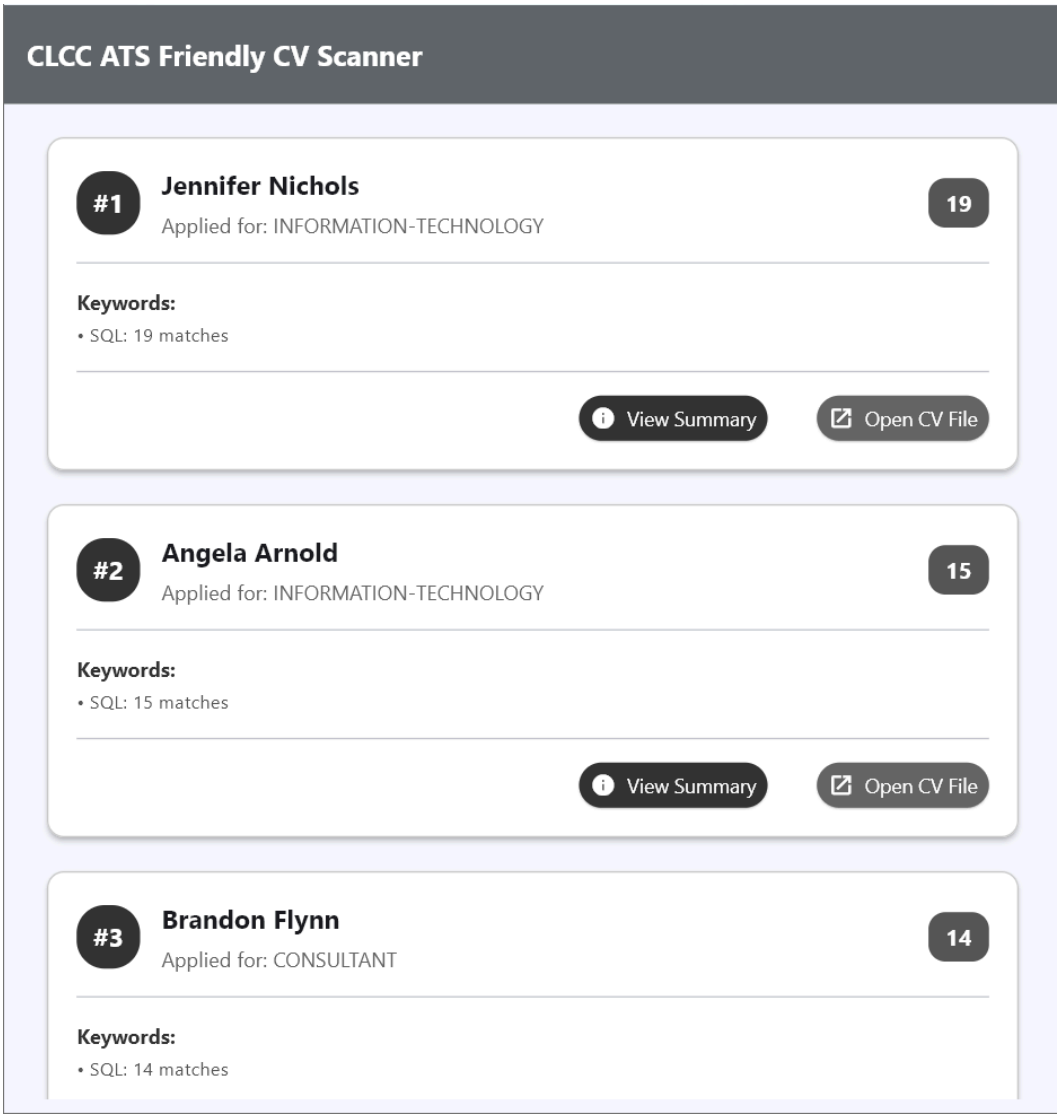
Query = Python, React, SQL

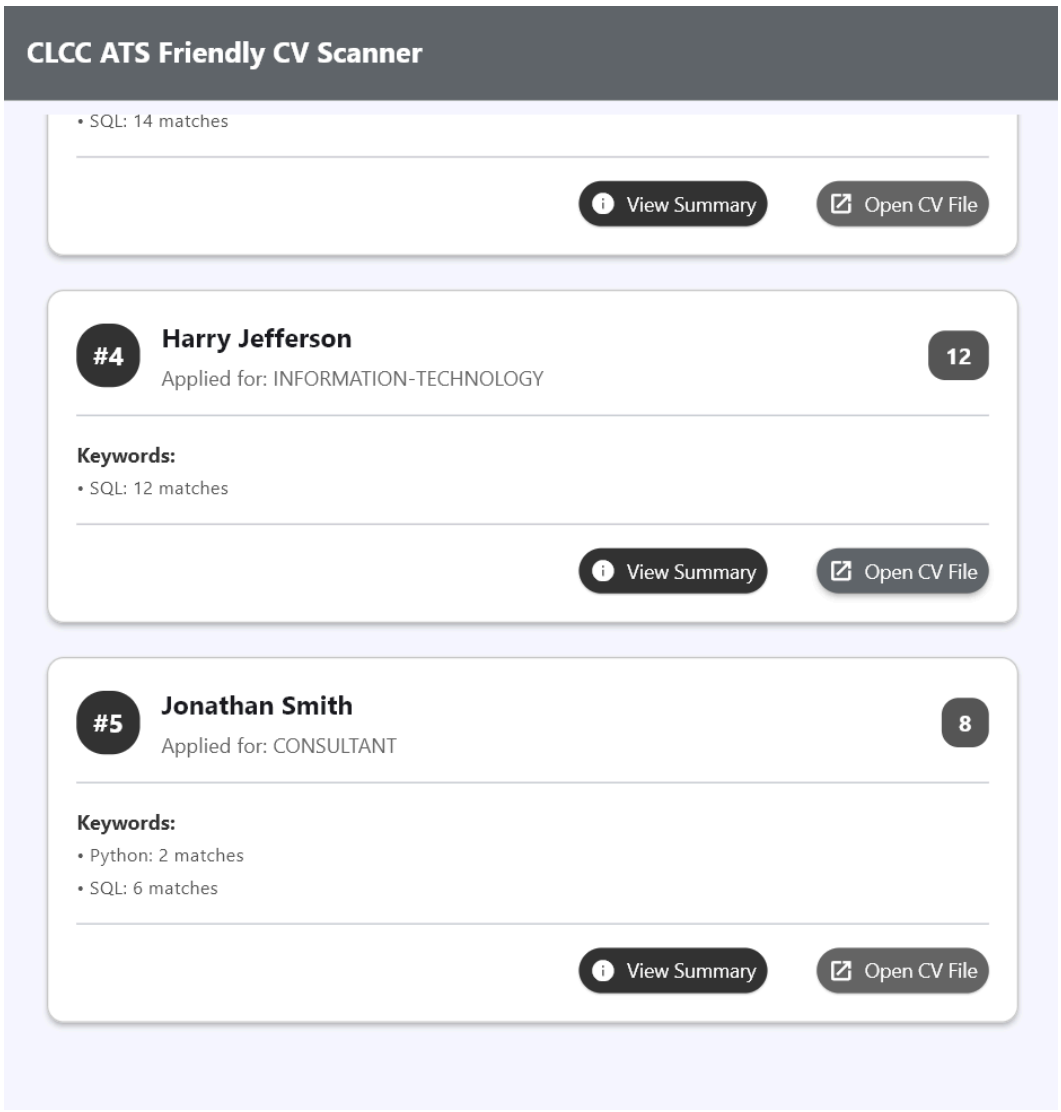
Algoritma	Hasil
KMP	

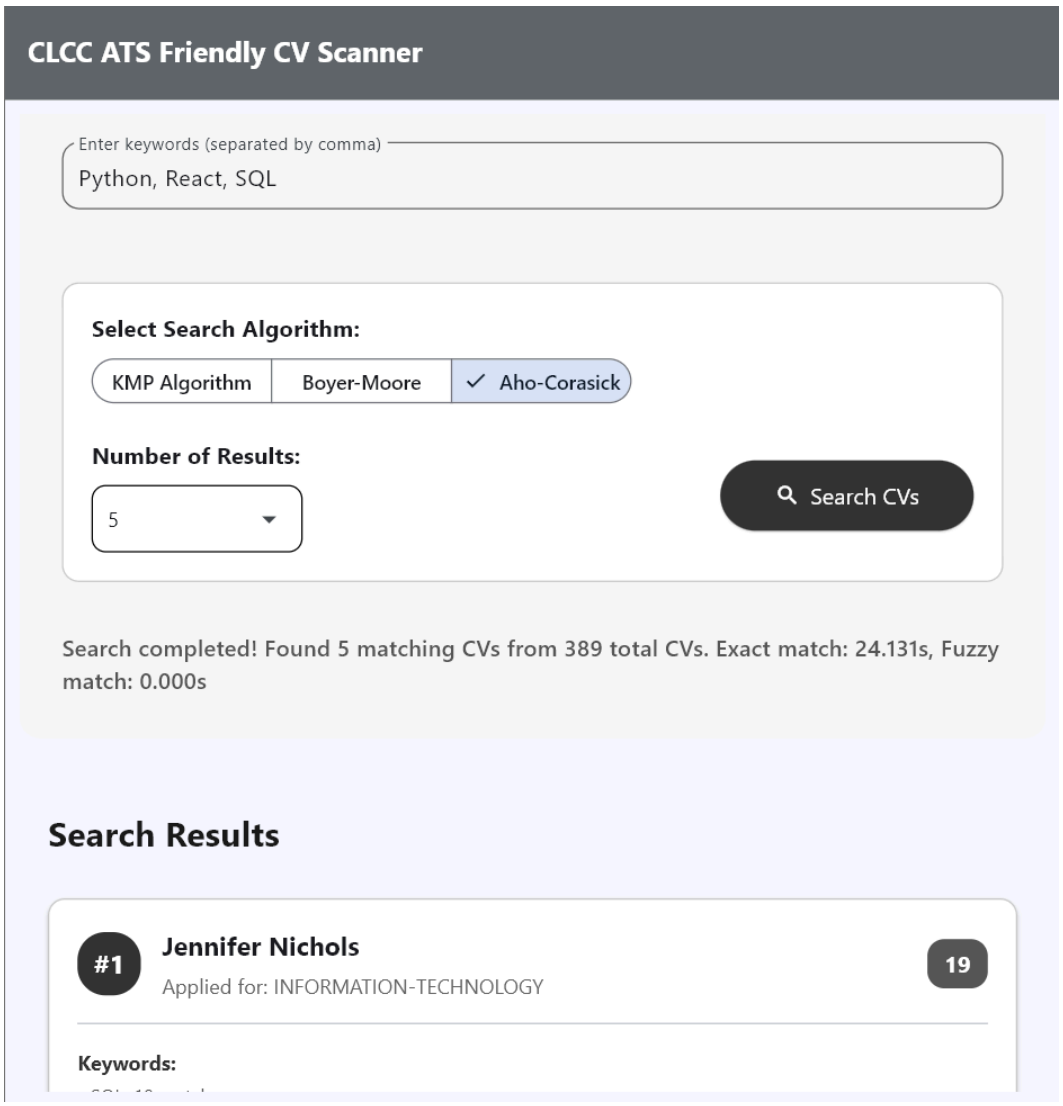

CLCC ATS Friendly CV Scanner
#1
Jennifer Nichols
Applied for: INFORMATION-TECHNOLOGY
19
Keywords:
• SQL: 19 matches
View Summary
Open CV File
#2
Angela Arnold
Applied for: INFORMATION-TECHNOLOGY
15
Keywords:
• SQL: 15 matches
View Summary
Open CV File
#3
Brandon Flynn
Applied for: CONSULTANT
14
Keywords:
• SQL: 14 matches

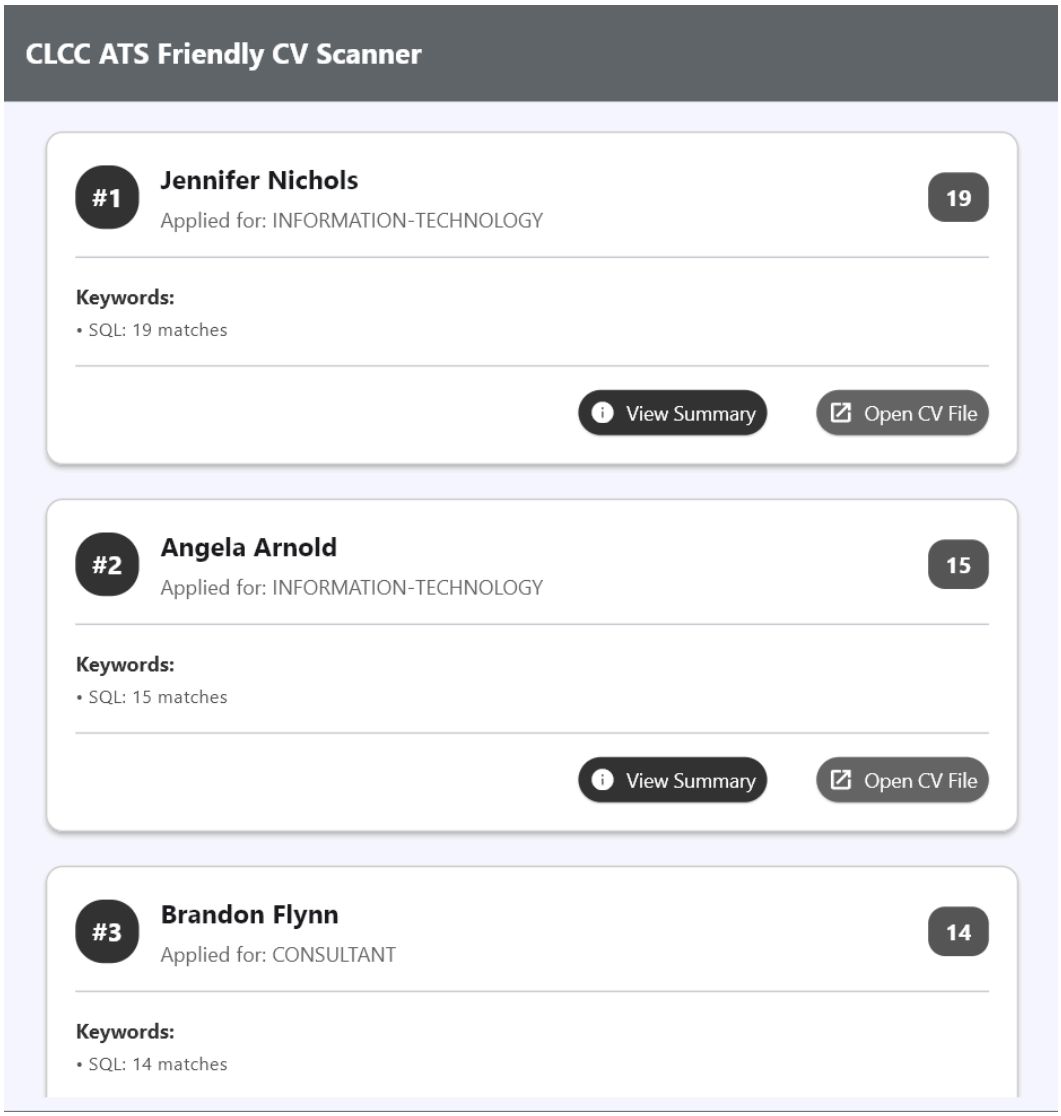

CLCC ATS Friendly CV Scanner
• SQL: 14 matches
View Summary
Open CV File
#4
Harry Jefferson
Applied for: INFORMATION-TECHNOLOGY
12
Keywords:
• SQL: 12 matches
View Summary
Open CV File
#5
Jonathan Smith
Applied for: CONSULTANT
8
Keywords:
• Python: 2 matches
• SQL: 6 matches
View Summary
Open CV File

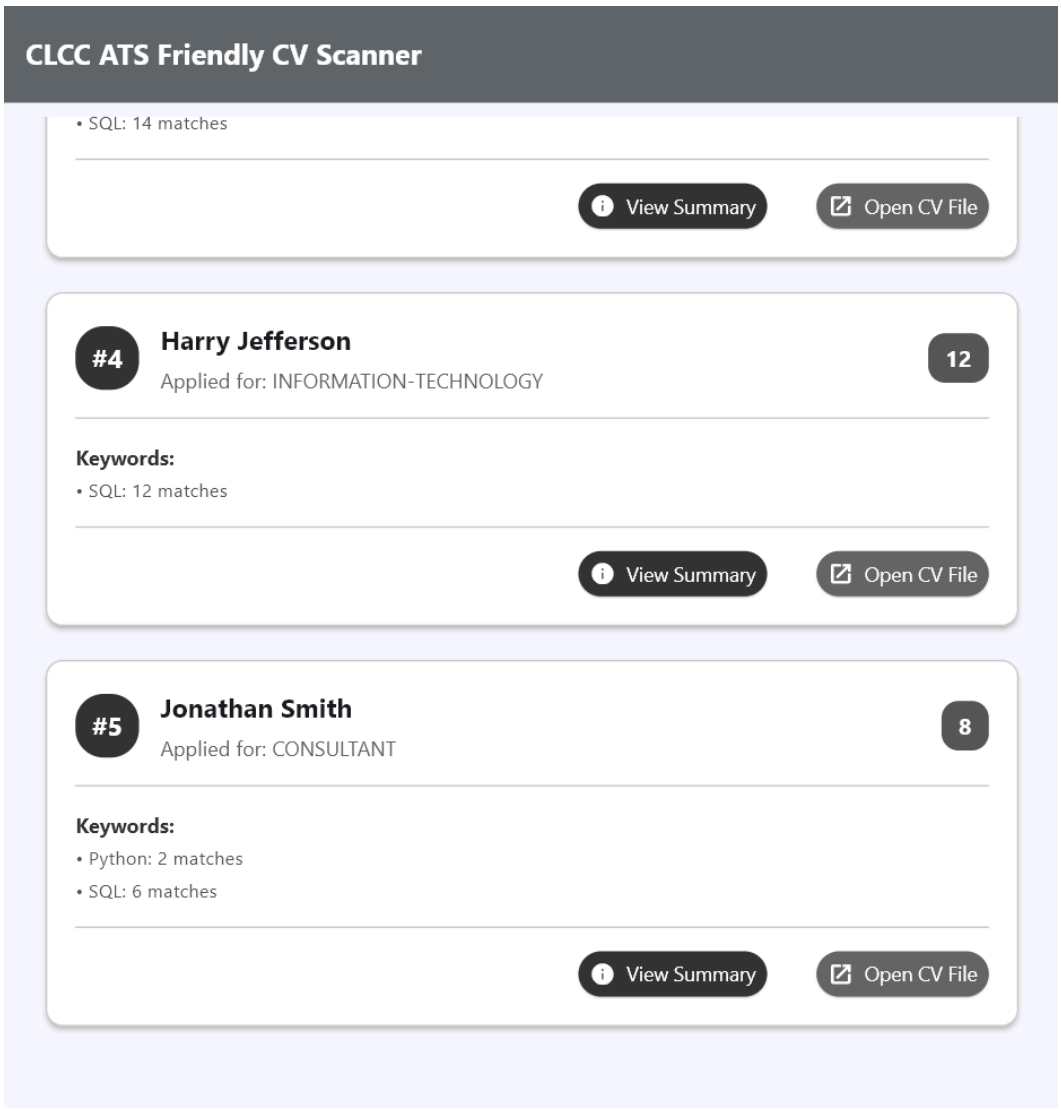
BM	 CLCC ATS Friendly CV Scanner Enter keywords (separated by comma) Python, React, SQL Select Search Algorithm: KMP Algorithm Boyer-Moore Aho-Corasick Number of Results: 5 Search CVs Search completed! Found 5 matching CVs from 389 total CVs. Exact match: 27.938s, Fuzzy match: 0.000s Search Results #1 Jennifer Nichols Applied for: INFORMATION-TECHNOLOGY 19 Keywords: • SQL: 19 matches


