Implementasi *Approximate Pattern Matching* pada Pencarian Teks Berbasis Algoritma Komputasi *Levenshtein Distance*

Penerapan pada Aplikasi Berbasis Web

Raffi Zulvian Muzhaffar 13519003
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13519003@std.stei.itb.ac.id

***Abstract*—Pencarian teks pada sebuah dokumen dengan *exact pattern matching* memiliki sebuah kekurangan, yaitu ketika pengguna tidak tahu pasti seperti apa persisnya teks yang ingin dicari. Dengan menggunakan *approximate pattern matching* pencarian dapat memberikan hasil yang lebih fleksibel. Pada makalah ini penulis melakukan penerapan *approximate pattern matching* dengan menggunakan algoritma *Levenshtein distance* untuk melakukan perhitungan nilai *edit distance* dari kata yang dijadikan kueri terhadap setiap kata pada dokumen. Varian yang digunakan untuk menyelesaikan permasalahan ini adalah varian dengan dua buah array yang memberikan optimasi dalam penggunaan ruang penyimpanan. Dengan menggunakan *framework* React, penulis mengimplementasikan konsep ini untuk membuat aplikasi berbasis web untuk melakukan pencarian teks berbasis *approximate pattern matching* dengan *Levenshtein distance algorithm*. Hasil implementasi yang dibuat menunjukan hasil yang baik, aplikasi yang dibuat mampu melakukan pencarian dengan toleransi kesalahan tertentu. Namun algoritma ini masih belum bisa menangani pencarian pada teks yang terlalu panjang dikarenakan lamanya waktu pemrosesan semakin bertambah berdasarkan *O*(*mn*).**

Kata kunci: Levenshtein distance, approximation pattern matching, algoritma, pencocokan pola;

I. Pendahuluan

Dalam berkegiatan digital sehari-hari, sangat umum dijumpai kondisi ketika seseorang butuh untuk dapat melakukan pencarian suatu kata atau frasa pada suatu dokumen. Namun, banyak mesin pencari mengharuskan pengguna untuk memasukan kata atau frasa yang tepat sebagai kueri pencarian. Hal ini dapat membatasi potensi pengguna untuk melakukan pencarian jika melakukan kesalahan penulisan atau jika tidak tau bagaimana mengeja sebuah kata.

Berbagai metode bisa digunakan untuk mengatasi hal ini, dengan memberi keleluasaan pada mesin pencari untuk menemukan kata yang berdekatan dengan kata yang dicari. Salah satu metode yang dapat digunakan adalah *approximation pattern matching*.

Makalah ini bertujuan untuk memaparkan sebuah pendekatan lain yang dapat digunakan pada mesin pencari, yaitu dengan melakukan *approximate pattern matching* atau aproksimasi pencocokan pola. Dengan menggunakan pendekatan *approximate pattern matching* diharapkan dapat memudahkan pengguna dalam melakukan pencarian teks pada sebuah dokumen dengan tidak harus memberikan kueri berupa kata atau frasa yang persis sama. Dengan itu maka peluang keberhasilan pengguna untuk dapat menemukan kata atau frasa yang dikehendaki bisa meningkat lebih baik.

Pendekatan yang digunakan penulis pada makalah ini akan diimplementasikan menggunakan algoritma komputasi *Levenshtein Distance*. Algoritma ini akan melakukan perhitungan "jarak" antara kata yang ingin dicari pengguna dengan tiap kata yang ada pada dokumen. Terminologi "jarak" ini dapat diartikan sebagai *edit distance* kedua buah kata atau banyaknya perubahan yang diperlukan untuk mengubah satu kata menjadi kata yang lain dengan melibatkan operasi penambahan, penghapusan, atau pertukaran karakter pada kata tersebut.

