Implementasi Pencocokan String untuk Plagiarism Checker pada Makalah Mahasiswa di Website Rinaldi Munir dengan W-Shingling, KMP, dan Levensthein Distance

Dhafin Fawaz Ikramullah - 13522084
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
Email (gmail): 13522084@std.stei.itb.ac.id

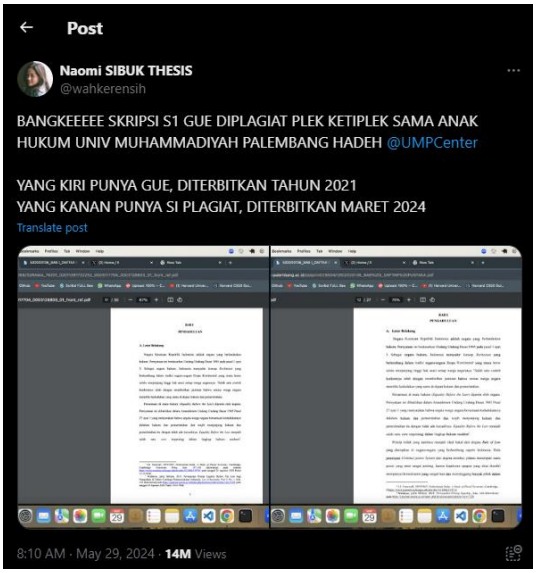
***Abstract*—Salah satu isu serius di kalangan mahasiswa adalah plagiarisme. Untuk menangani masalah ini, kita dapat mengecek apakah suatu karya terindikasi plagiarisme secara manual. Namun jika dilakukan secara manual akan menghabiskan waktu yang sangat banyak. Sebab itulah diperlukan sebuah alat yang efisien untuk mendeteksi kesamaan teks pada karya dari mahasiswa. Tentunya kita bisa menggunakan layanan seperti *turnitin*, tetapi data pada layanan tersebut belum tentu merupakan data yang terbaru dan juga tidak gratis. Makalah ini membahas pengembangan dan implementasi dari plagiarism checker dengan algoritma pencocokan string. Algoritma yang digunakan dalam makalah ini diantaranya Knuth-Morris-Pratt, Shingles atau n-grams, dan Levenshtein Distance, yang masing-masing memiliki akan berguna dalam konteks tertentu. Dari hasil pengujian dapat ditunjukkan bahwa algoritma string matching yang digunakan dapat menghasilkan sebuah sistem efisien untuk melakukan pengecekan plagiarisme secara signifikan dibandingkan dengan melakukan pengecekan secara manual. Sistem yang telah dibuat juga dapat digunakan sebagai alat pendukung dalam menjaga integritas secara akademik.**

Kata kunci—string matching; plagiarisme; algoritma; shingles; w-shingling; n-grams; knuth morris pratt; levenshtein distance;

I. PENDAHULUAN

Selama menjalankan perkuliahan, tentu terdapat keluh kesah tentang masalah yang dialami setiap mahasiswa. Salah satu contohnya adalah rasa malas dari mahasiswa tersebut. Namun jangan sampai rasa malas tadi membuat kita memilih untuk melakukan plagiarisme. Plagiarisme merupakan sebuah tindakan penjiplakan karya orang lain dan mengklaim bahwa karya maupun gagasan tersebut adalah miliknya dan hasil pemikirannya sendiri.

Karena teknologi yang semakin berkembang, tindakan plagiarisme juga semakin mudah untuk dilakukan. Tetapi ini juga berdampak pada semakin canggihnya alat pemberantas plagiarisme. Walaupun plagiarisme smakin mudah, alat untuk mendeteksi plagiarisme juga semakin berkembang. Dengan adanya berkembangnya alat pendeteksi plagiarisme, diharapkan plagiarisme dapat berkurang.



Gambar 1. Contoh berita plagiarisme skripsi yang sedang viral (Sumber: https://x.com/wahkerensih/status/1795623830893076747)

II. LANDASAN TEORI

A. Plagiarisme

Berdasarkan Peraturan Menteri Pendidikan Nasional Nomor 17 Tahun 2010 tentang Pencegahan dan Penanggulangan Plagiat di Perguruan Tinggi [1], plagiarisme merupakan tindakan dengan sengaja maupun tidak sengaja dalam mendapatkan atau mencoba mendapatkan kredit atau nilai pada suatu karya ilmiah, dengan cara mengutip beberapa bagian atau keseluruhan karya ilmiah orang lain serta mengakui sebagai pemilik karya ilmiah tersebut tanpa menyatakan sumber secara tepat dan memadai.