CLCC ATS Friendly CV Scanner
#1
Jennifer Nichols
Applied for: INFORMATION-TECHNOLOGY
19
Keywords:
• SQL: 19 matches
View Summary
Open CV File
#2
Angela Arnold
Applied for: INFORMATION-TECHNOLOGY
15
Keywords:
• SQL: 15 matches
View Summary
Open CV File
#3
Brandon Flynn
Applied for: CONSULTANT
14
Keywords:
• SQL: 14 matches


CLCC ATS Friendly CV Scanner
• SQL: 14 matches
View Summary
Open CV File
#4
Harry Jefferson
Applied for: INFORMATION-TECHNOLOGY
12
Keywords:
• SQL: 12 matches
View Summary
Open CV File
#5
Jonathan Smith
Applied for: CONSULTANT
8
Keywords:
• Python: 2 matches
• SQL: 6 matches
View Summary
Open CV File

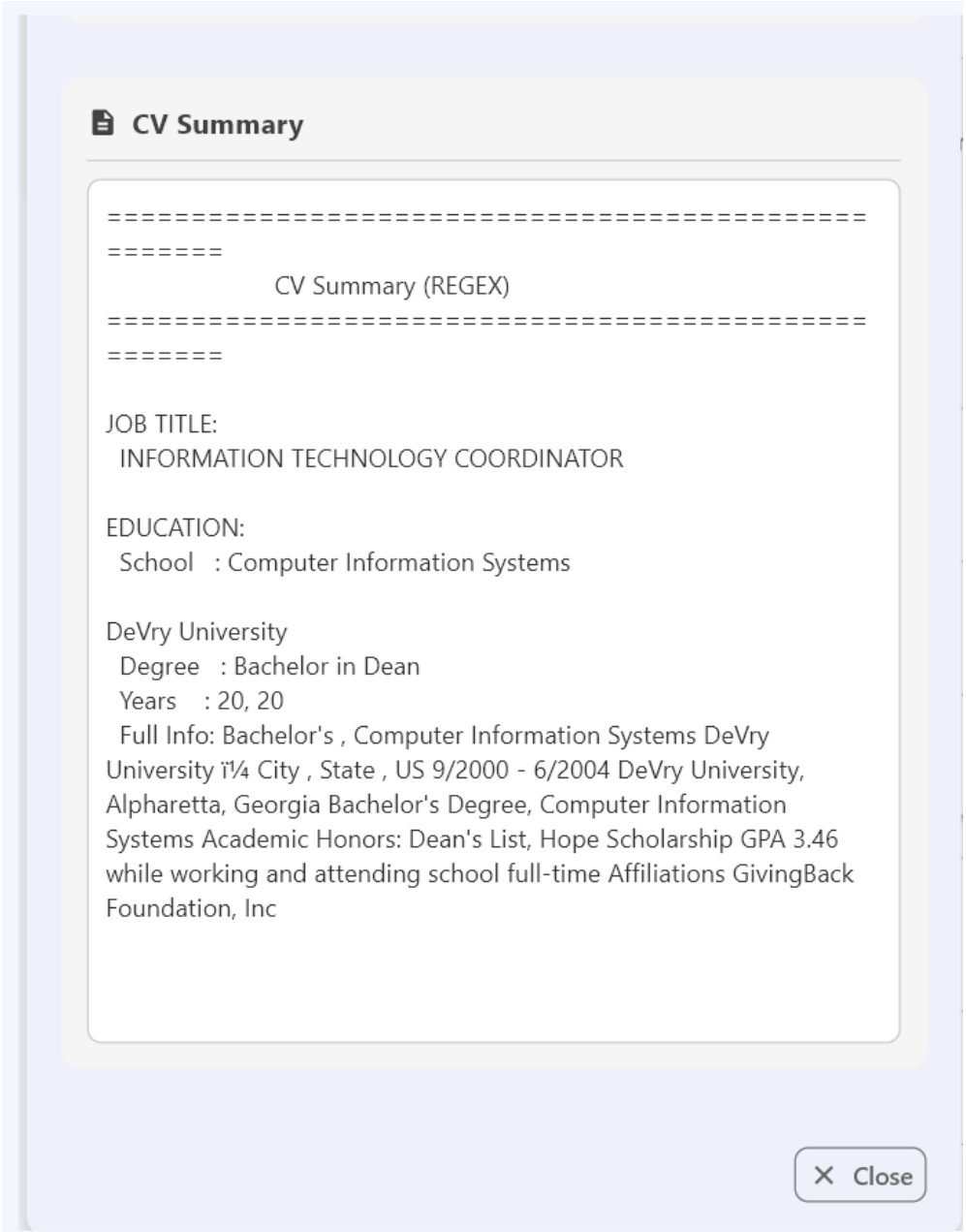
Aho-C	 CLCC ATS Friendly CV Scanner Enter keywords (separated by comma) Python, React, SQL Select Search Algorithm: KMP Algorithm Boyer-Moore Aho-Corasick Number of Results: 5 Search CVs Search completed! Found 5 matching CVs from 389 total CVs. Exact match: 24.131s, Fuzzy match: 0.000s Search Results #1 Jennifer Nichols Applied for: INFORMATION-TECHNOLOGY 19 Keywords:


CLCC ATS Friendly CV Scanner
#1
Jennifer Nichols
Applied for: INFORMATION-TECHNOLOGY
19
Keywords:
• SQL: 19 matches
View Summary
Open CV File
#2
Angela Arnold
Applied for: INFORMATION-TECHNOLOGY
15
Keywords:
• SQL: 15 matches
View Summary
Open CV File
#3
Brandon Flynn
Applied for: CONSULTANT
14
Keywords:
• SQL: 14 matches


CLCC ATS Friendly CV Scanner
• SQL: 14 matches
View Summary
Open CV File
#4
Harry Jefferson
Applied for: INFORMATION-TECHNOLOGY
12
Keywords:
• SQL: 12 matches
View Summary
Open CV File
#5
Jonathan Smith
Applied for: CONSULTANT
8
Keywords:
• Python: 2 matches
• SQL: 6 matches
View Summary
Open CV File

Hasil CV Summary:

Jennifer Nichols (TOP 1)



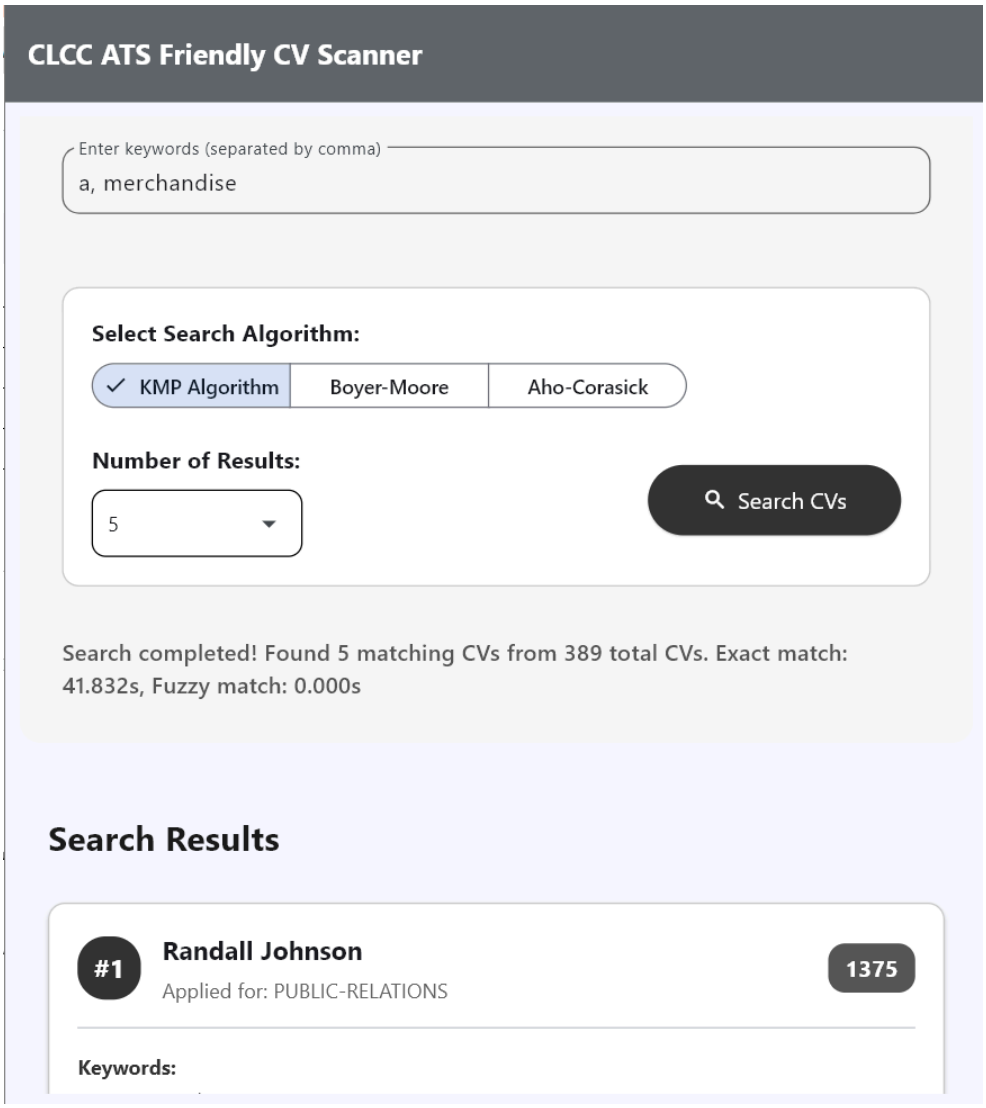
CV Summary

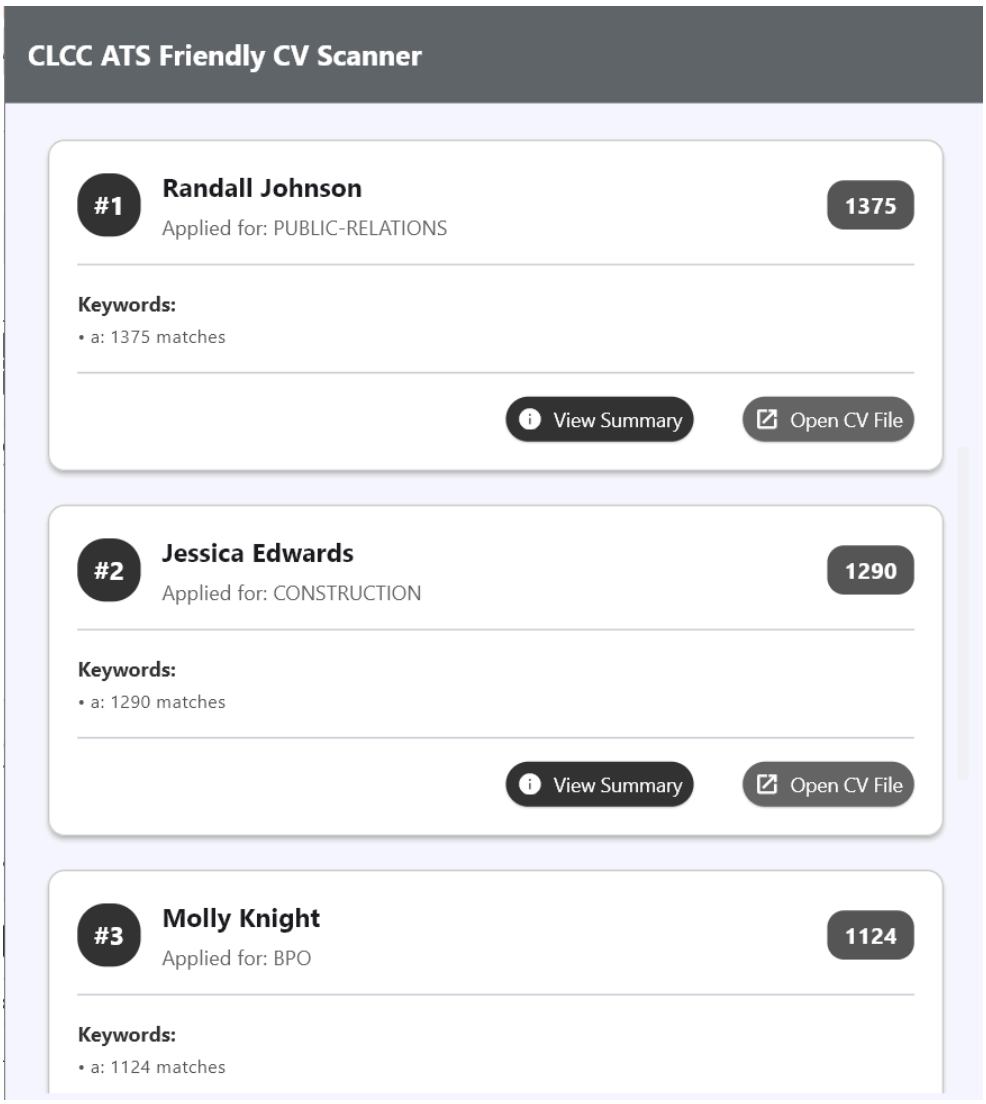
```
==========================================
=======
                CV Summary (REGEX)
==========================================
=======

JOB TITLE:
  INFORMATION TECHNOLOGY COORDINATOR

EDUCATION:
  School   : Computer Information Systems

DeVry University
  Degree   : Bachelor in Dean
  Years    : 20, 20
  Full Info: Bachelor's , Computer Information Systems DeVry
University ï¼ City , State , US 9/2000 - 6/2004 DeVry University,
Alpharetta, Georgia Bachelor's Degree, Computer Information
Systems Academic Honors: Dean's List, Hope Scholarship GPA 3.46
while working and attending school full-time Affiliations GivingBack
Foundation, Inc
```