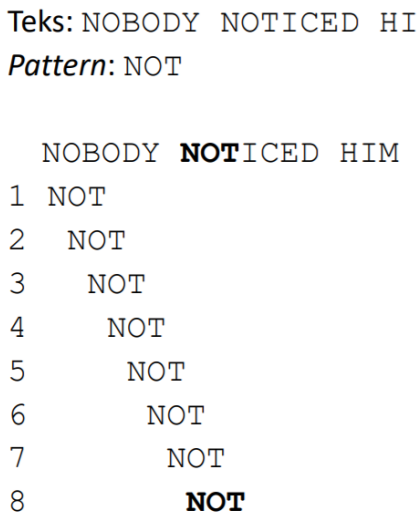
Pada implementasinya ini, algoritma akan melakukan pencarian serta perhitungan jarak dengan optimal pada *string*. Namun algoritma ini hanya dapat bekerja optimal jika jumlah karakter dari masukkan berjumlah sedikit, karena untuk *string* dengan ukuran besar atau berkarakter banyak membutuhkan waktu komputasi yang terus meningkat seiring dengan bertambahnya jumlah karakter.

Meski terdapat strategi untuk mengatasi waktu komputasi ini, diantaranya adalah membagi dokumen menjadi beberapa bagian baru kemudian hasil nya akan digabung, namun perlu studi lebih lanjut untuk dapat menilai seberapa efektif cara itu untuk dapat mengurangi waktu komputasi yang tinggi. Sehingga penulis saat ini hanya akan melakukan implementasi dengan jumlah masukan yang terbilang kecil.

II. Dasar Teori

A. *Approximate Pattern Matching*

Aproksimasi pencocokan pola merupakan sebuah persoalan dalam domain ilmu komputer khususnya algoritma pemrosesan string. Dalam persoalan ini diberikan dua buah kata, *t* dan *p*, dengan panjang masing-masing adalah *m* dan *n*, serta sebuah bilangan bulat *k* yang mengindikasikan jumlah maksimal *edit distance* (akan didiskusikan kemudian) antara kedua kata yang masih ditolerir [1]. Solusi klasik dari persoalan ini menggunakan strategi algoritma dynamic programming atau pemrograman dinamis dengan kompleksitas waktu O(mn).



Gambar 2.1 *Exact Pattern Matching* dengan *Brute Force*
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Berbeda dengan *exact pattern matching* pada umumnya, approximate pattern matching memberikan toleransi perbedaan antara pola yang diminta dengan teks yang diperiksa. Hal ini memberikan keleluasaan dalam penggunaannya untuk dapat diterapkan dalam berbagai bidang serta membuka ruang untuk melakukan modifikasi yang disesuaikan dengan kebutuhan.

Beberapa contoh aplikasi dan penerapan aproksimasi pencocokan pola diantaranya:

1. *Spell checking*

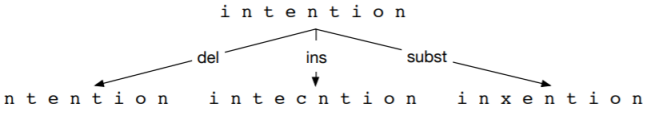
   Karena kemampuannya untuk dapat menghitung perbedaan serta kesalahan sebuah kata dibandingkan kata lain maka membuat *approximate pattern checking* ini dapat digunakan untuk memberikan masukan kepada pengguna jika sebuah kata yang diketikkan memiliki kesalahan pengejaan baik yang disebabkan oleh ketidaktahuan pengguna atau diakibatkan kesalahan pengetikan.

2. *Nucleotide matching*

   Dalam ilmu biologi, atau lebih spesifiknya adalah bioinformatika, DNA merupakan komponen yang sangat penting. Pemrosesan dan pengolahan data pada DNA membutuhkan pembacaan pada pola yang ada pada susunan nukleotida.

B. *Edit Distances*

Edit distance merupakan sebuah metode yang melakukan perhitungan banyaknya penambahan, pengurangan, substitusi, dan/atau transposisi antar karakter pada dua buah kata [2]. Metode ini diperkenalkan oleh Damerau pada 1964 sebagai bagian dari presentasinya mengenai metode mengoreksi kesalahan pengejaan.



Gambar 2.2 Operasi perubahan
Sumber: "Regular Expressions, Text Normalization, Edit Distance." Speech and Language Processing: an Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, by Daniel Jurafsky and James H. Martin, Pearson, 2020, pp. 22–23.