B. String Matching

Algoritma pencocokan string, atau dapat disebut juga String Matching, merupakan algoritma yang dapat digunakan untuk

menemukan kemunculan suatu pola antara text atau string [2]. Algoritma penocokan string dapat dibagi menjadi 2 yaitu pencocokan string akurat dan pencocokan string aproksimasi [3].

Contoh algoritma pencocokan string akurat diantaranya

- Naive
- Knuth Morris Pratt
- Boyer Moore
- Automation Matcher (Determenistic Finite Automata)
- Aho Corasic
- Rabin Karp

Contoh algoritma pencocokan string apoksimasi diantaranya

- Hamming Distance
- Levenshtein Distance

Pada konteks string, prefix merupakan substring atau himpunan bagian dari string yang diambil dari elemen pertama dari elemen ke N. Suffix dari string merupakan substring atau himpunan bagian dari string yang siambil dari elemen ke N sampai elemen terakhir. Proper prefix merupakan prefix yang string itu sendiri tidak termasuk. Maksudnya elemen yang diambil hanya elemen pertama sampai elemen ke N dimana N kurang dari panjang string tersebut.

C. *W-Shingling*

W-shingling merupakan sebuah metode yang popular digunakan dalam pemrosesan Natural Language Processing. Metode ini adalah sebuah teknik pembentukan kumpulan substring (disebut shingle) dari sebuah teks dengan panjang tertentu [4]. Setiap shingle adalah urutan karakter atau kata yang berurutan dalam teks. Misalkan kita menganggap kata sebagai sebuah unit, maka ilustrasinya sebagai berikut.

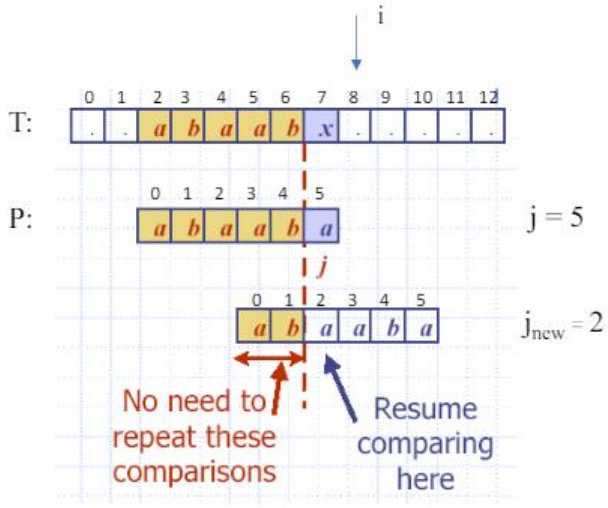
Text = “Saya suka makan ayam goreng 45.”

W = 3 (bisa disebut 3-shingling)

Shingles = [“Saya suka makan”, “suka makan ayam”, “makan ayam goreng”, “ayam goreng 45”]

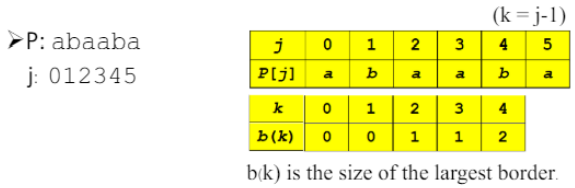
D. *Knuth Morris Pratt (KMP)*

Algoritma KMP merupakan algoritma pencarian pola pada sebuah text dengan urutan dari kiri ke kanan, tetapi pola yang boros untuk dicari lebih sedikit. Jika terjadi *mismatch* antara text dan pola P di P[j] misalnya T[i] ≠ P[j], terdapat penggeseran pola yang bisa kita lakukan agar pengecekan tidak boros yaitu prefix terbesar dari P[0..j-1] yang juga merupakan suffix dari P[1..j-1] [5].