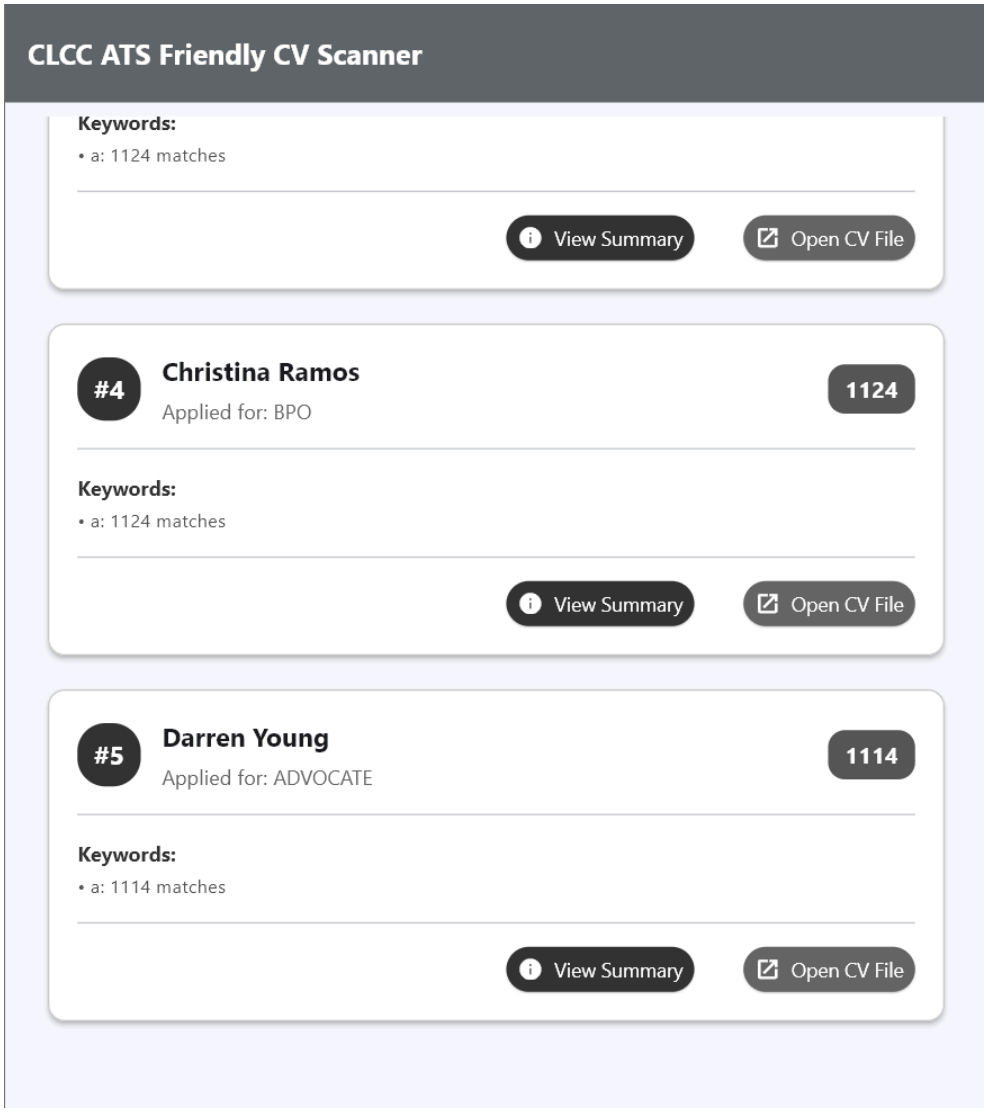
Close

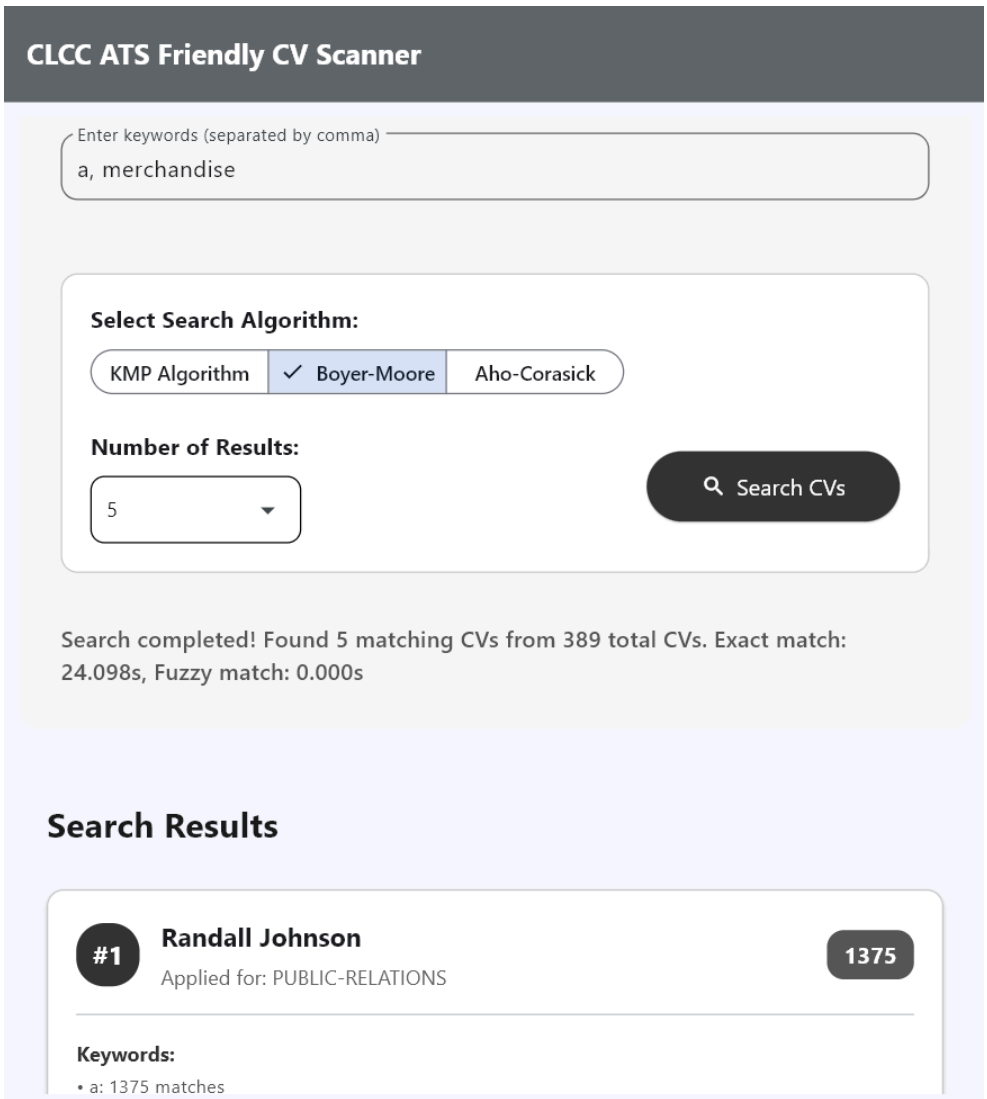
4.3.2 Pengujian 2

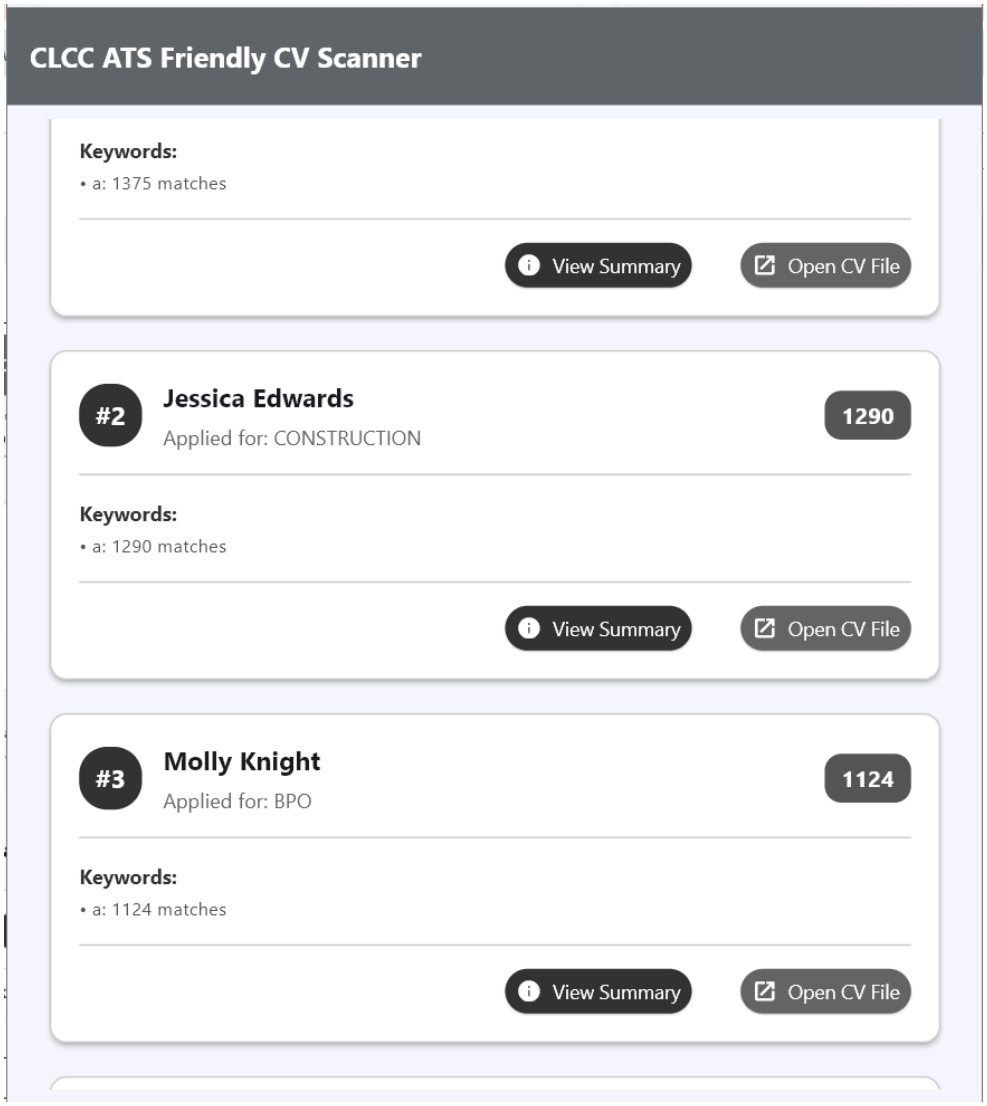
Query = a, merchandise

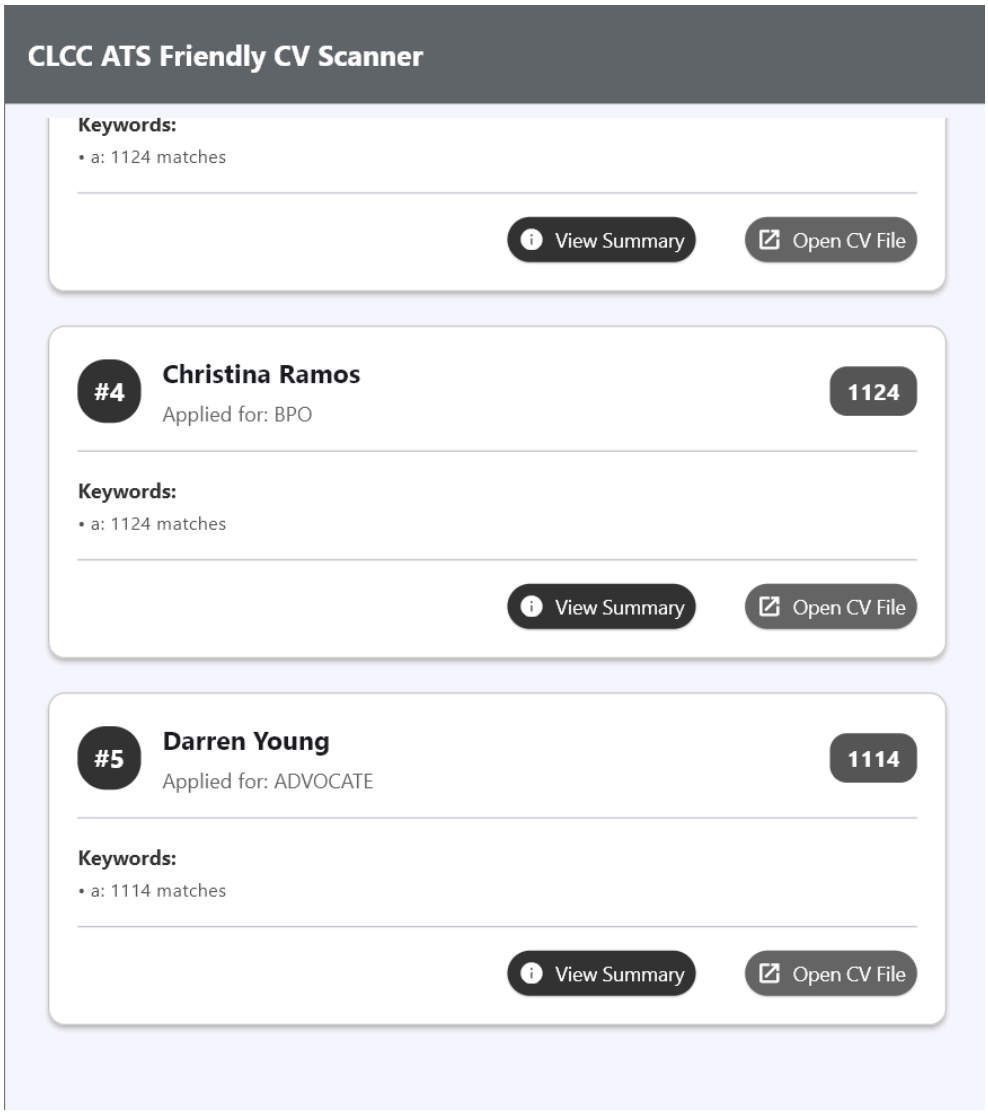
Algortima	Hasil
KMP	

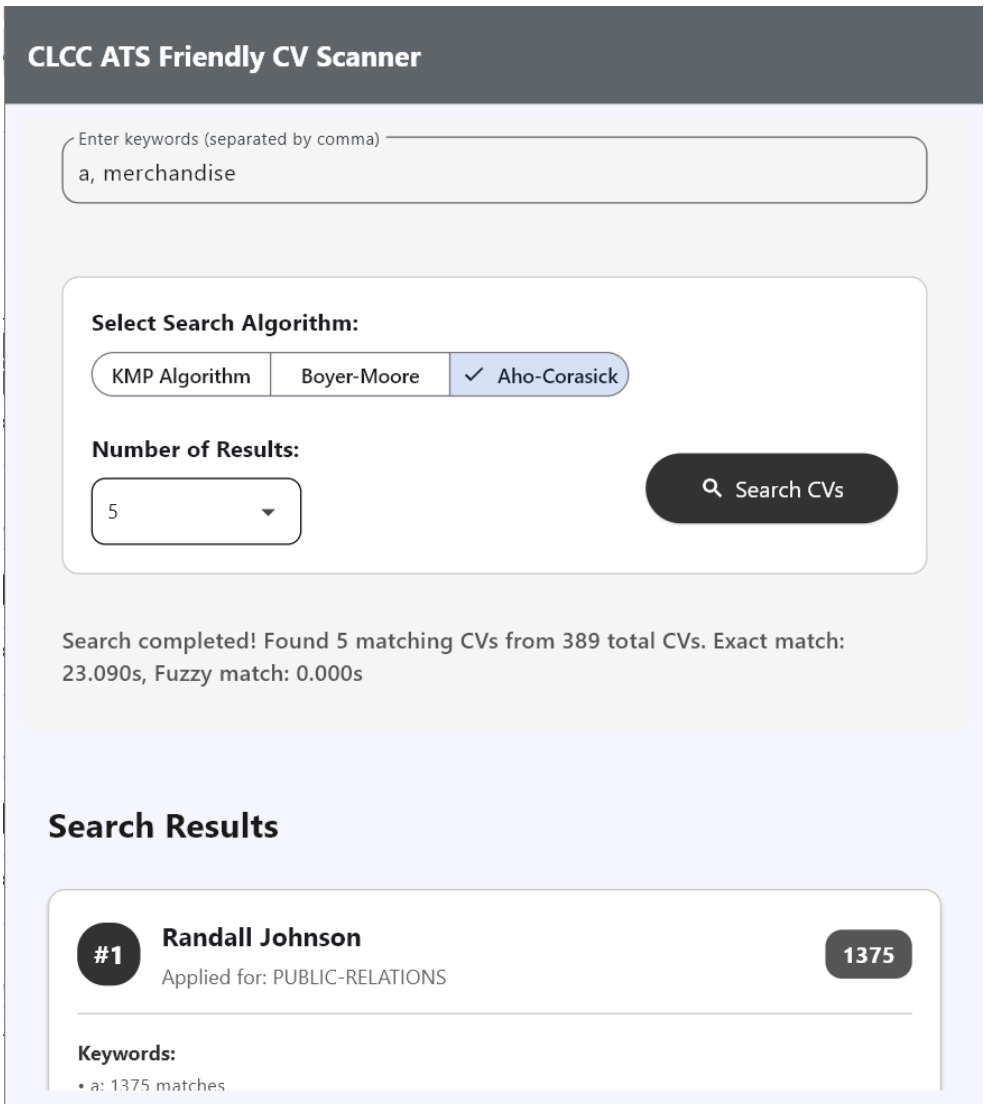

CLCC ATS Friendly CV Scanner
#1
Randall Johnson
Applied for: PUBLIC-RELATIONS
1375
Keywords:
• a: 1375 matches
View Summary
Open CV File
#2
Jessica Edwards
Applied for: CONSTRUCTION
1290
Keywords:
• a: 1290 matches
View Summary
Open CV File
#3
Molly Knight
Applied for: BPO
1124
Keywords:
• a: 1124 matches

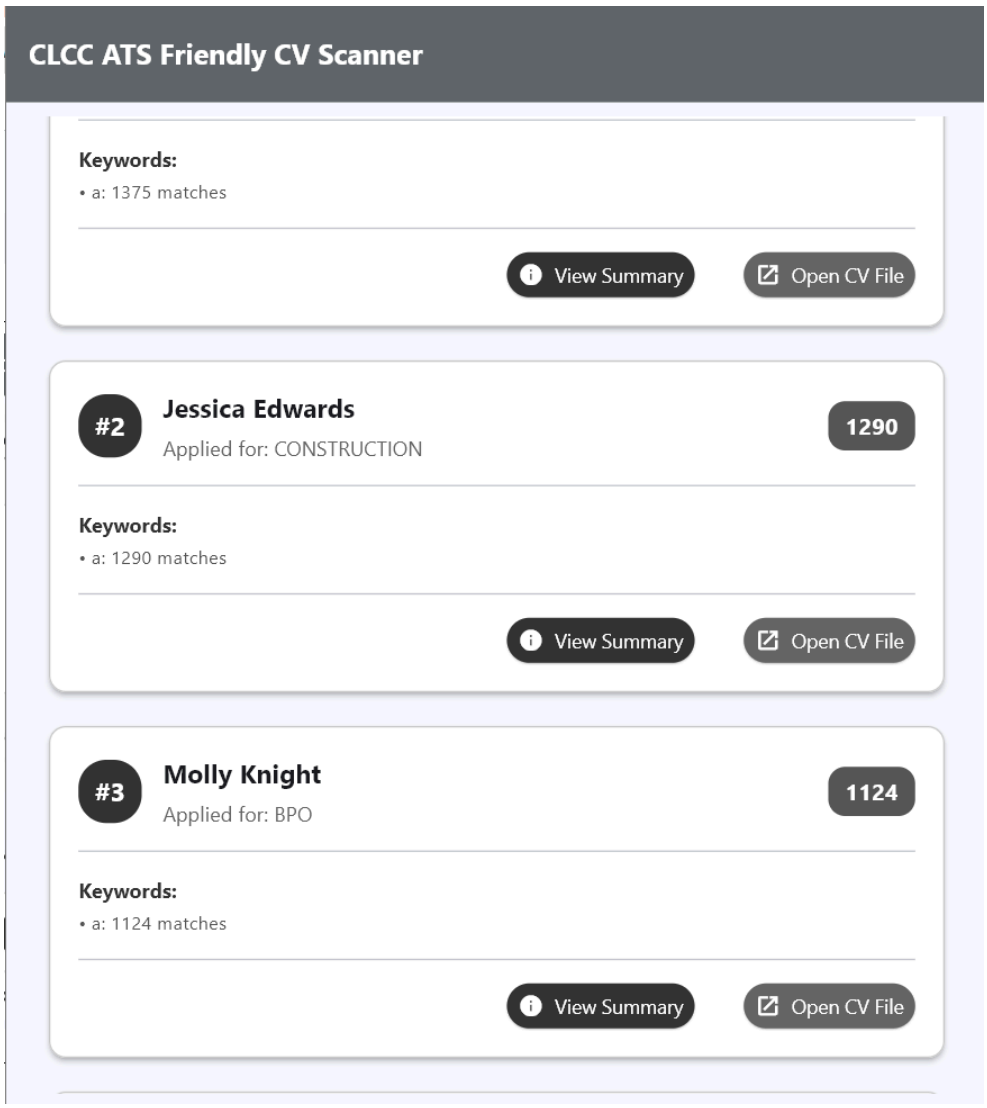

CLCC ATS Friendly CV Scanner
Keywords:
• a: 1124 matches
View Summary
Open CV File
#4
Christina Ramos
Applied for: BPO
1124
Keywords:
• a: 1124 matches
View Summary
Open CV File
#5
Darren Young
Applied for: ADVOCATE
1114
Keywords:
• a: 1114 matches
View Summary
Open CV File