Secara formal, *edit distance* d(*a*, *b*) didefinisikan sebagai jumlah minimum operasi yang dilakukan untuk mengubah *a* menjadi *b*, dengan *a* dan *b* adalah sebuah *strings*. Algoritma perhitungan *edit distance* yang paling umum diperkenalkan oleh Levenshtein pada 1966 [3]. Namun berbeda dengan metode yang digunakan oleh Damerau, Levenshtein hanya memperhitungkan tiga buah operasi alih-alih empat seperti yang digunakan oleh Damerau. Ketiga operasi ini adalah penambahan, penghapusan, serta substitusi karakter.

d(kitten, sitting)
1. kitten → sitten
2. sitten → sittin
3. sittin → sitting

Contoh 2.1

Pada contoh diatas dapat dilihat bahwa untuk mengubah kata kitten menjadi sitting diperlukan tiga buah langkah yang melibatkan dua buah operasi. Hal ini menunjukan bahwa nilai dari Levenshtein edit distance untuk d(kitten, sitting) adalah 3.

Nilai edit distance ini akan dapat berbeda tergantung dengan algoritma perhitungan yang digunakan. Sebagai contoh jika algoritma *Longest Common Subsequence* (LCS) yang digunakan maka nilai dari d(kitten, sitting) akan menjadi 5. Hal ini terjadi karena pada algoritma itu hanya mengenal dua buah operasi yaitu penambahan dan penghapusan karakter saja.

C. *Levenshtein Edit Distances*

Jarak edit Levenshtein, seperti yang telah disebutkan pada bagian sebelumnya, merupakan salah satu metode untuk dapat menghitung *edit distance* dari dua buah *string*. *Levenshtein Edit Distances* menggunakan strategi pemrograman dinamis untuk melakukan perhitungan.

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}(\operatorname{tail}(a), \operatorname{tail}(b)) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} \operatorname{lev}(\operatorname{tail}(a), b) \\ \operatorname{lev}(a, \operatorname{tail}(b)) \\ \operatorname{lev}(\operatorname{tail}(a), \operatorname{tail}(b)) \end{cases} & \text{otherwise.} \end{cases}$$

Gambar 2.3 Definisi *Levenshtein Edit Distances*
Sumber: https://wikimedia.org/api/rest_v1/media/math/render/svg/10554aecc5e56da9be4657acd75b9a67b5e8b394

Levenshtein Edit Distances pada awalnya didefinisikan sebagai sebuah fungsi rekursif. Namun algoritma ini telah banyak dikembankan, dimodifikasi, serta diimprovisasi menjadi berbagai varian. Terdapat beberapa macam varian dari *Levenshtein Edit Distances* ini. Masing masing varian memiliki cara implementasi nya sendiri-sendiri.

1. Varian rekursif

   Varian ini merupakan varian asli seperti apa yang didefinisikan saat awal disusun oleh Levenshtein. Kasus basis pada fungsi rekursif ini adalah ketika salah satu dari masukkan merupakan *string* kosong. Dengan begitu maka fungsi akan mengembalikan panjang dari *string* lainnya sebagai keluaran.

   Cara rekursif seperti ini adalah cara yang tidak efektif dan membuang-buang sumber daya, karena di setiap rekursi, algoritma akan menghitung ulang kembali *substring* yang sudah dihitung sebelumnya.

2. Varian dengan matriks

   Dengan menerapkan cara iteratif, pada varian ini algoritma *Levenshtein Edit Distance* diselesaikan dengan membagi permasalahan menjadi sub-masalah yang lebih kecil, sebagaimana permasalahan dalam *dynamic programming* lainnya diselesaikan. Hal ini membantu mengurangi beban komputasi yang dilakukan secara berulang.

   Dengan varian ini persoalan dapat diselesaikan dengan jumlah komputasi lebih sedikit karena penyelesaian persoalan pada langkah ke *i* dapat diselesaikan dengan menggunakan hasil dari penyelesaian persoalan pada langkah ke *i*-1, penyelesaian persoalan pada langkah ke *i-1* dapat diselesaikan dengan menggunakan hasil dari penyelesaian persoalan pada langkah ke *i*-2, begitu seterusnya .