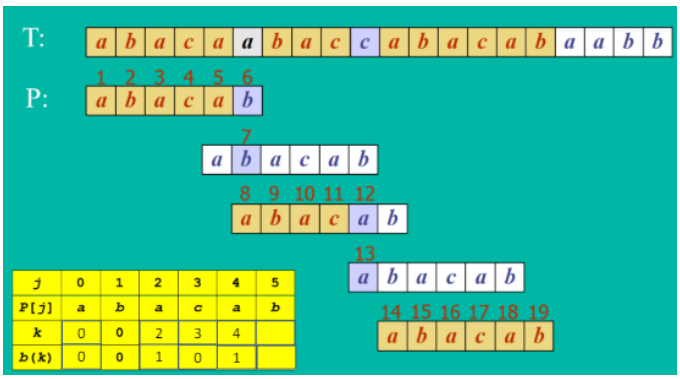
Gambar 2. Contoh pengecekan boros (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf)

Algoritma ini diimplementasikan dengan *preprocessing* dari pola untuk fungsi pinggiran yaitu ukuran dari prefix terbesar dari P[0..k] yang juga merupakan suffix dari P[1..k]. fungsi pinggiran ini akan digunakan saat terjadi mismatch pada P[j] lalu k kita geser menjadi j-1, dan cari j yang baru yaitu b(k). Contohnya sebagai berikut.



Gambar 3. Contoh fungsi pinggiran (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf)

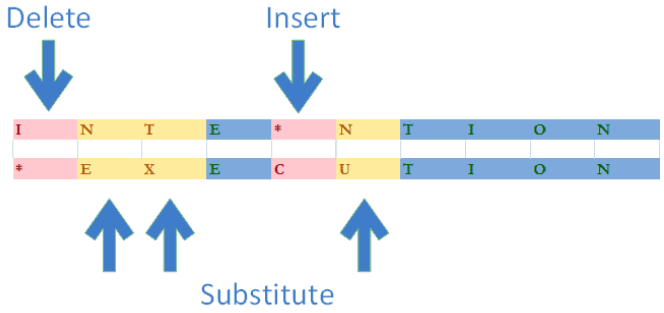
Untuk contoh dari ilustrasi dapat dilihat sebagai berikut



Gambar 4. Ilustrasi KMP (Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf)

E. *Levenshtein Distance*

Levenshtein Distance merupakan jumlah operasi edit yang bisa dicapai untuk mengubah sebuah string menjadi string lain [6]. Maksud dari operasi edit ini adalah operasi delete, insert, atau substitute sebuah karakter. Ilustrasinya sebagai berikut.



Gambar 5. Ilustrasi Levenshtein Distance (Sumber: https://www.baeldung.com/cs/levenshtein-distance-computation#:~:text=Levenshtein%20distance%20is%20the%20smallest,insertions%2C%20deletions%2C%20and%20substitutions.)

Secara matematis, levenshtein distance dapat dihitung sebagai berikut.

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Gambar 6. Levenshtein Distance secara matematis (Sumber: https://www.baeldung.com/cs/levenshtein-distance-computation#:~:text=Levenshtein%20distance%20is%20the%20smallest,insertions%2C%20deletions%2C%20and%20substitutions.)

F. *Web Scrapping*

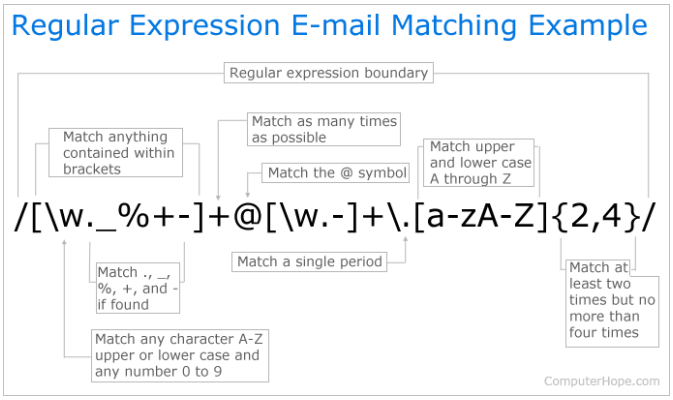
Web scrapping merupakan sebuah proses untuk mengekstraksi konten dan data dari sebuah website [7]. Saat kita membuka sebuah website, kita bisa mendownload gambar ataupun *copy paste* teks. Dalam web scrapping, kita melakukan hal yang mirip, namun dilakukan dengan otomatis dengan bot untuk mengekstrak data-data tersebut sesuka kita.

G. *Multithreading*

Thread merupakan sebuah unit eksekusi yang dibuat dalam sebuah proses pada sistem operasi. Multitheading merupakan sebuah teknik untuk memodelkan program dengan cara membuat beberapa thread agar jika ada sebuah instruksi yang lama, tidak membloklir program untuk melakukan hal lain [8]. Teknik ini berguna dalam web scrapping jika kita ingin sambil melakukan hal lain saat terjadi web request karena saat melakukan web request, thread yang melakukannya akan menunggu tanpa melakukan apa pun. Pada makalah ini sebenarnya yang dimanfaatkan adalah Concurency karena Typescript merupakan bahasa dengan hanya singlethreaded. Tapi konsep ini mirip yaitu kita bisa tetap melakukan tugas lain misalnya selama melakukan download file pdf.