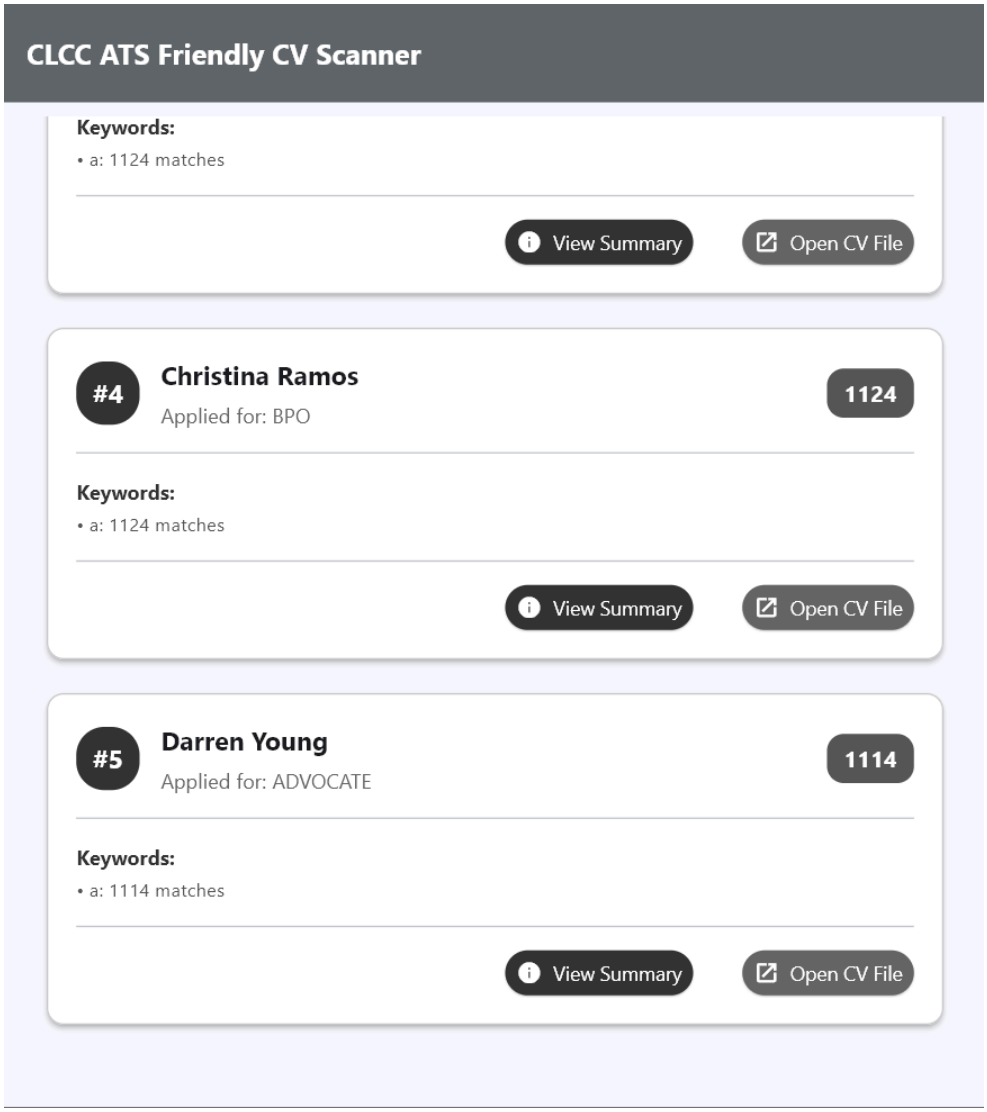
BM	 CLCC ATS Friendly CV Scanner Enter keywords (separated by comma) a, merchandise Select Search Algorithm: KMP Algorithm Boyer-Moore Aho-Corasick Number of Results: 5 Search CVs Search completed! Found 5 matching CVs from 389 total CVs. Exact match: 24.098s, Fuzzy match: 0.000s Search Results #1 Randall Johnson Applied for: PUBLIC-RELATIONS 1375 Keywords: • a: 1375 matches


CLCC ATS Friendly CV Scanner
Keywords:
• a: 1375 matches
View Summary
Open CV File
#2
Jessica Edwards
Applied for: CONSTRUCTION
1290
Keywords:
• a: 1290 matches
View Summary
Open CV File
#3
Molly Knight
Applied for: BPO
1124
Keywords:
• a: 1124 matches
View Summary
Open CV File


CLCC ATS Friendly CV Scanner
Keywords:
• a: 1124 matches
View Summary
Open CV File
#4
Christina Ramos
Applied for: BPO
1124
Keywords:
• a: 1124 matches
View Summary
Open CV File
#5
Darren Young
Applied for: ADVOCATE
1114
Keywords:
• a: 1114 matches
View Summary
Open CV File

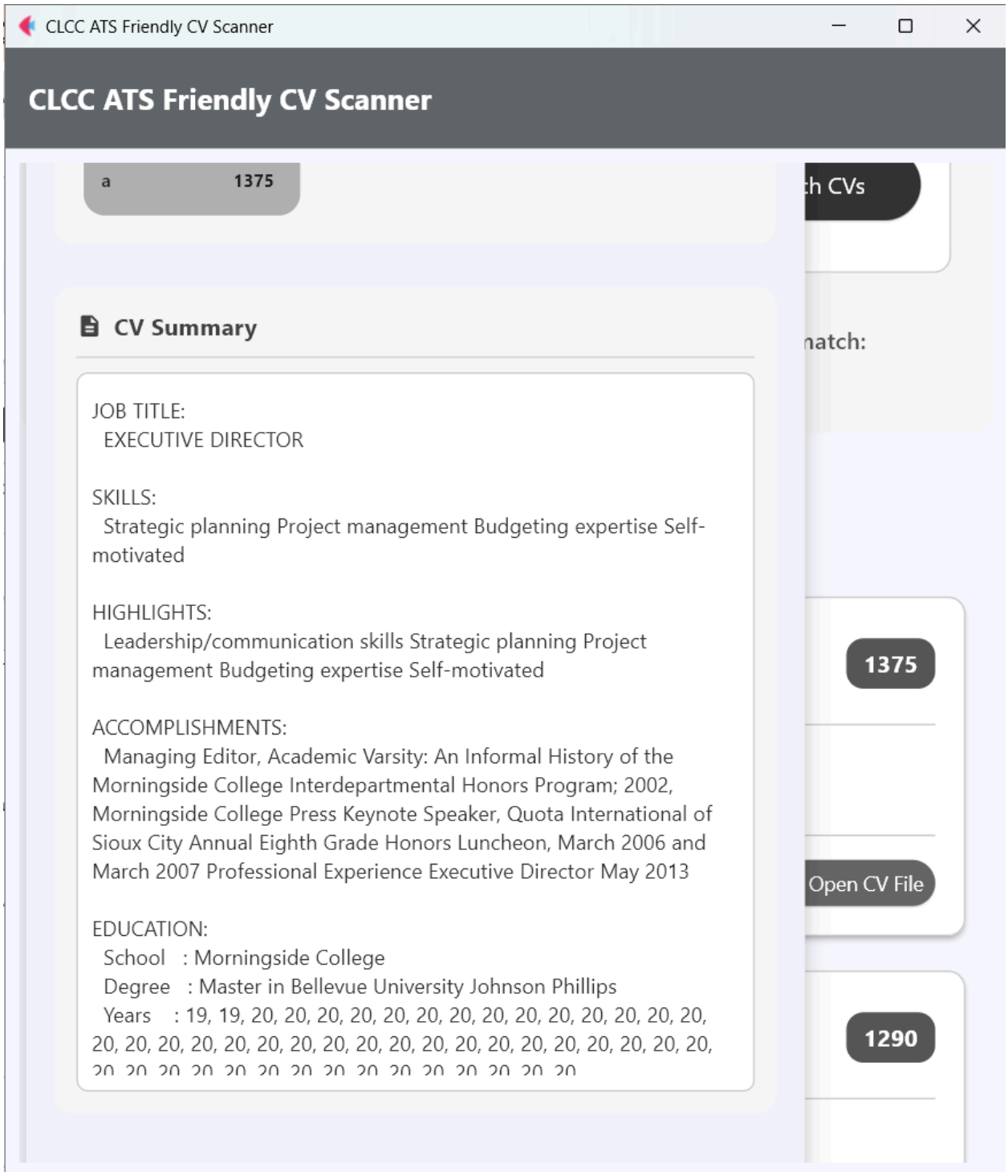
Aho-C	 CLCC ATS Friendly CV Scanner Enter keywords (separated by comma) a, merchandise Select Search Algorithm: KMP Algorithm Boyer-Moore Aho-Corasick Number of Results: 5 Search CVs Search completed! Found 5 matching CVs from 389 total CVs. Exact match: 23.090s, Fuzzy match: 0.000s Search Results #1 Randall Johnson Applied for: PUBLIC-RELATIONS 1375 Keywords: • a: 1375 matches


CLCC ATS Friendly CV Scanner
Keywords:
• a: 1375 matches
View Summary
Open CV File
#2
Jessica Edwards
Applied for: CONSTRUCTION
1290
Keywords:
• a: 1290 matches
View Summary
Open CV File
#3
Molly Knight
Applied for: BPO
1124
Keywords:
• a: 1124 matches
View Summary
Open CV File


CLCC ATS Friendly CV Scanner
Keywords:
• a: 1124 matches
View Summary
Open CV File
#4
Christina Ramos
Applied for: BPO
1124
Keywords:
• a: 1124 matches
View Summary
Open CV File
#5
Darren Young
Applied for: ADVOCATE
1114
Keywords:
• a: 1114 matches
View Summary
Open CV File