Src\Tar	#	e	x	e	c	u	t	i	o	n
#	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	6	7	8
n	2	3	4	5	6	7	8	7	8	7
t	3	4	5	6	7	8	7	8	9	8
e	4	3	4	5	6	7	8	9	10	9
n	5	4	5	6	7	8	9	10	11	10
t	6	5	6	7	8	9	8	9	10	11
i	7	6	7	8	9	10	9	8	9	10
o	8	7	8	9	10	11	10	9	8	9
n	9	8	9	10	11	12	11	10	9	8

Gambar 2.4 Contoh matriks penyelesaian *Levenshtein Edit Distance*
Sumber: "Regular Expressions, Text Normalization, Edit Distance." Speech and Language Processing: an Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, by Daniel Jurafsky and James H. Martin, Pearson, 2020, pp. 22–23.

Angka-angka pada baris pertama merupakan *cost* atau besarnya perubahan yang harus dilakukan jika ingin mengubah *substring*(0, *t*) dari kueri menjadi *string* kosong. Pada baris kedua angka yang ada merupakan *cost* untuk mengubah *substring*(0, *t*) kueri menjadi *substring*(0, 0) dari *string* acuan. Dan begitu seterusnya sehingga angka-angka pada baris ke-*s* merupakan *cost* yang dibutuhkan untuk mengubah *substring*(0, *t*) kueri menjadi *substring*(0, *s*+1) dari *string* acuan.

Untuk mengetahui *cost* pada sel (*t*, *s*), algoritma ini hanya perlu untuk melihat nilai-nilai dari subpersoalan sebelumnya yang terletak pada sel (*t*-1, *s*-1), (*t*-1, *s*), dan (*t*, *s*-1). Hal ini memudahkan perhitungan karena tidak perlu menghitung kembali *cost* dari awal.

Dalam perhitungannya pun akan terbentuk beberapa pola perhitungan:

a. Saat a[t] sama dengan b[s] maka nilai *cost* nya pada sel (*t, s*) didapat dengan mengambil nilai dari sel (*t*-1*, s*-1)

b. Saat a[t] tidak sama dengan b[s] dan operasi yang dilakukan adalah penambahan karakter maka nilai *cost* nya pada sel (*t, s*) didapat dengan mengambil nilai dari sel (*t, s*-1) ditambah 1

c. Saat a[t] tidak sama dengan b[s] dan operasi yang dilakukan adalah penghapusan karakter maka nilai *cost* nya pada sel (*t, s*) didapat dengan mengambil nilai dari sel (*t*-1*, s*) ditambah 1

d. Saat a[t] tidak sama dengan b[s] dan operasi yang dilakukan adalah substitusi karakter maka nilai *cost* nya pada sel (*t, s*) didapat dengan mengambil nilai dari sel (*t*-1*, s*-1) ditambah 1

Pada praktiknya yang dapat kondisi yang dapat diketahui oleh algoritma hanya kondisi a), maka implementasinya adalah jika a[t] tidak sama dengan b[s] maka nilai *cost* nya pada sel (*t, s*) didapat dari *min(cost(t-1, s-1), cost(t-1, s), cost(t, s-1))* ditambah 1.

3. Varian dengan array

   Pada varian dengan matriks dapat dilihat bahwa hanya diperlukan satu buah baris sebelum sel aktif dan satu buah kolom sebelum sel aktif yang diperlukan nilainya. Maka dari itu hadirlah varian ini dengan hanya dibentuk oleh dua buah array yang akan menampung *cost* dari baris dan kolom sebelum berurutan.