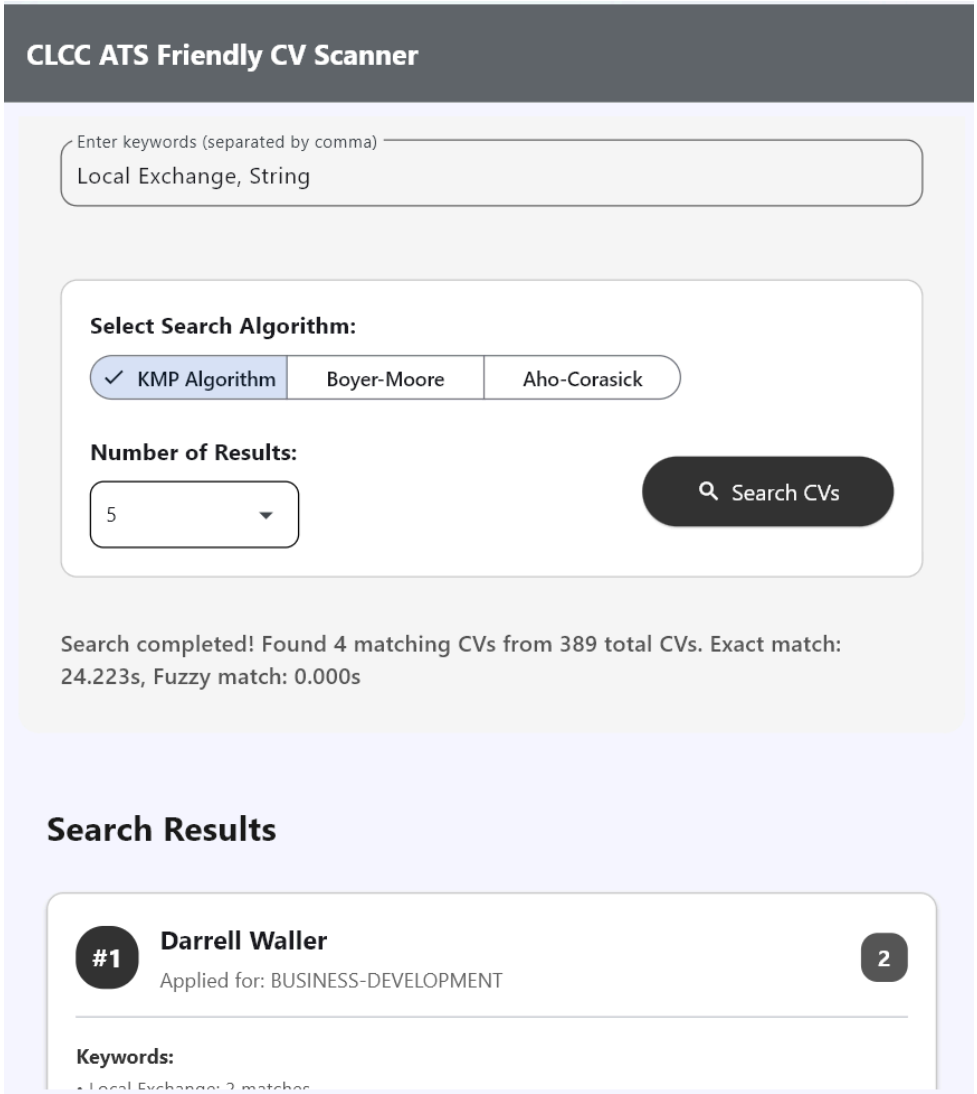
Hasil CV Summary

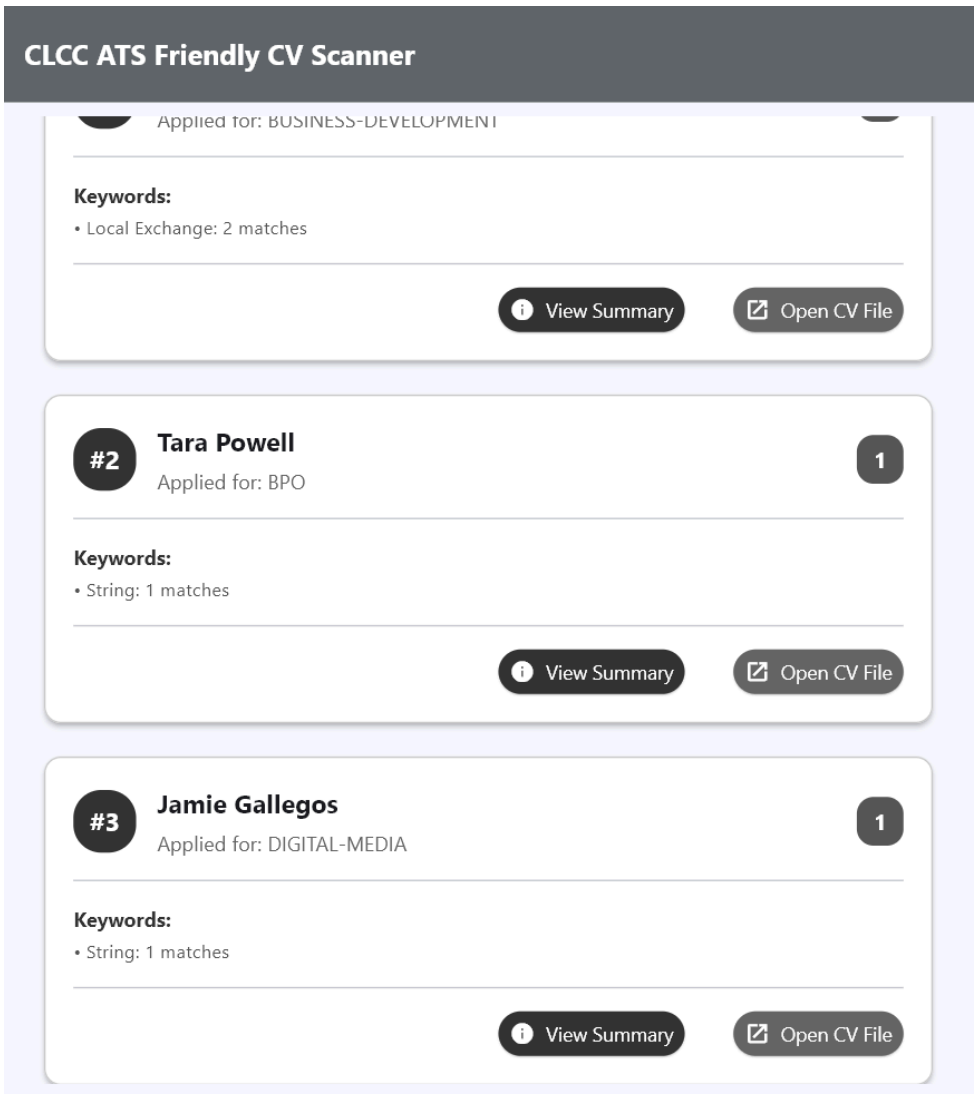
Randall Johnson

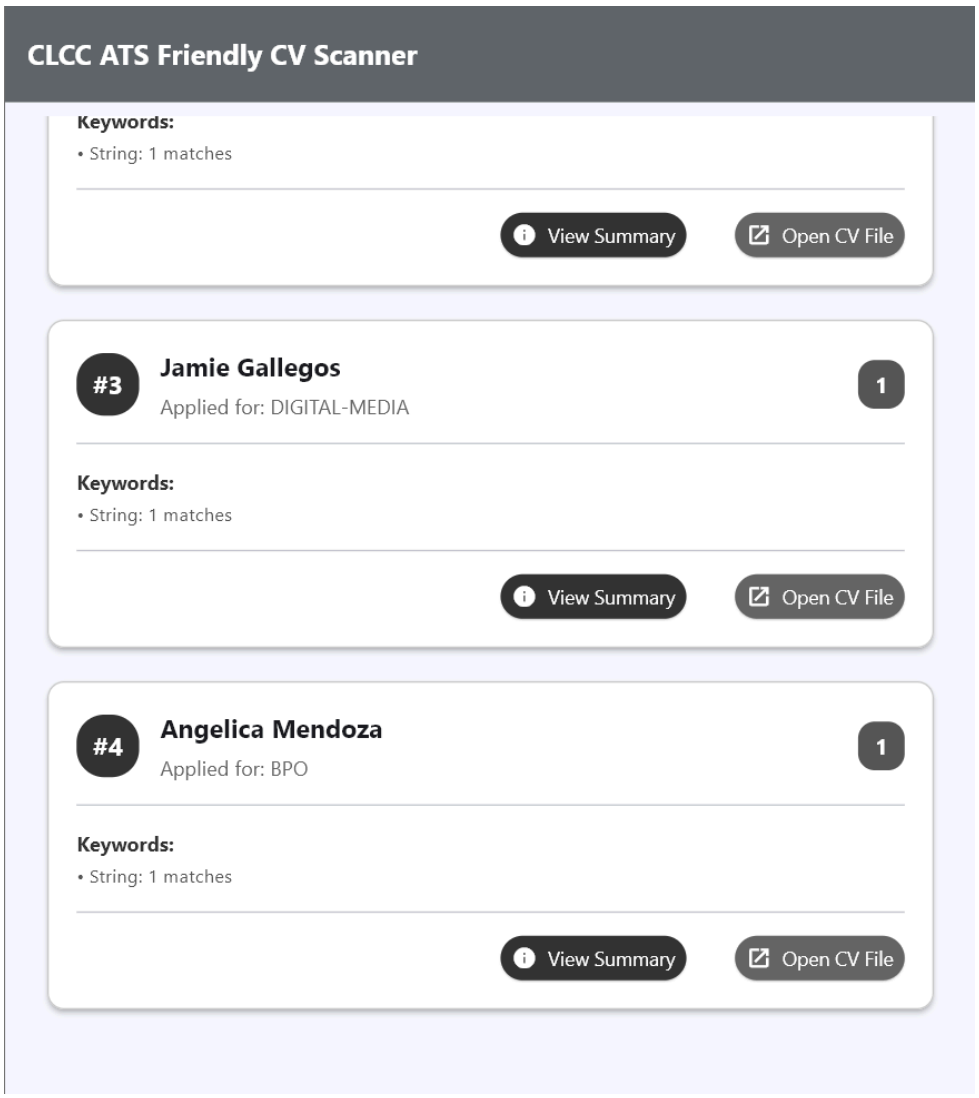


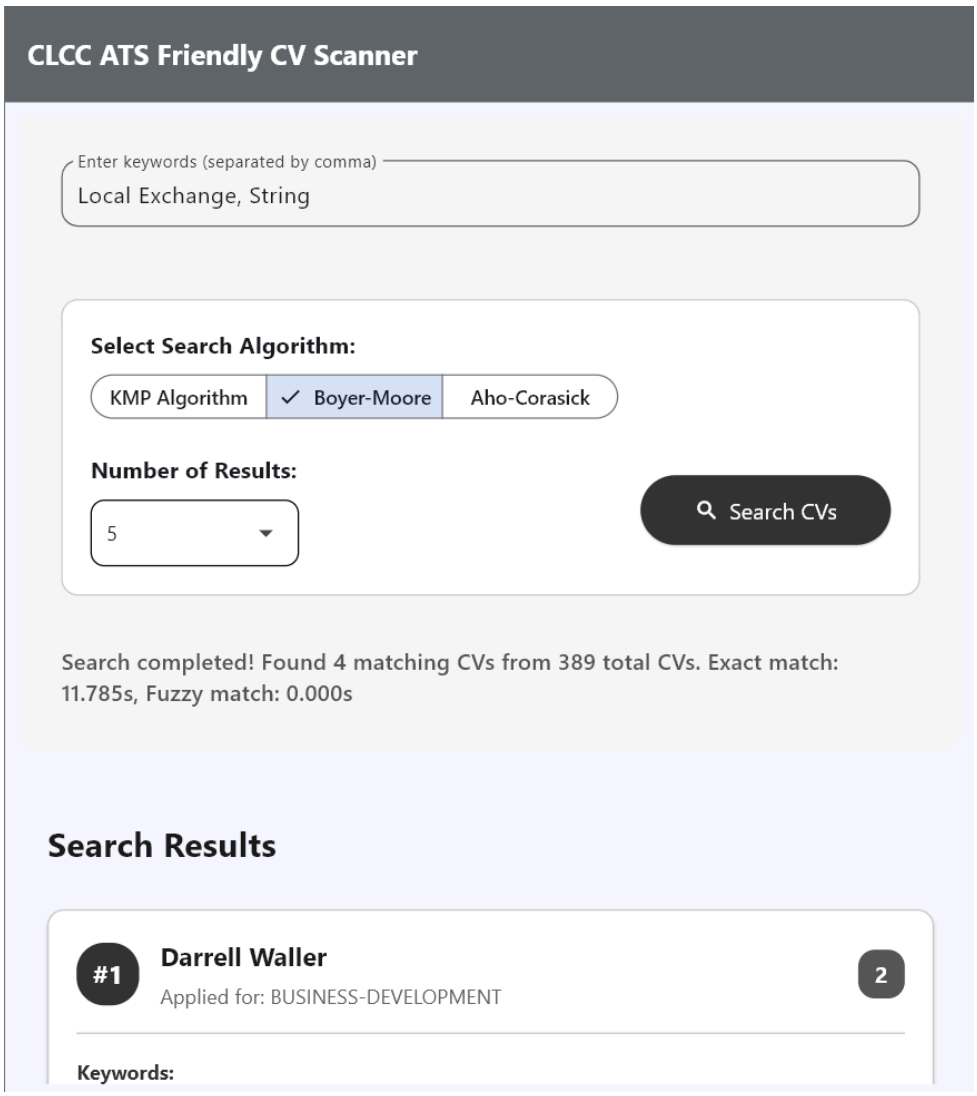
4.3.3 Pengujian 3

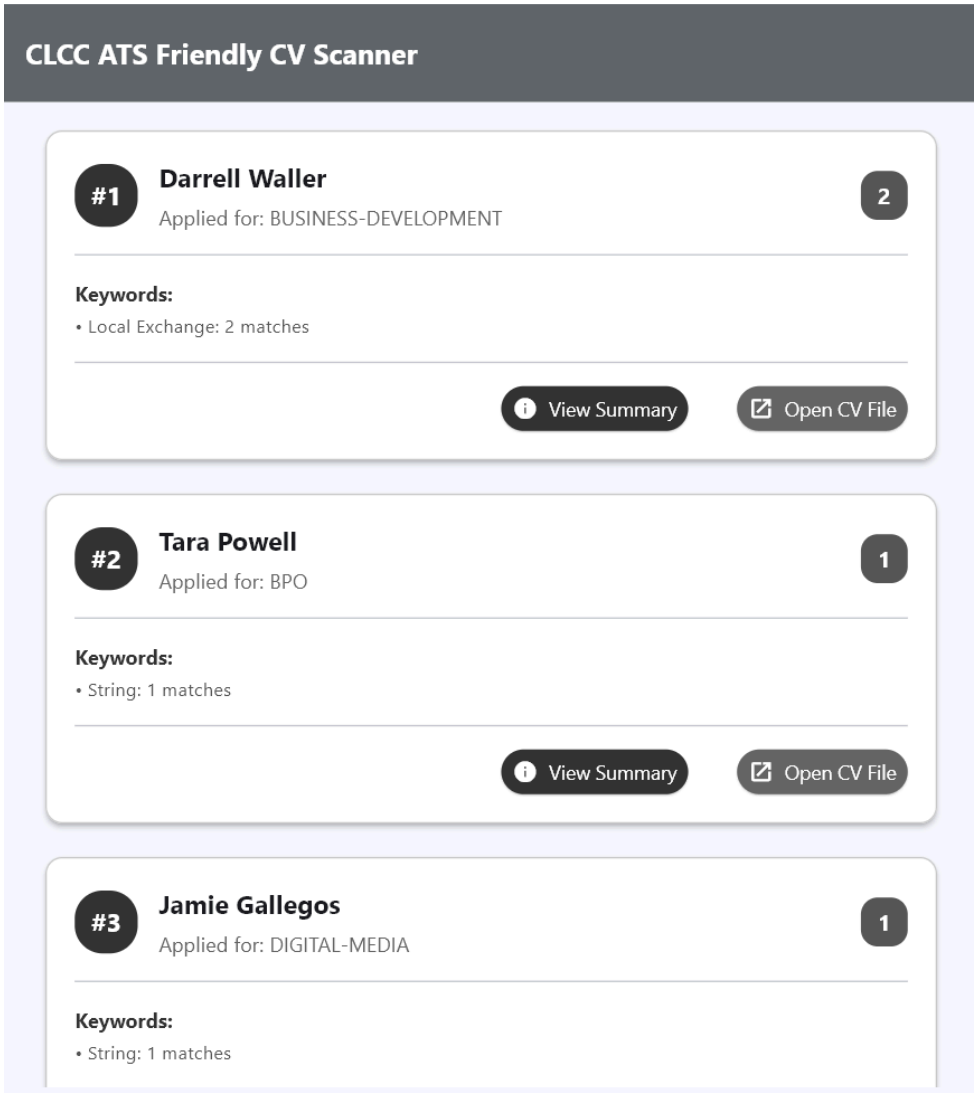
Query = Local Exchange, String

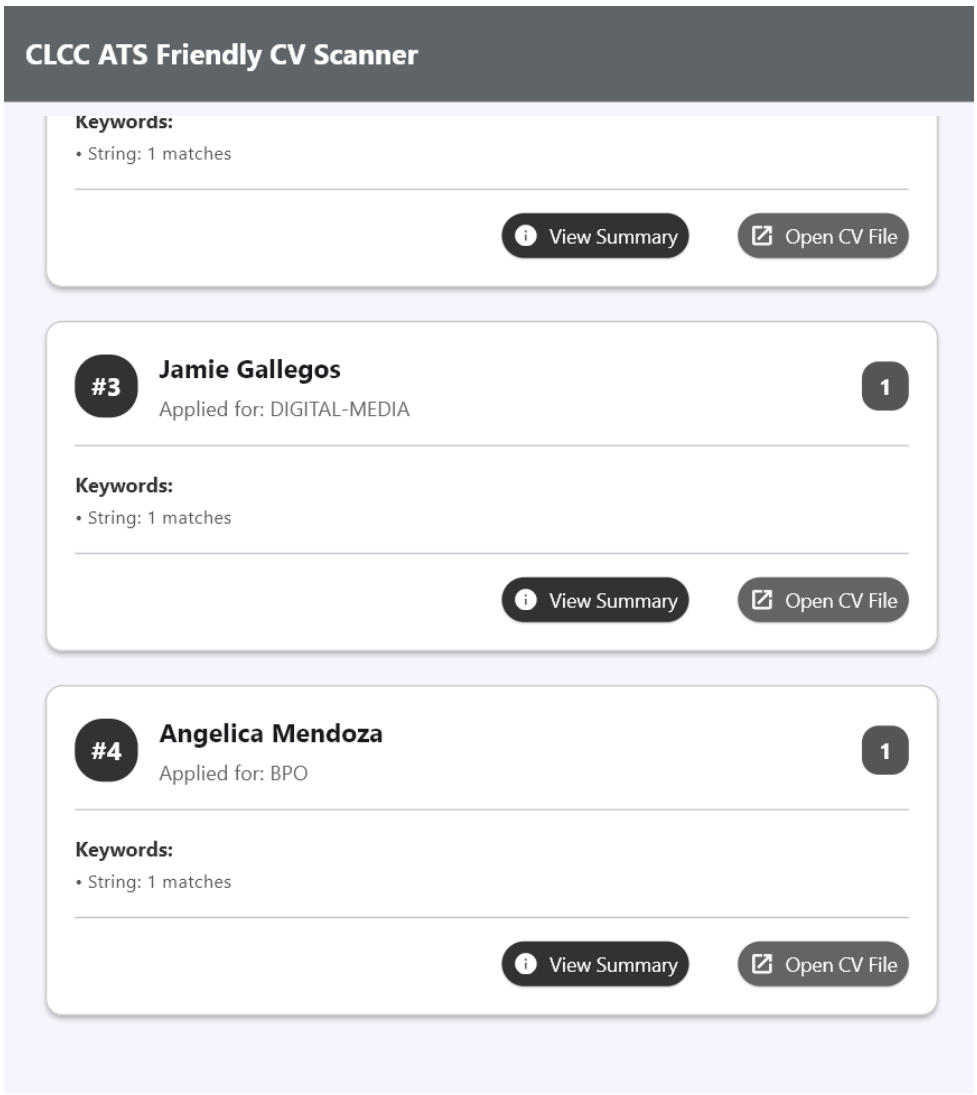
Algoritma	Hasil
KMP	

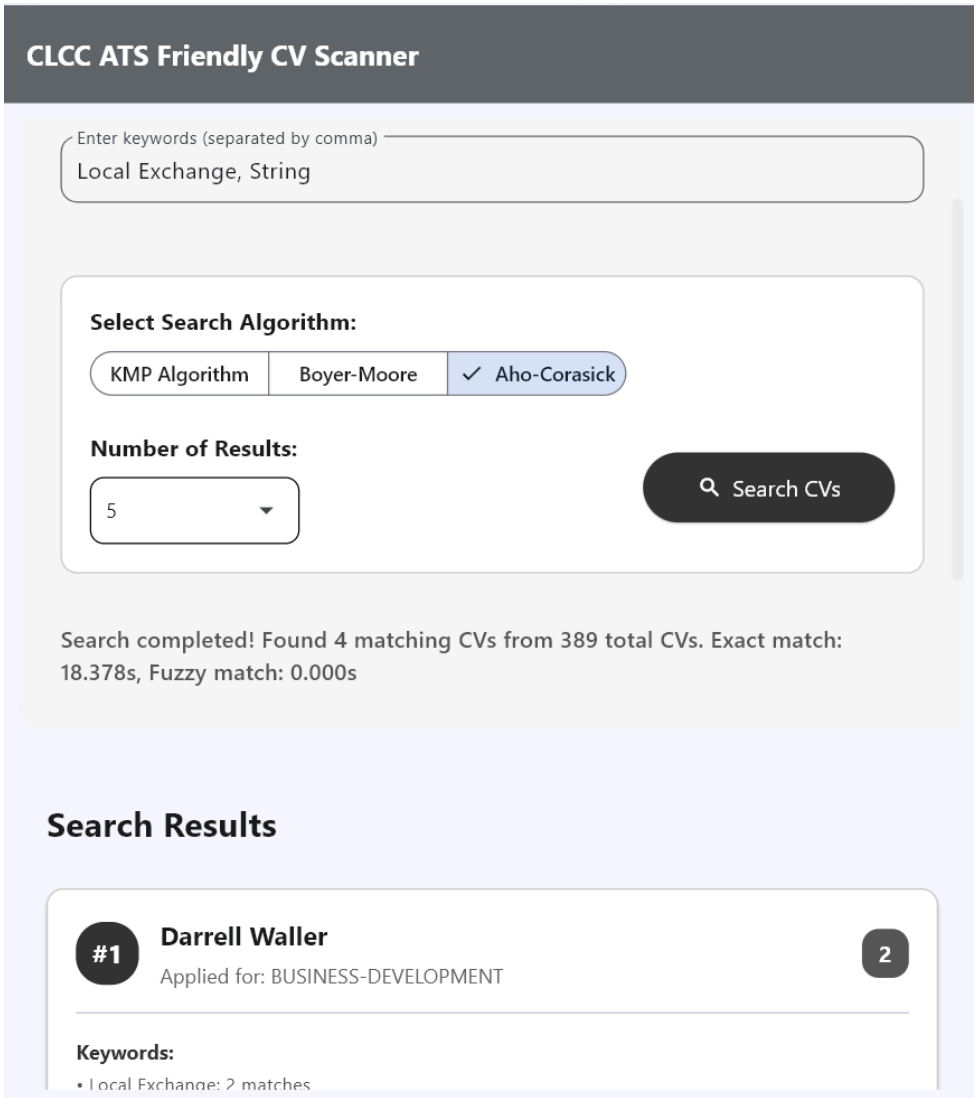
	 CLCC ATS Friendly CV Scanner Applied for: BUSINESS-DEVELOPMENT Keywords: • Local Exchange: 2 matches View Summary Open CV File #2 Tara Powell Applied for: BPO 1 Keywords: • String: 1 matches View Summary Open CV File #3 Jamie Gallegos Applied for: DIGITAL-MEDIA 1 Keywords: • String: 1 matches View Summary Open CV File

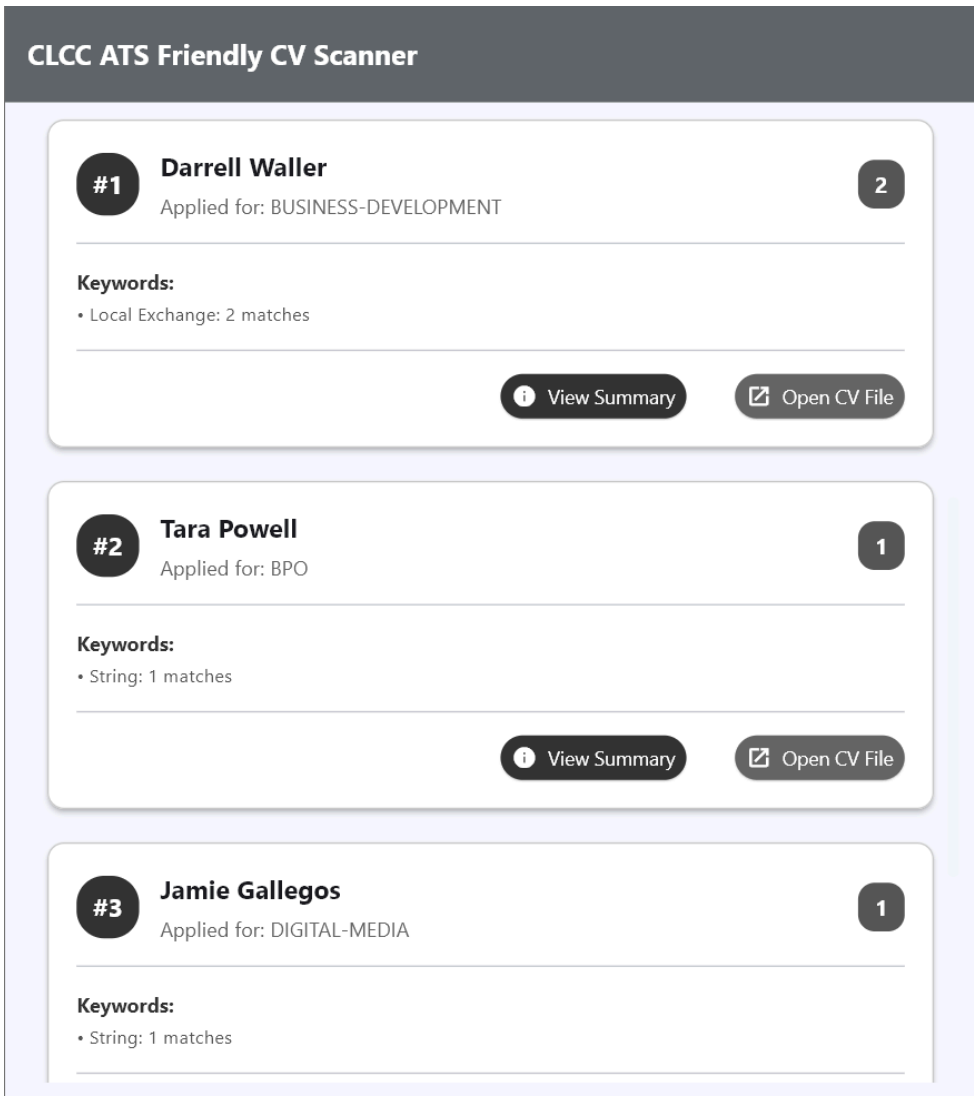

CLCC ATS Friendly CV Scanner
Keywords:
• String: 1 matches
View Summary
Open CV File
#3
Jamie Gallegos
Applied for: DIGITAL-MEDIA
1
Keywords:
• String: 1 matches
View Summary
Open CV File
#4
Angelica Mendoza
Applied for: BPO
1
Keywords:
• String: 1 matches
View Summary
Open CV File

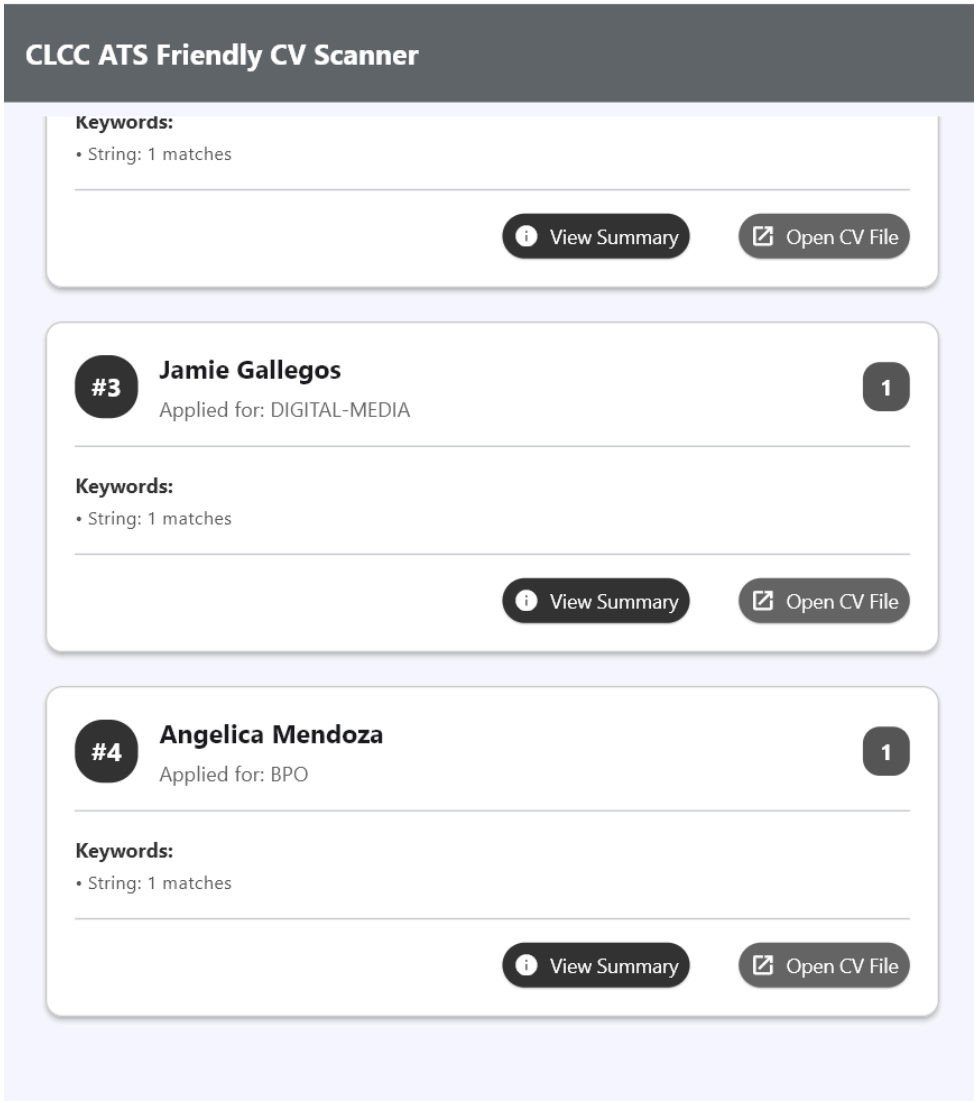
BM	 CLCC ATS Friendly CV Scanner Enter keywords (separated by comma) Local Exchange, String Select Search Algorithm: KMP Algorithm Boyer-Moore Aho-Corasick Number of Results: 5 Search CVs Search completed! Found 4 matching CVs from 389 total CVs. Exact match: 11.785s, Fuzzy match: 0.000s Search Results #1 Darrell Waller Applied for: BUSINESS-DEVELOPMENT 2 Keywords:


CLCC ATS Friendly CV Scanner
#1
Darrell Waller
Applied for: BUSINESS-DEVELOPMENT
2
Keywords:
• Local Exchange: 2 matches
View Summary
Open CV File
#2
Tara Powell
Applied for: BPO
1
Keywords:
• String: 1 matches
View Summary
Open CV File
#3
Jamie Gallegos
Applied for: DIGITAL-MEDIA
1
Keywords:
• String: 1 matches


CLCC ATS Friendly CV Scanner
Keywords:
• String: 1 matches
View Summary
Open CV File
#3
Jamie Gallegos
Applied for: DIGITAL-MEDIA
1
Keywords:
• String: 1 matches
View Summary
Open CV File
#4
Angelica Mendoza
Applied for: BPO
1
Keywords:
• String: 1 matches
View Summary
Open CV File

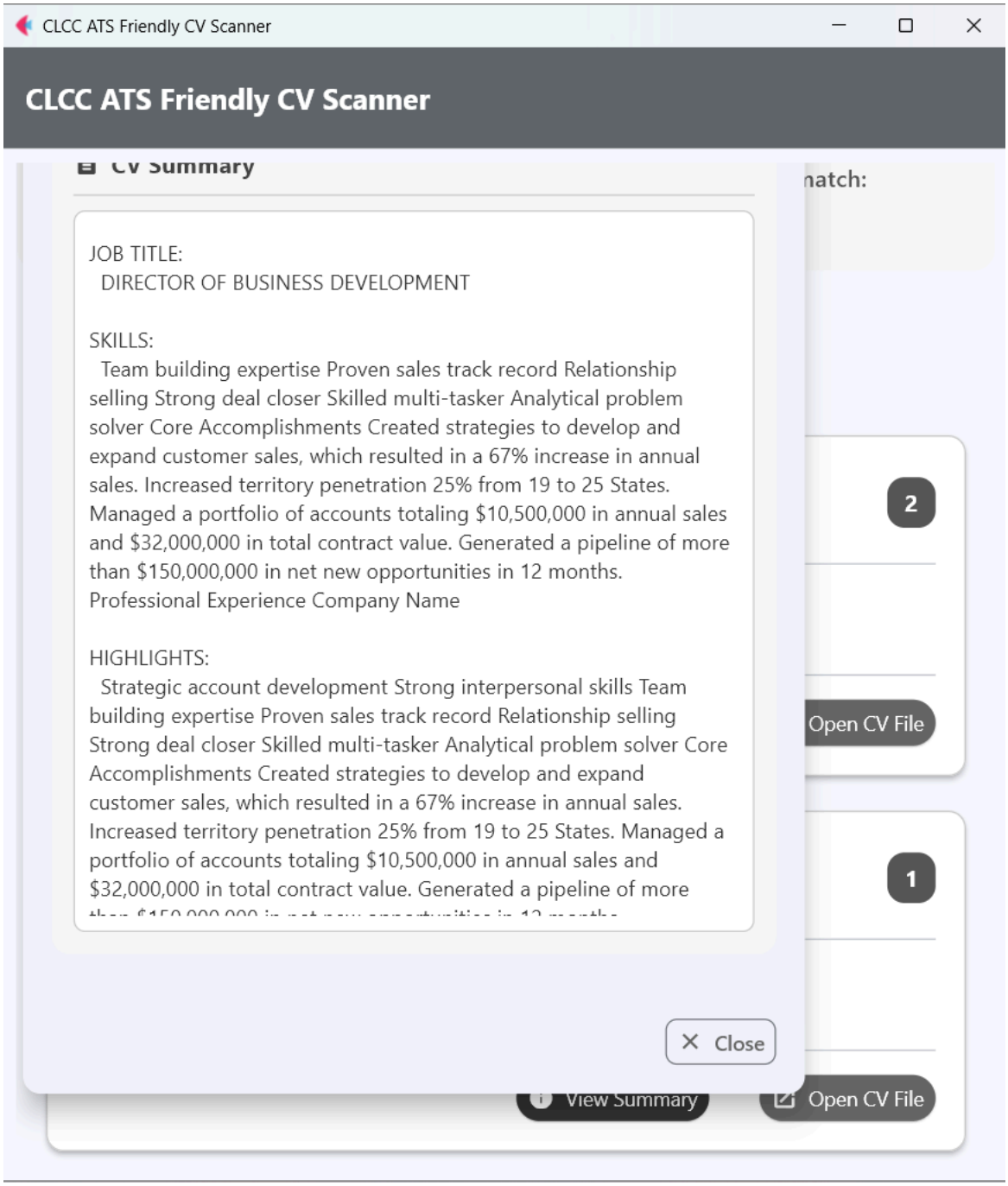
Aho-C	 CLCC ATS Friendly CV Scanner Enter keywords (separated by comma) Local Exchange, String Select Search Algorithm: KMP Algorithm Boyer-Moore Aho-Corasick Number of Results: 5 Search CVs Search completed! Found 4 matching CVs from 389 total CVs. Exact match: 18.378s, Fuzzy match: 0.000s Search Results #1 Darrell Waller Applied for: BUSINESS-DEVELOPMENT 2 Keywords: • Local Exchange: 2 matches


CLCC ATS Friendly CV Scanner
#1
Darrell Waller
Applied for: BUSINESS-DEVELOPMENT
2
Keywords:
• Local Exchange: 2 matches
View Summary
Open CV File
#2
Tara Powell
Applied for: BPO
1
Keywords:
• String: 1 matches
View Summary
Open CV File
#3
Jamie Gallegos
Applied for: DIGITAL-MEDIA
1
Keywords:
• String: 1 matches


CLCC ATS Friendly CV Scanner
Keywords:
• String: 1 matches
View Summary
Open CV File
#3
Jamie Gallegos
Applied for: DIGITAL-MEDIA
1
Keywords:
• String: 1 matches
View Summary
Open CV File
#4
Angelica Mendoza
Applied for: BPO
1
Keywords:
• String: 1 matches
View Summary
Open CV File

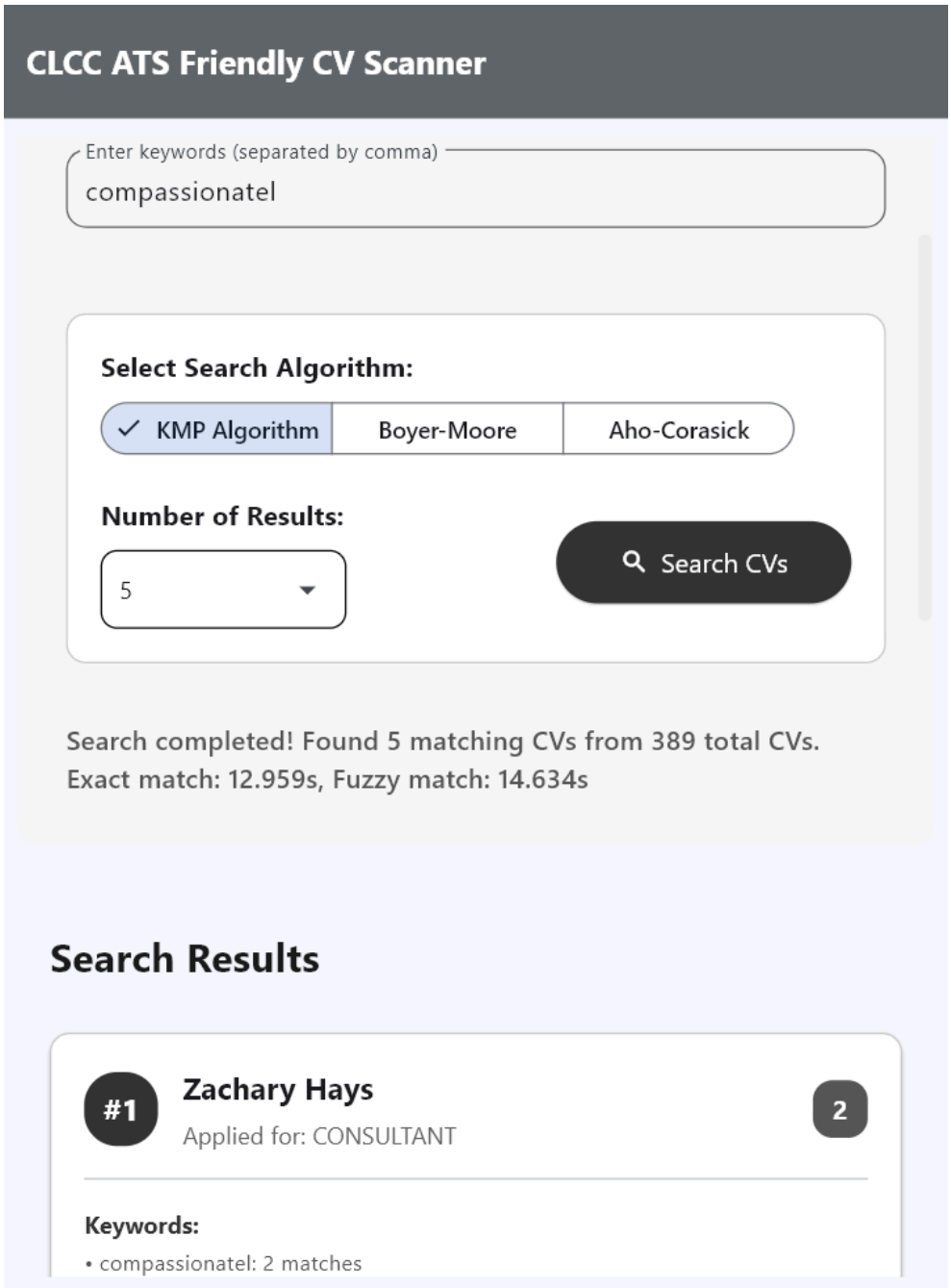
Hasil CV Summary:

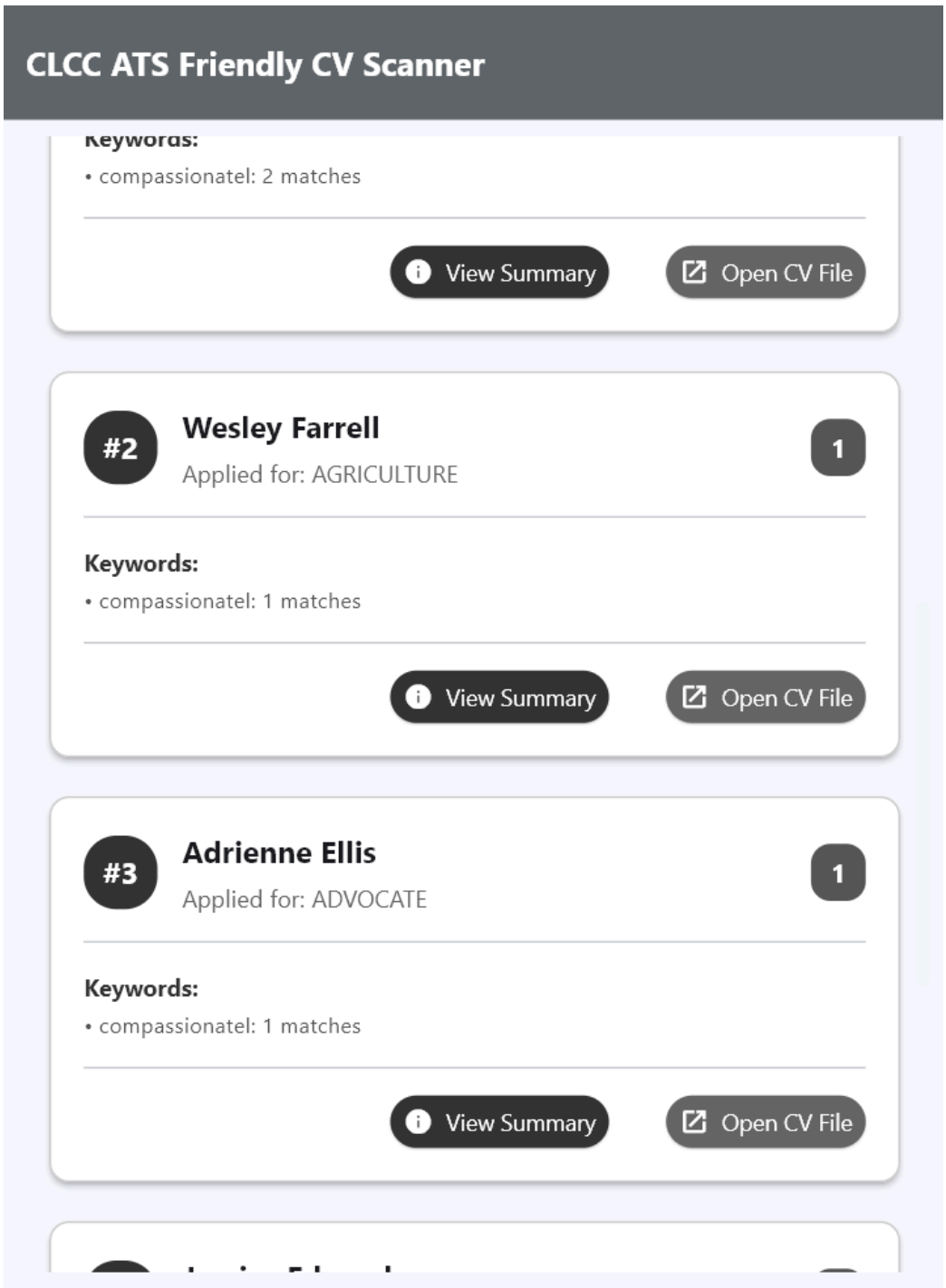
Darrel Waller (TOP 1):