III. Implementasi Levenshtein Distance

Penulis akan menggunakan varian ketiga dari *Levenshtein Edit Distance* dalam implementasi ini. Pertama-tama perlu diketahui bagaimana algoritma ini bekerja. Algoritma dari varian ini dijabarkan sebagai berikut:

1. Tetapkan *m* dan n sebagai panjang dari string a dan b secara berurutan. Jika m = 0, kembalikan n dan keluar. Jika n = 0, kembalikan m dan keluar.
2. Inisialisasi array t dengan nilai [0..n]
3. Periksa setiap karakter a (i)
4. Periksa setiap karakter b (j)
5. Inisialisasi s[0] dengan i + 1
6. Jika a[i] sama dengan b[j], cost = t[j]
7. Jika a[i] tidak sama dengan b[j], cost = min(t[j], t[j+1], s[j])
8. Salin nilai s ke t dan kembali ke tahap 3
9. kembalikan nilai s[n]

Dengan algoritma yang telah didefinisikan di atas, maka algoritma ini dapat segera diimplementasikan menjadi sebuah program untuk mendapatkan daftar kata yang memenuhi persyaratan dalam pencarian.

Program ini harus menerima tiga buah masukkan, yaitu kata yang ingin dicari (input), dokumen berisi banyak kata (document), dan yang terakhir adalah ambang edit distance yang dikehendaki (maxDistance). Program ini akan mengembalikan daftar kata kata apa saja yang memenuhi kriteria.

Gambar 3.1 menampilkan *pseudocode* dari program yang dijelaskan ini. Program ini akan melakukan iterasi pada tiap kata yang ada di dokumen dan menghitung nilai edit distance tiap-tiap kata dengan input yang diberikan. Lalu dilakukan filtrasi pada kata kata yang melewati ambang batas. Hanya kata-kata yang nilai *edit distance* nya di bawah ambang batas akan dimasukkan ke dalam list dan dikembalikan oleh program.

```
function getMatchWords(input, document, maxDistance):
    matchWords: Array of word
    wordScore : Integer

    for word in document:
        wordScore = calculateDistance(input, word)
        if wordScore < maxDistance:
            matchWord.push(word)

     return matchWords
```

Gambar 3.1 Pseudocode program

Dengan sudah didapatkannya daftar kata yang memiliki kecocokan tersebut maka dapat dengan mudah selanjutnya diproses pada web untuk menampilkan kata kata tersebut di layar. Hal ini dapat dilakukan dengan mengiterasi kata pada list hasil keluaran program sebelumnya dan melakukan mengganti perannya di berkas file HTML menjadi sebuah tag span dengan kelas khusus, sehingga dapat diberikan *highlight* atau penanda kata mana saja yang memiliki kecocokan dengan masukkan pengguna.

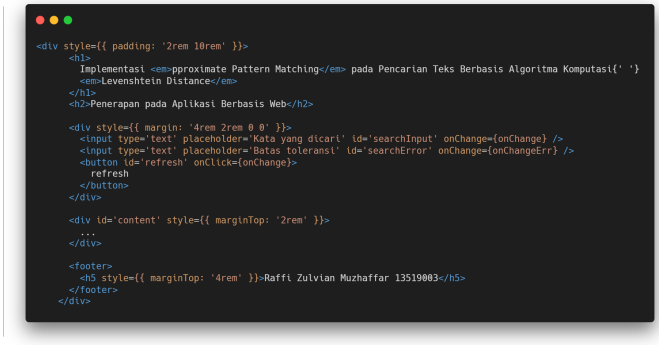
Berikut merupakan implementasi program ini pada bahasa javascript oleh penulis:

```
class ApproximatePatternMatcher {
  constructor() {}

  /**
   * Implementasi Levenshtein Edit Distance menggunakan model 2 array
   * @param {String} a
   * @param {String} b
   */
  calculateDistances(a, b) {
    let m = a.length;
    let n = b.length;
    let t = [];
    let s;
    let j;

    // Kasus basis pertama
    if (m === 0) {
      return n;
    }

    // Kasus basis kedua
    if (n === 0) {
      return m;
    }

    // Inisialisasi cost awal
    for (let i = 0; i <= n; i++) {
      t[i] = i;
    }

    // Perhitungan
    for (let i = 0; i < m; i++) {
      for (s = [i + 1], j = 0; j < n; j++) {
        if (a[i] === b[j]) {
          s[j + 1] = t[j];
        } else {
          s[j + 1] = Math.min(t[j], t[j + 1], s[j]) + 1;
        }
      }
      t = s;
    }

    return s[n];
  }
}
```