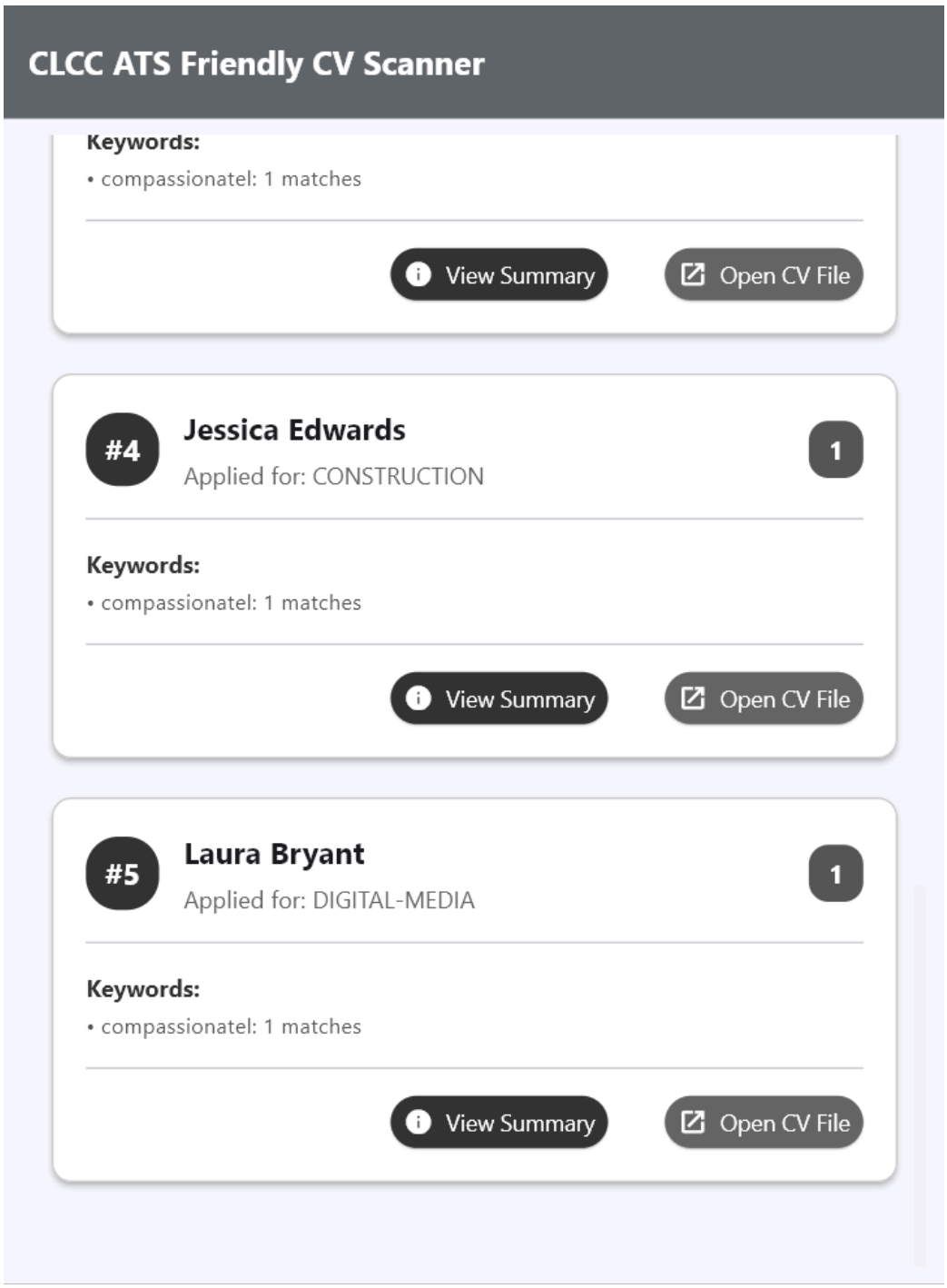


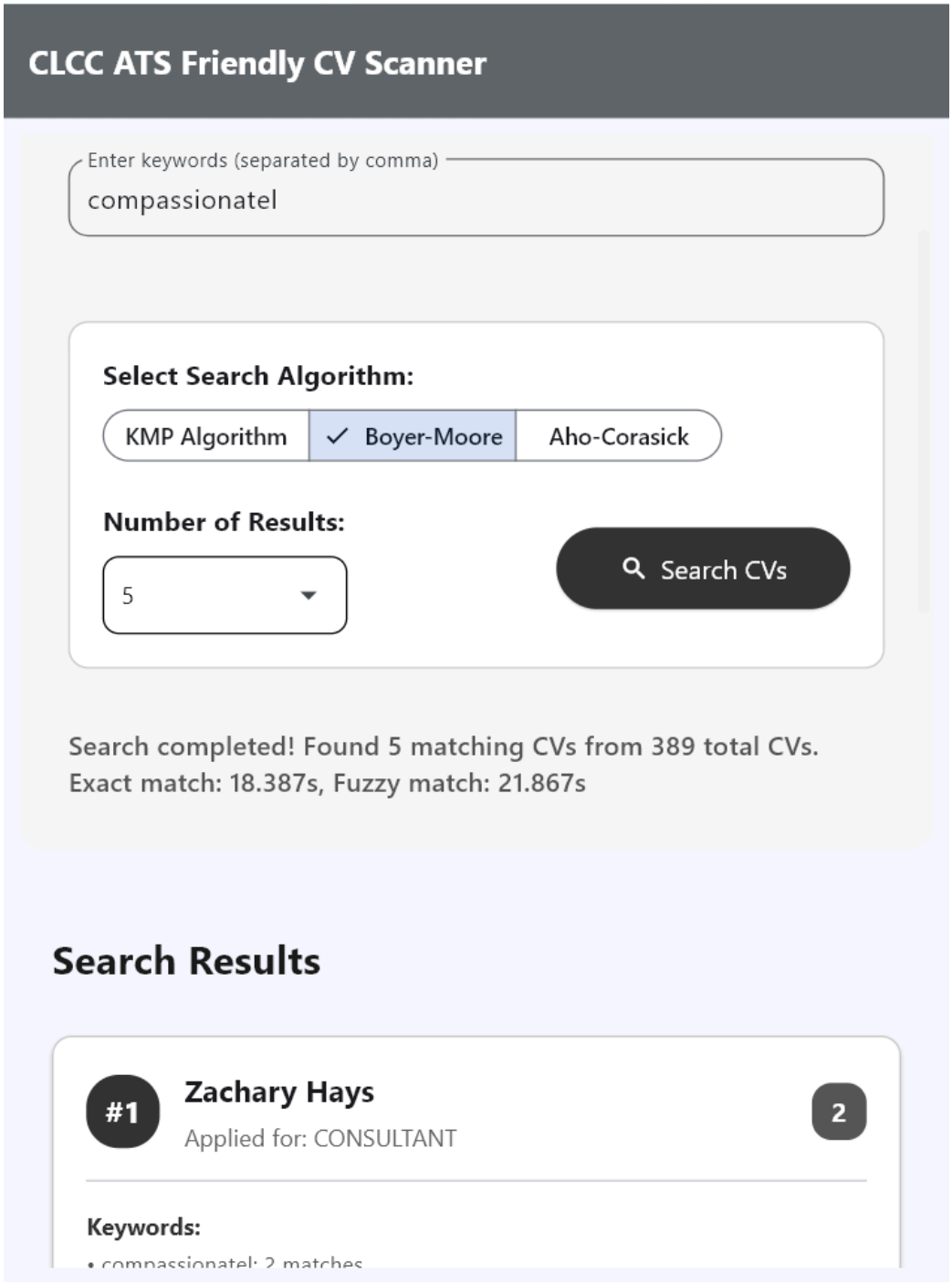
4.3.4 Pengujian 4

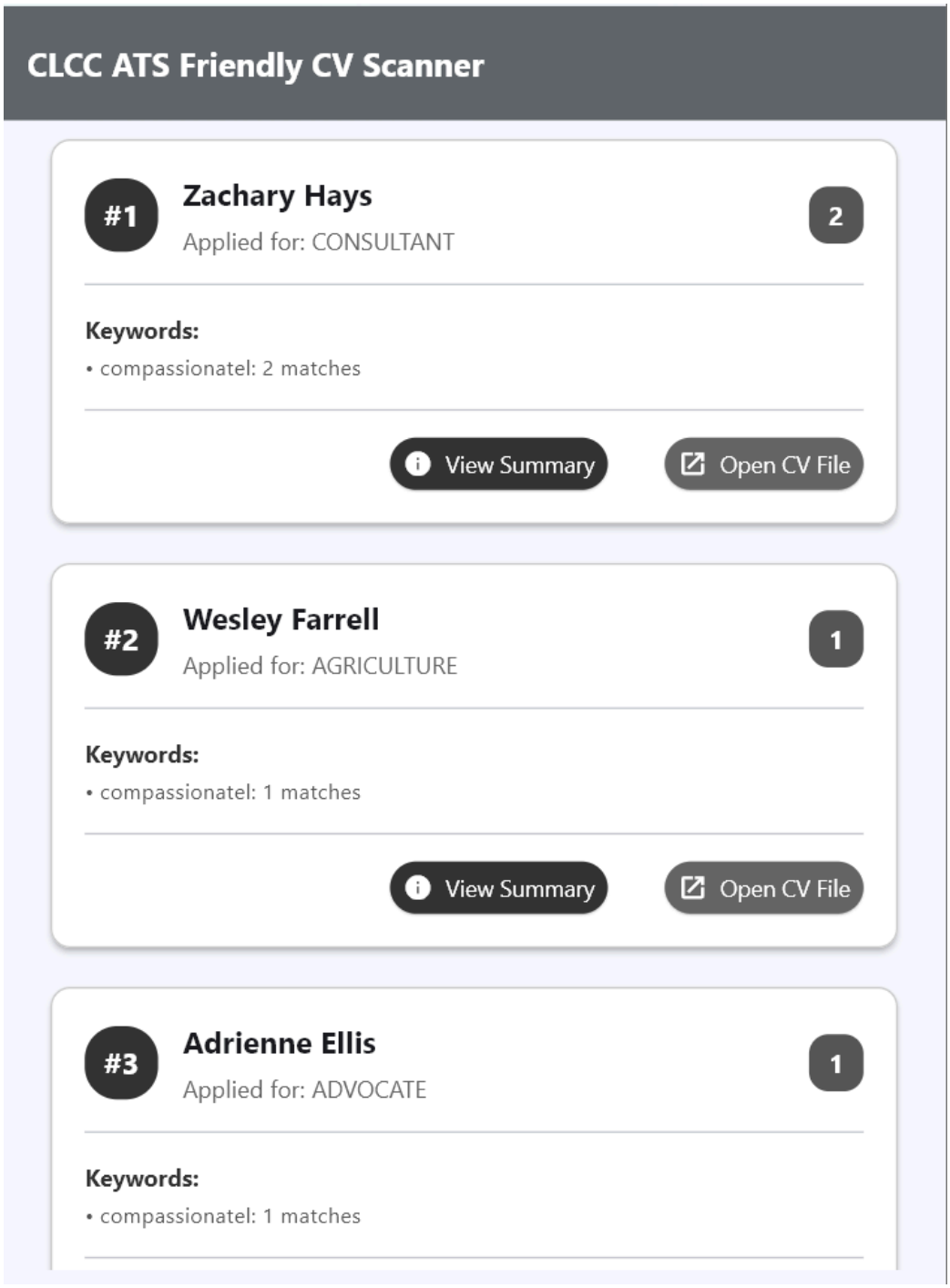
Query: compassionatel

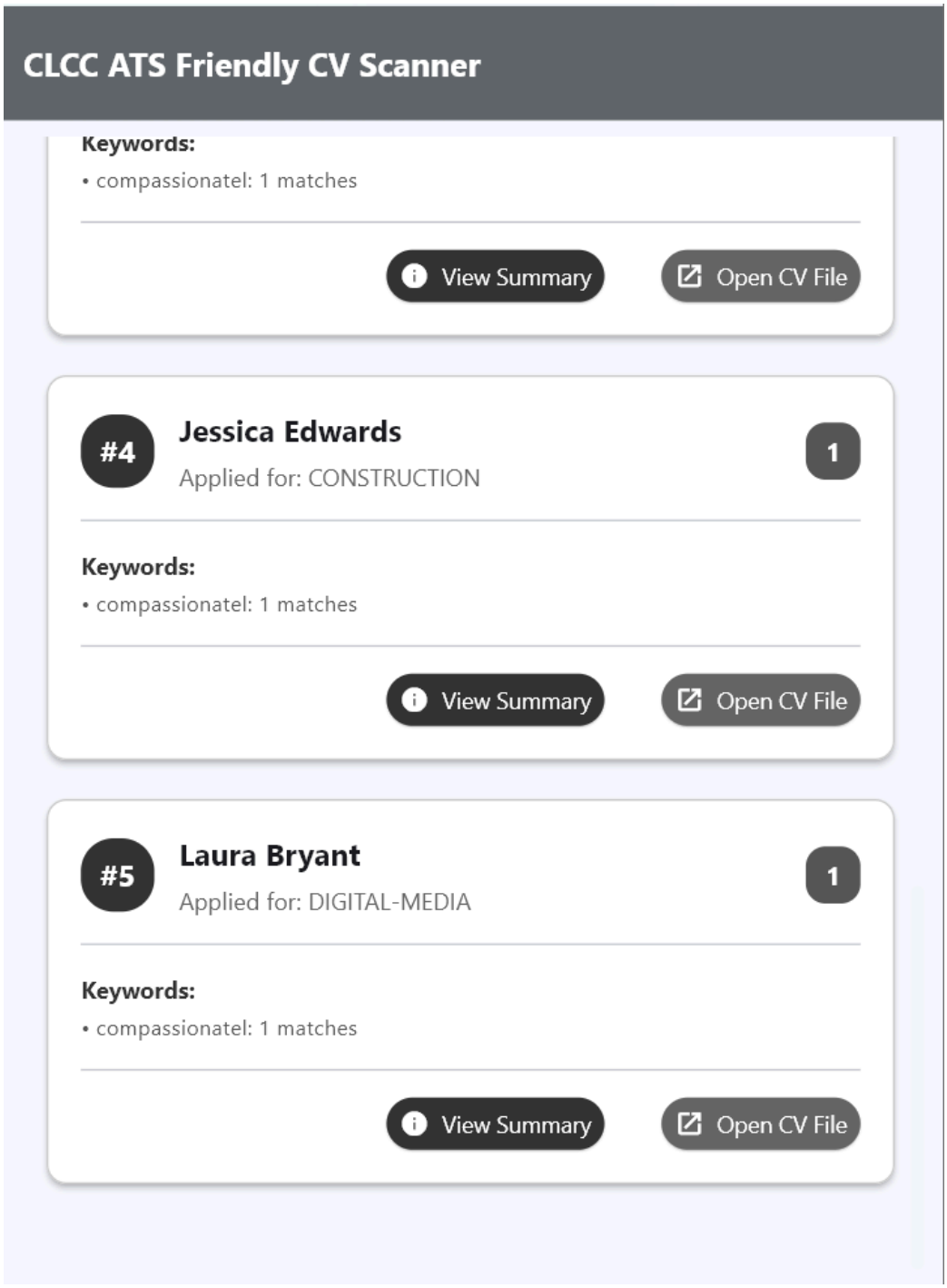
Algoritma	Hasil
KMP	

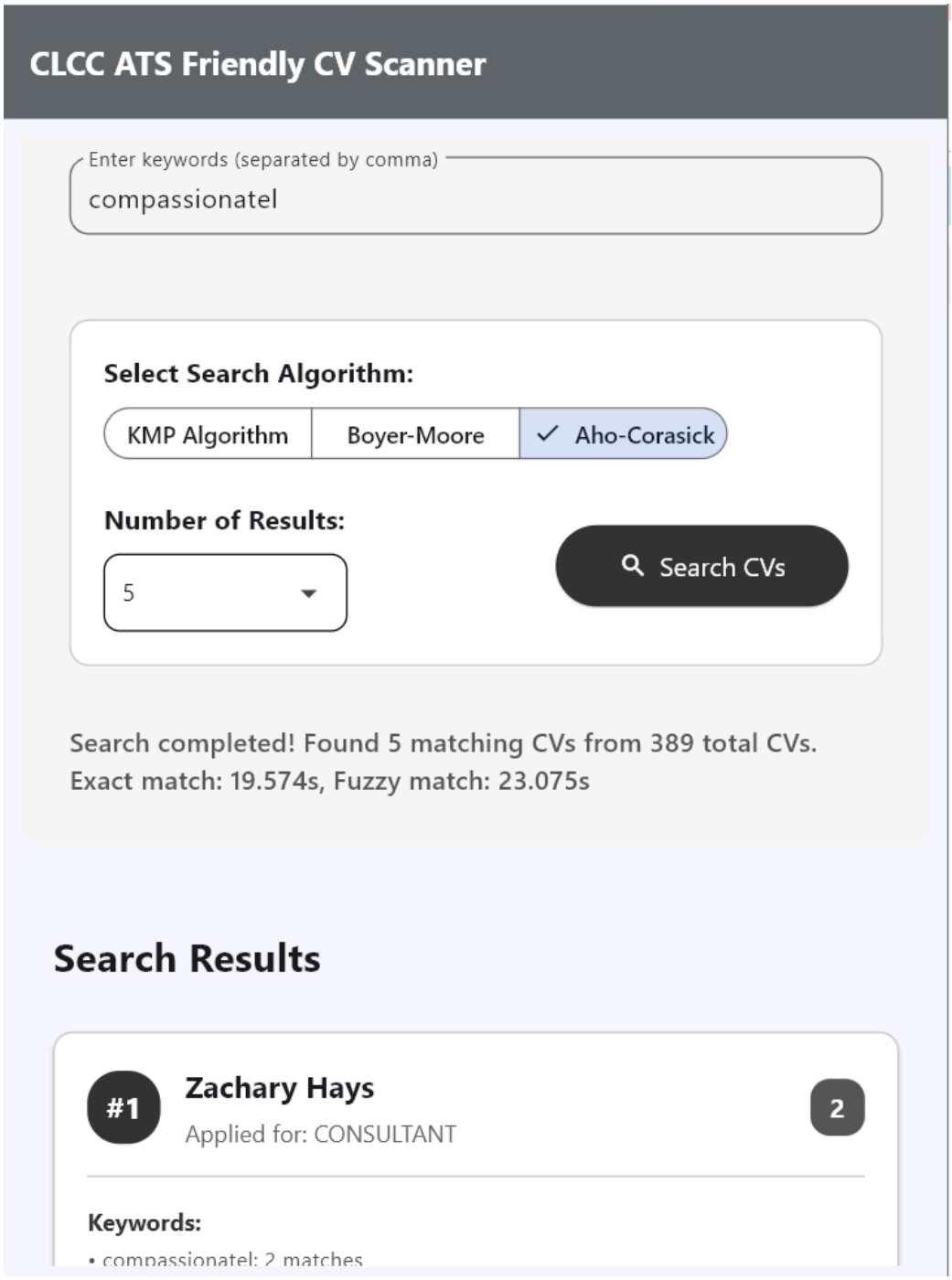

CLCC ATS Friendly CV Scanner
Keywords:
• compassionatel: 2 matches
View Summary
Open CV File
#2
Wesley Farrell
Applied for: AGRICULTURE
1
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File
#3
Adrienne Ellis
Applied for: ADVOCATE
1
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File

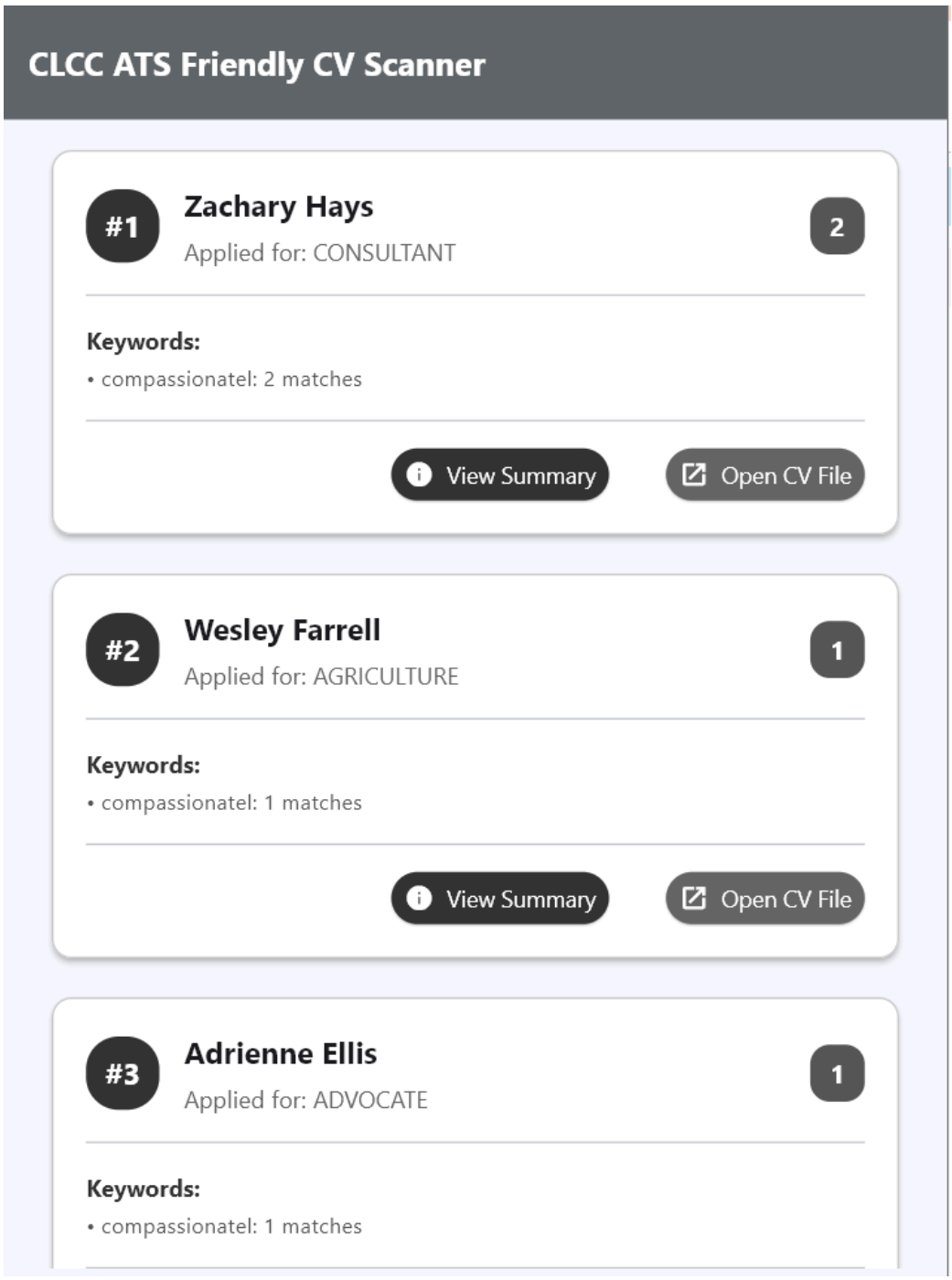

CLCC ATS Friendly CV Scanner
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File
#4
Jessica Edwards
Applied for: CONSTRUCTION
1
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File
#5
Laura Bryant
Applied for: DIGITAL-MEDIA
1
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File