Gambar 3.2 Implementasi pada javascript

```
const onChange = (e) => {
    e.preventDefault();

    const content = document.getElementById('content');
    const search = document.getElementById('searchInput');
    const text = content.innerText;

    if (search.value.length > 0) {
      let words = content.innerText.split(' ');

      const levdis = new ApproximatePatternMatcher();
      const newWords = words.filter((word, i, a) => {
        return levdis.calculateDistances(search.value, word) < toleransi && a.indexOf(word) === i;
      });

      let newText = text;
      newWords.forEach((word) => {
        let re = new RegExp(word, 'g');
        newText = newText.replace(re, `<span class='highlight'>${word}</span>`);
      });

      content.innerHTML = newText;
    } else {
      content.innerHTML = text;
    }
  };
```

Gambar 3.3 Implementasi pada javascript



```
<div style={{ padding: '2rem 10rem' }}>
      <h1>
        Implementasi <em>pproximate Pattern Matching</em> pada Pencarian Teks Berbasis Algoritma Komputasi{' '}
        <em>Levenshtein Distance</em>
      </h1>
      <h2>Penerapan pada Aplikasi Berbasis Web</h2>

      <div style={{ margin: '4rem 2rem 0 0' }}>
        <input type='text' placeholder='Kata yang dicari' id='searchInput' onChange={onChange} />
        <input type='text' placeholder='Batas toleransi' id='searchError' onChange={onChangeErr} />
        <button id='refresh' onClick={onChange}>
          refresh
        </button>
      </div>

      <div id='content' style={{ marginTop: '2rem' }}>
        ...
      </div>

      <footer>
        <h5 style={{ marginTop: '4rem' }}>Raffi Zulvian Muzhaffar 13519003</h5>
      </footer>
    </div>
```

Gambar 3.4 Potongan HTML

Program yang penulis buat berhasil dijalankan dan memberikan hasil sesuai yang diharapkan, yaitu memberi highlight pada kata yang masih dalam ambang batas *edit distance*. Pada halaman web yang penulis buat, pengguna dapat memasukkan kata yang ingin dicari serta ambang batas edit distance yang diinginkan.

Implementasi *pproximate Pattern Matching* pada Pencarian Teks Berbasis Algoritma Komputasi *Levenshtein Distance*

Penerapan pada Aplikasi Berbasis Web

Kata yang dicari | Batas toleransi | refresh

Tugas Makalah IF2.1. Strategi Algoritma Sem. II Tahun 2.2./2.2.span class='highlight'>1.1. Latar Belakang Mahasiswa Teknik Informatika sebaiknya tidak hanya mampu membangun program aplikasi, tetapi juga diharapkan mampu menulis karya ilmiah. Dengan membuat tulisan, maka berbagai pemikiran, karya, maupun, penelitian di bidang Informatika dapat dikomunikasikan ke tengah masyarakat. Salah satu mata kuliah dasar di Teknik Informatika adalah IF2.1. Strategi Algoritma. Mahasiswa yang mengambil mata kuliah IF2.1. diwajibkan menulis sebuah makalah ilmiah dan video makalah dengan memilih salah satu topik strategi algoritma (lihat penjelasan tugas di bawah ini). Dengan menulis makalah maka kita telah ikut mendokumentasikan pengetahuan dan berbagi ilmu dengan orang lain. Bonus: Video makalah. Video memberikan gambarkan tentang persoalan yang diangkat dan solusinya. Video diunggah ke YouTube. Tujuan penulisan makalah: 1. Memotivasi mahasiswa agar memiliki kemampuan menulis untuk menuangkan ide-ide atau hasil risetnya; 2. Melakukan eksplorasi terhadap isu, metode, dan masalah yang dipelajari dalam pengembangan serta menyebarkan aplikasi yang mendukung teknologi informasi; 3. Sebagai media untuk berbagi informasi hasil-hasil pemikiran dan penelitian. Sem.a makalah mahasiswa akan dimuat di dalam website dosen, sehingga siapa pun dapat melihat dan membaca karya ilmiah mahasiswa Teknik Informatika Tahun kedua; Tujuan membuat video (bonus): 1. Memotivasi mahasiswa agar dapat menyajikan hasil riset mandirinya dalam bentuk multimedia (audic visual) yang menarik dan mudah dimengerti oleh pembelajar lain.ya. 2. Berbagi pengetahuan lewat video publik.

Raffi Zulvian Muzhaffar 13519003

Gambar 3.5 Tampilan awal halaman web

Implementasi *pproximate Pattern Matching* pada Pencarian Teks Berbasis Algoritma Komputasi *Levenshtein Distance*

Penerapan pada Aplikasi Berbasis Web

maka | Batas toleransi | refresh

Tugas Makalah IF2211 Strategi Algoritma Sem. II Tahun 2020/2021 1. Latar Belakang Mahasiswa Teknik Informatika sebaiknya tidak hanya mampu membangun program aplikasi, tetapi juga diharapkan mampu menulis karya ilmiah. Dengan membuat tulisan, maka berbagai pemikiran, karya, maupun, penelitian di bidang Informatika dapat dikomunikasikan ke tengah masyarakat. Salah satu mata kuliah dasar di Teknik Informatika adalah IF2211 Strategi Algoritma. Mahasiswa yang mengambil mata kuliah IF2211 diwajibkan menulis sebuah makalah ilmiah dan video makalah dengan memilih salah satu topik strategi algoritma (lihat penjelasan tugas di bawah ini). Dengan menulis makalah maka kita telah ikut mendokumentasikan pengetahuan dan berbagi ilmu dengan orang lain. Bonus: Video makalah. Video memberikan gambarkan tentang persoalan yang diangkat dan solusinya. Video diunggah ke YouTube. Tujuan penulisan makalah: 1. Memotivasi mahasiswa agar memiliki kemampuan menulis untuk menuangkan ide-ide atau hasil risetnya; 2. Melakukan eksplorasi terhadap isu, metode, dan masalah yang dipelajari dalam pengembangan serta menyebarkan aplikasi yang mendukung teknologi informasi; 3. Sebagai media untuk berbagi informasi hasil-hasil pemikiran dan penelitian. Semua makalah mahasiswa akan dimuat di dalam website dosen, sehingga siapa pun dapat melihat dan membaca karya ilmiah mahasiswa Teknik Informatika Tahun kedua; Tujuan membuat video (bonus): 1. Memotivasi mahasiswa agar dapat menyajikan hasil riset mandirinya dalam bentuk multimedia (audio visual) yang menarik dan mudah dimengerti oleh pembelajar lainnya. 2. Berbagi pengetahuan lewat video publik.

Raffi Zulvian Muzhaffar 13519003

Gambar 3.6 Tampilan web ketika memasukkan pencarian

Perlu diperhatikan bahwa ketika pengguna tidak memasukkan nilai sebagai ambang batas pada kolom yang disediakan, maka secara default ambang batas yang digunakan adalah 0 atau dengan kata lain hanya kata dengan kecocokan 100% saja yang akan ditampilkan.

Implementasi *pproximate Pattern Matching* pada Pencarian Teks Berbasis Algoritma Komputasi *Levenshtein Distance*