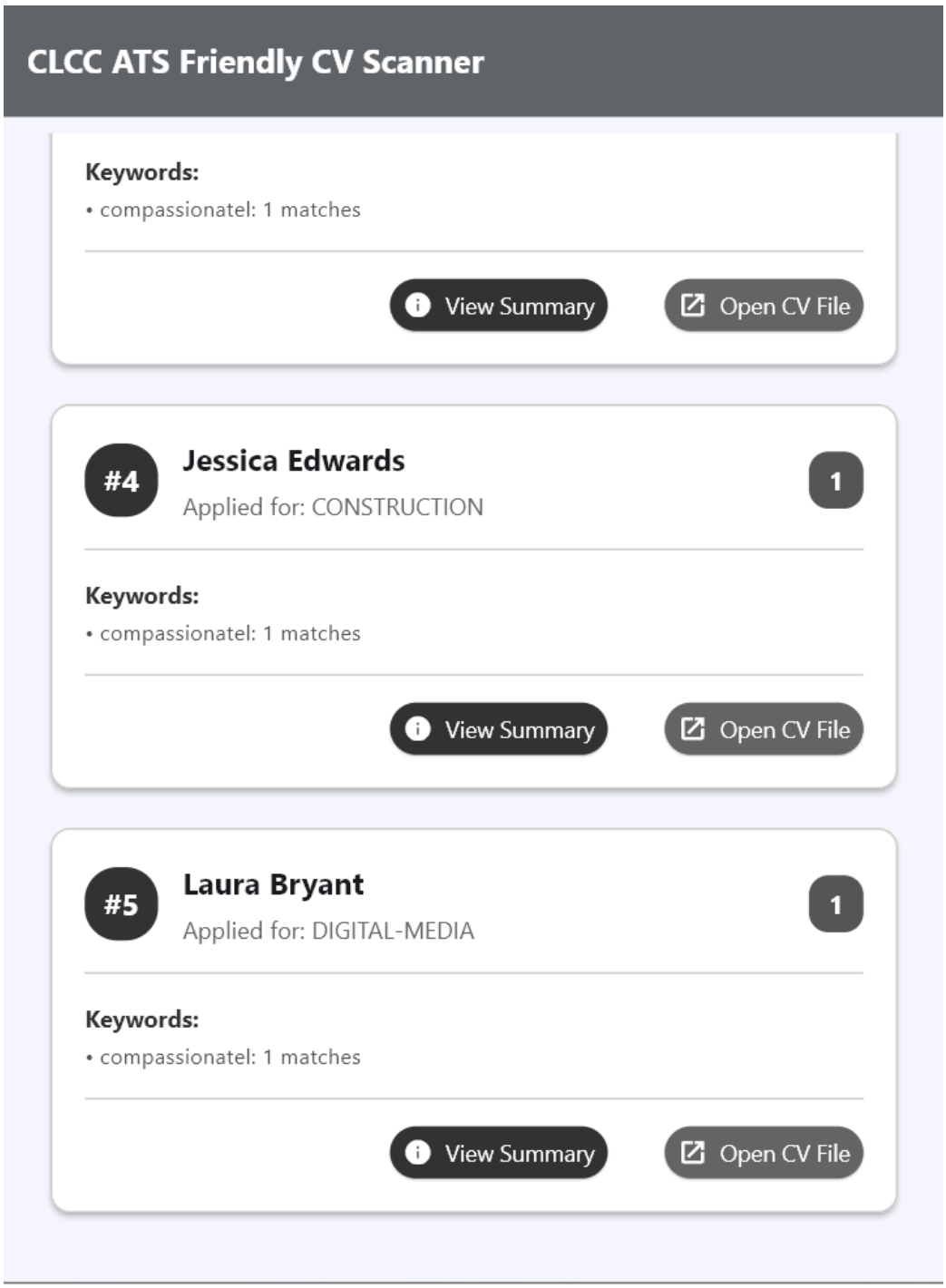
BM	 CLCC ATS Friendly CV Scanner Enter keywords (separated by comma) compassionatel Select Search Algorithm: KMP Algorithm Boyer-Moore Aho-Corasick Number of Results: 5 Search CVs Search completed! Found 5 matching CVs from 389 total CVs. Exact match: 18.387s, Fuzzy match: 21.867s Search Results #1 Zachary Hays Applied for: CONSULTANT 2 Keywords:


CLCC ATS Friendly CV Scanner
#1
Zachary Hays
Applied for: CONSULTANT
2
Keywords:
• compassionatel: 2 matches
View Summary
Open CV File
#2
Wesley Farrell
Applied for: AGRICULTURE
1
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File
#3
Adrienne Ellis
Applied for: ADVOCATE
1
Keywords:
• compassionatel: 1 matches


CLCC ATS Friendly CV Scanner
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File
#4
Jessica Edwards
Applied for: CONSTRUCTION
1
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File
#5
Laura Bryant
Applied for: DIGITAL-MEDIA
1
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File

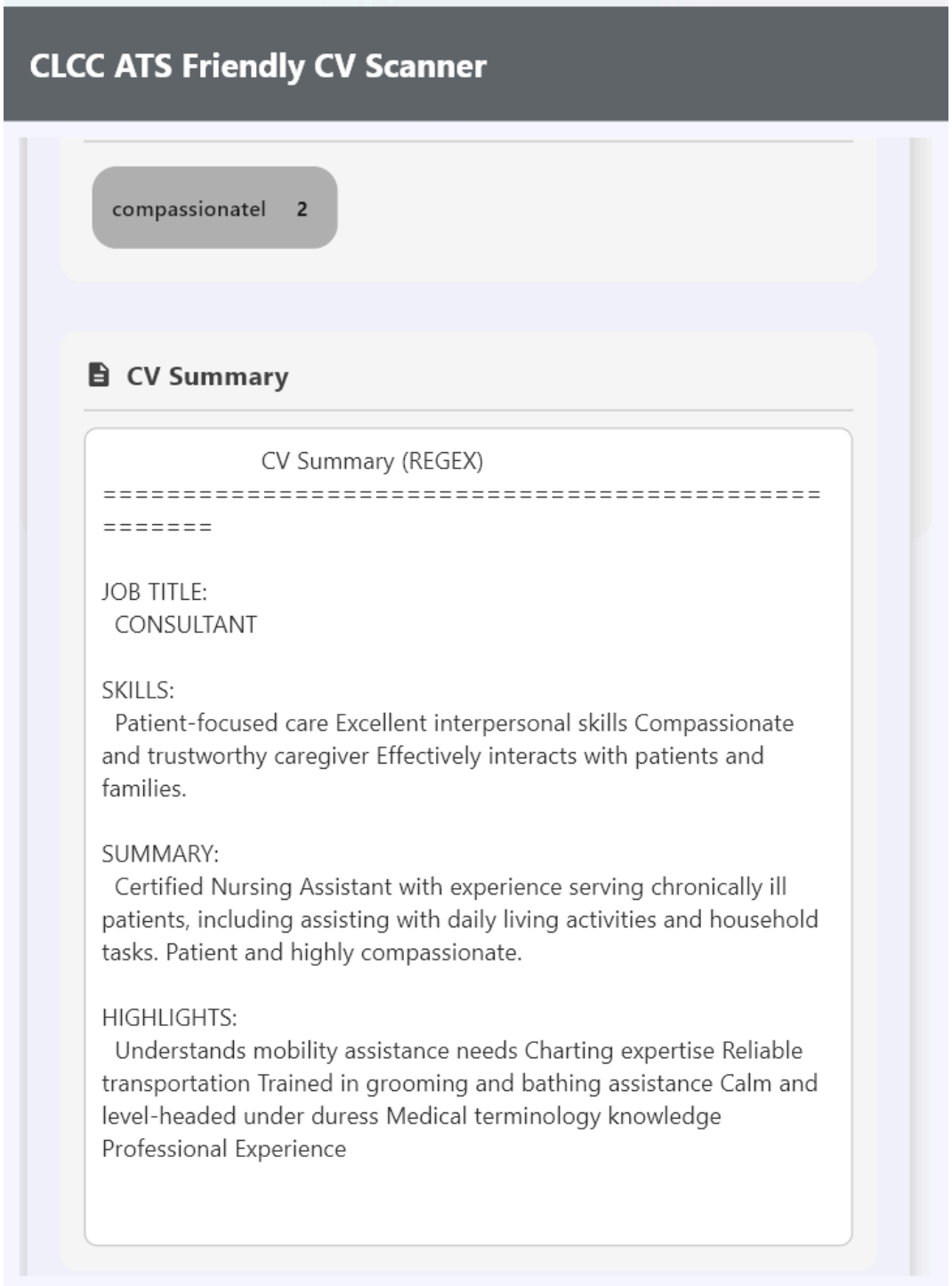
Aho-C	 CLCC ATS Friendly CV Scanner Enter keywords (separated by comma) compassionatel Select Search Algorithm: KMP Algorithm Boyer-Moore Aho-Corasick Number of Results: 5 Search CVs Search completed! Found 5 matching CVs from 389 total CVs. Exact match: 19.574s, Fuzzy match: 23.075s Search Results #1 Zachary Hays Applied for: CONSULTANT 2 Keywords: • compassionatel: 2 matches


CLCC ATS Friendly CV Scanner
#1
Zachary Hays
Applied for: CONSULTANT
2
Keywords:
• compassionatel: 2 matches
View Summary
Open CV File
#2
Wesley Farrell
Applied for: AGRICULTURE
1
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File
#3
Adrienne Ellis
Applied for: ADVOCATE
1
Keywords:
• compassionatel: 1 matches


CLCC ATS Friendly CV Scanner
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File
#4
Jessica Edwards
Applied for: CONSTRUCTION
1
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File
#5
Laura Bryant
Applied for: DIGITAL-MEDIA
1
Keywords:
• compassionatel: 1 matches
View Summary
Open CV File

Hasil CV Summary:

Zachary Hays (TOP 1):



4.4 Analisis Hasil Pengujian

Hasil pengujian sistem ATS dilakukan terhadap empat skenario query kata kunci yang berbeda dengan menggunakan tiga algoritma pencocokan string utama: Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), dan Aho-Corasick (AC). Masing-masing pengujian bertujuan mengevaluasi keakuratan dan kecepatan pencarian berdasarkan karakteristik algoritma yang digunakan serta efektivitas dalam menangani dokumen CV PDF yang tidak terstruktur.

4.4.1 Test 1: Query = Python, React, SQL

Pengujian ini merepresentasikan skenario pencarian multi-keyword dengan panjang sedang, umum ditemukan dalam pencarian developer. Hasil terbaik diperoleh oleh algoritma Aho-Corasick, yang secara konsisten dapat menemukan semua keyword dalam satu traversal, menjadikannya paling efisien dari segi waktu eksekusi. Sementara itu, KMP dan BM bekerja secara serial terhadap setiap kata kunci, yang membuat waktu pencarian relatif lebih lama, meskipun tetap menghasilkan hasil yang benar. Top kandidat yang teridentifikasi adalah Jennifer Nichols, menunjukkan keberhasilan sistem dalam mengurutkan berdasarkan jumlah kecocokan keyword.

4.4.2 Test 2: Query = a, merchandise

Query ini mengandung satu kata kunci yang sangat umum (“a”) dan satu spesifik (“merchandise”), menguji sistem dalam kondisi keyword ambigu atau terlalu pendek. Semua algoritma menghasilkan hasil yang identik (menemukan Randall Johnson), namun perbedaan utama muncul pada efisiensi:

- BM melakukan banyak lompatan karena pola pendek “a” sering ditemukan, menyebabkan evaluasi lebih boros meski efisien dalam string panjang.
- KMP unggul dalam stabilitas dan tetap linear meski keyword sangat pendek.
- Aho-Corasick tetap menjadi pilihan terbaik untuk kombinasi kata kunci pendek dan panjang karena menghindari proses per keyword.

4.4.3 Test 3: Query = Local Exchange, String

Kombinasi kata kunci dengan spasi dan huruf kapital diuji di sini. Sistem telah menangani normalisasi input (lowercase), sehingga semua algoritma berhasil menemukan kandidat yang sesuai (Darrel Waller). Namun, pencarian frasa seperti "Local Exchange" menjadi tantangan:

- BM dan KMP tetap efektif karena pencarian exact string tetap terjaga.
- Aho-Corasick bekerja sangat baik, terutama karena mampu mengidentifikasi kedua kata sekaligus meski dipisah dalam teks.

Pengujian ini menunjukkan pentingnya preprocessing input teks, dan keunggulan AC dalam pencarian konteks ganda.

4.4.4 Test 4: Query = compassionatel

Query ini merupakan typo dari “compassionate”, digunakan untuk menguji fallback fuzzy matching. Pada semua algoritma, awalnya tidak ditemukan exact match karena kata ini tidak eksis dalam bentuk persis. Namun, sistem memicu Levenshtein Distance sebagai fallback untuk mendeteksi kecocokan fuzzy. Hasil menunjukkan sistem mampu mengidentifikasi Zachary Hays sebagai kandidat, yang memiliki kata “compassionate” dalam CV-nya, walau ada typo pada input. Hal ini menegaskan bahwa integrasi fuzzy matching berhasil memperluas cakupan pencarian tanpa mengorbankan akurasi, terutama saat keyword mengandung kesalahan pengetikan.

BAB V
KESIMPULAN, SARAN, DAN REFLEKSI

5.1 Kesimpulan

Pada Tugas Besar 3 Strategi Algoritma ini kami telah berhasil mengimplementasikan dan mengevaluasi sebuah sistem *Applicant Tracking System* (ATS) berbasis CV digital. Sistem ini memanfaatkan beberapa algoritma pencocokan string, yaitu Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), dan Aho-Corasick, untuk melakukan pencarian kata kunci secara *exact match*. Setiap algoritma berhasil diimplementasikan sesuai dengan fungsinya, di mana pengguna dapat memilih salah satu untuk digunakan dalam proses pencarian. Selain itu, sistem ini juga didukung oleh algoritma Levenshtein Distance untuk *fuzzy matching* dan Regular Expression (Regex) untuk ekstraksi informasi penting dari CV.

Pengujian dilakukan dengan mengonversi setiap dokumen CV dalam format PDF menjadi satu string panjang untuk dianalisis. Algoritma Boyer-Moore secara umum menunjukkan keunggulan dalam kecepatan eksekusi pada sebagian besar kasus uji, terutama saat mencari kata kunci tunggal dalam teks yang panjang. Di sisi lain, algoritma Knuth-Morris-Pratt menunjukkan performa yang sangat stabil dan efisien. Sebagai implementasi bonus, algoritma Aho-Corasick memberikan keunggulan signifikan sebagai metode pencarian *multi-pattern matching* yang efisien. Dengan kemampuannya mencocokkan seluruh daftar kata kunci dalam satu kali proses traversal teks, algoritma ini menjadi lebih cepat dibandingkan pendekatan konvensional yang menjalankan KMP atau BM secara berulang untuk setiap kata kunci. Sementara itu, implementasi Levenshtein Distance terbukti efektif dalam menangani kesalahan pengetikan (*typo*) , dan Regular Expression berhasil mengekstraksi data penting seperti keahlian dan riwayat pendidikan secara akurat.

Berdasarkan keseluruhan pengujian dan analisis, sistem ATS yang dikembangkan telah berhasil memenuhi tujuan perancangan sistem. Ketiga algoritma *exact matching* yaitu KMP, BM, dan Aho-Corasick telah terbukti menjadi solusi yang andal, di mana Aho-Corasick menawarkan efisiensi tertinggi untuk pencarian *multi-keywords*. Di sisi lain, KMP menunjukkan stabilitas pada pencarian kata kunci yang sangat pendek, sementara BM tetap efektif untuk pencarian

frasa. Kombinasi dari *exact matching*, *fuzzy matching*, dan ekstraksi Regex menunjukkan sebuah pendekatan yang komprehensif dan kuat untuk tantangan penyaringan CV. Dengan demikian, sistem yang dibangun merupakan solusi yang optimal dan efisien untuk mengotomasikan proses rekrutmen awal.

5.2 Saran

Dalam proses pengerjaan Tugas Besar 3 ini, terdapat beberapa saran yang akan kami perhatikan untuk dikembangkan kedepannya, antara lain:

1. Sebaiknya mengeksplorasi algoritma peringkat (*ranking*) yang lebih kompleks karena sistem saat ini hanya mengurutkan berdasarkan jumlah total *keywords* yang cocok, misal dapat mencoba TF-IDF (*Term Frequency-Inverse Document Frequency*) yang memberikan bobot berbeda untuk setiap *keyword* berdasarkan relevansinya di dalam satu CV dan di seluruh dataset sehingga menghasilkan urutan kandidat yang lebih relevan.
2. Sebaiknya UI/UX dikembangkan lebih lanjut agar lebih interaktif dan menarik, misalnya menambahkan fitur *drag-and-drop* untuk menggugah file CV serta menyajikan hasil pencocokan dalam bentuk visualisasi data yang lebih informatif, tidak hanya berupa daftar *cards*.

5.3 Refleksi

Pengerjaan tugas besar ini memberi kami wawasan yang lebih mendalam mengenai penerapan algoritma *string matching* dan *text processing* dalam konteks dunia nyata. Kami belajar untuk tidak hanya memahami teori di balik algoritma Knuth-Morris-Pratt (KMP), Boyer-Moore (BM), Aro-Corasick, dan Levenshtein Distance, serta *Regular Expression* (Regex), namun juga bagaimana mengimplementasikannya dalam sebuah sistem yang fungsional, lengkap dengan *interface* pengguna dan integrasi basis data.

Tantangan seperti proses ekstraksi teks dari berbagai format dokumen CV PDF, mengelola integrasi dengan basis data MySQL, merancang *interface* pengguna yang intuitif, serta mengimplementasikan algoritma enkripsi dari awal tanpa pustaka bawaan untuk fitur bonus menjadi pengalaman teknis yang sangat berharga bagi kami. Ke depannya, kami akan lebih

memperhatikan modularitas dan keterbacaan kode serta meningkatkan manajemen waktu pengerjaan agar proses pengembangan berjalan lebih efisien dan terstruktur.

LAMPIRAN

a. *Repository* GitHub : https://github.com/0xNathaniel/Tubes3_CLCC

b. Video : https://youtu.be/vDMZrhda0vk

c. Tabel :

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

Geeks for Geeks (2025). Aho-Corasick Algorithm for Pattern Searching. [Online] Tersedia: https://www.geeksforgeeks.org/dsa/aho-corasick-algorithm-pattern-searching/ [Diakses 6 Juni 2025].

Geeks for Geeks (2025). Boyer Moore Algorithm for Pattern Searching. [Online] Tersedia: https://www.geeksforgeeks.org/dsa/boyer-moore-algorithm-for-pattern-searching/ [Diakses 6 Juni 2025].

Geeks for Geeks (2025). KMP Algorithm for Pattern Searching. [Online] Tersedia: https://www.geeksforgeeks.org/dsa/kmp-algorithm-for-pattern-searching/ [Diakses 6 Juni 2025].

Khodra, Masayu Leylia. (2025). String Matching dengan Regular Expression. [Online] Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/24-String-Matching-dengan-Regex-(2025).pdf [Diakses 6 Juni 2025].

Munir, R. (2025). Pencocokan String (*String/Pattern Matching*). [Online] Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf [Diakses 6 Juni 2025].