Penerapan pada Aplikasi Berbasis Web

maka | 4 | refresh

Tugas Makalah IF2211 Strategi Algoritma Sem. II Tahun 2020/2021 1. Latar Belakang Mahasiswa Teknik Informatika sebaiknya tidak hanya mampu membangun program aplikasi, tetapi juga diharapkan mampu menulis karya ilmiah. Dengan membuat tulisan, maka berbagai pemikiran, karya, maupun, penelitian di bidang Informatika dapat dikomunikasikan ke tengah masyarakat. Salah satu mata kuliah dasar di Teknik Informatika adalah IF2211 Strategi Algoritma. Mahasiswa yang mengambil mata kuliah IF2211 diwajibkan menulis sebuah makalah ilmiah dan video makalah dengan memilih salah satu topik strategi algoritma (lihat penjelasan tugas di bawah ini). Dengan menulis makalah maka kita telah ikut mendokumentasikan pengetahuan dan berbagi ilmu dengan orang lain. Bonus: Video makalah. Video memberikan gambarkan tentang persoalan yang diangkat dan solusinya. Video diunggah ke YouTube. Tujuan penulisan makalah: 1. Memotivasi mahasiswa agar memiliki kemampuan menulis untuk menuangkan ide-ide atau hasil risetnya; 2. Melakukan eksplorasi terhadap isu, metode, dan masalah yang dipelajari dalam pengembangan serta menyebarkan aplikasi yang mendukung teknologi informasi; 3. Sebagai media untuk berbagi informasi hasil-hasil pemikiran dan penelitian. Semua makalah mahasiswa akan dimuat di dalam website dosen, sehingga siapa pun dapat melihat dan membaca karya ilmiah mahasiswa Teknik Informatika Tahun kedua; Tujuan membuat video (bonus): 1. Memotivasi mahasiswa agar dapat menyajikan hasil riset mandirinya dalam bentuk multimedia (audio visual) yang menarik dan mudah dimengerti oleh pembelajar lainnya. 2. Berbagi pengetahuan lewat video publik.

Raffi Zulvian Muzhaffar 13519003

Gambar 3.7 Tampilan pencarian dengan ambang batas

Dapat dilihat pada gambar 3.7 bahwa ketika memasukkan ambang batas, kata-kata yang di-*highlight* atau ditampilkan pada halaman web bertambah banyak mengikuti ketentuan yang ada. Hal ini menunjukan keberhasilan algoritma *Levenshtein Edit Distance* yang dapat digunakan untuk melakukan pencarian dengan pola *approximate pattern matching*.

IV. KESIMPULAN

Algoritma *Levenshtein Edit Distance* dapat digunakan dengan baik untuk dalam aplikasinya untuk melakukan pencarian dengan *approximate pattern matching* yang memberikan hasil pencarian terdekat dengan kata yang dicari sebagai kueri.

Meskipun algoritma ini berhasil diimplementasikan sebagaimana harapan dari penulis, namun rupanya performa dari algoritma ini belum cukup baik untuk aplikasi pada skala besar. Kompleksitas waktu yang masih tinggi membuat algoritma ini berjalan sangat lambat pada dokumen atau teks dengan jumlah kata yang besar. Selain itu penggunaan memori pun meningkat dengan bertambahnya jumlah kata yang harus diproses dari dalam dokumen.

Dengan kendala yang ada ini diperlukan studi lebih lanjut untuk dapat menemukan dan menciptakan optimasi dari algoritma ini agar dapat berjalan dengan kompleksitas yang lebih rendah, sehingga dapat digunakan untuk dokumen berskala besar.

Penulis berharap bahwa makalah ini dapat menjadi bahan pembelajaran pembaca mengenai topik yang sedang dibahas dan dapat digunakan sebagai acuan baik untuk studi akademik, pembelajaran mandiri, maupun sebagai bahan pertimangan dalam melakukan sebuah proyek pribadi pembaca. Pembaca pun diharapkan tidak hanya berhenti pada makalah ini, dan melakukan riset lebih lanjut ke berbagai sumber informasi yang tersedia seperti buku, jurnal akademik, website, maupun kepada para pakar yang ahli di bidang ini.

Video Link at Youtube

...

Daftar Pustaka

[1] Baeza-Yates, R., & Navarro, G. (1996). A faster algorithm for approximate string matching [Abstract]. Combinatorial Pattern Matching, 1-23. doi:10.1007/3-540-61258-0_1

[2] Boytsov, L. (2011). Indexing methods for approximate dictionary searching. ACM Journal of Experimental Algorithmics, 16, 16–17. https://doi.org/10.1145/1963190.1963191

[3] "Regular Expressions, Text Normalization, Edit Distance." *Speech and Language Processing: an Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, by Daniel Jurafsky and James H. Martin, Pearson, 2020, pp. 22–23.

[4] Gilleland, M. (n.d.). Levenshtein Distance. University of Pittsburg. Retrieved May 11, 2021, from http://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Mei 2021

Raffi Zulvian Muzhaffar 13519003