H. *Regular Expression*

Regular expression merupakan sebuah string yang dapat kita buat untuk membuat pola yang bisa digunakan untuk menemukan sebuah string pada string lain [9]. Salah satu contoh pemanfaatan regex adalah untuk pengecekan email seperti sebagai berikut.



Gambar 7. Contoh regex untuk pengecekan email (Sumber: https://www.computerhope.com/jargon/r/regex.htm)

III. Implementasi dan Pengujian

Sistem yang dibuat akan diimplementasikan dengan bahasa pemrograman Typescript. Framework yang digunakan untuk *frontend website* dan *backend server* adalah Next.js. Data-data makalah yang akan digunakan untuk pengecekan plagiarism di *preprocess* terlebih dahulu berupa file dalam format *json*.

A. *Web Scrapping & Multithreading*

Sebelum memulai implementasi, data-data makalah harus kita *download* dan *extract* menjadi teks terlebih dahulu. Karena url pada website Rinaldi Munir tidak konsisten, maka setiap url yang di dalam halaman websitenya terdapat kumpulan url file pdf akan di simpan terlebih dahulu. Totalnya terdapat 90 url yang halaman websitenya memiliki kumpulan url ke file pdf tiap makalah.

```
[
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/Makalah2023.htm",
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Makalah2022.htm",
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Makalah2021.htm",
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2019-2020/Makalah2020.htm",
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2018-2019/Makalah2019.htm",
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Makalah2018.htm",
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2016-2017/Makalah2017.htm",
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2015-2016/Makalah2016.htm",
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2014-2015/Makalah2015.htm",
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2013-2014-genap/Makalah2014.htm",
    "https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2013-2014/Makalah2013.htm",
```

Gambar 8. Kumpulan url yang akan di scrap. (Sumber: https://github.com/DhafinFawwaz/Plagiarism-Checker-Makalah/blob/main/app/dataset/json/scrape-links.json)

Dari setiap url tersebut, akan diakses halamannya dan diambil semua url yang berakhir dengan akhiran *.pdf* dan digabungkan ke sebuah larik besar. Totalnya terdapat 6413 makalah yang akan dicek.

Agar scrapping lebih cepat, diimplementasikan multithreading. Namun kita tidak boleh melakukan web request dengan jumlah sangat banyak dalam satu waktu karena ini

merupakan serangan ke website secara tidak sengaja. Ini merupakan Tindakan illegal dan akan diblok oleh website tersebut. Maka jumlah thread yang akan digunakan hanya 16 thread sehingga dalam satu waktu hanya ada maksimal 16 buah file pdf yang akan didownload. Dengan menghitung rata-rata pdf yang selesai didownload, kita juga dapat menghitung perkiraan lama waktu proses scrapping selesai. Rumusnya sebagai berikut

$$t = (t1 - t0) / i \times (N - i)$$

t = Approksimasi Sisa Waktu

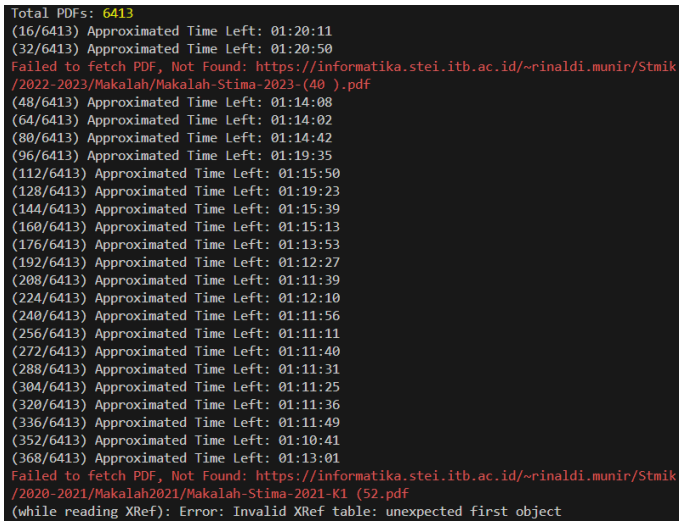
t1 = Waktu sekarang

t0 = Waktu awal

i = Jumlah file pdf yang telah didownload

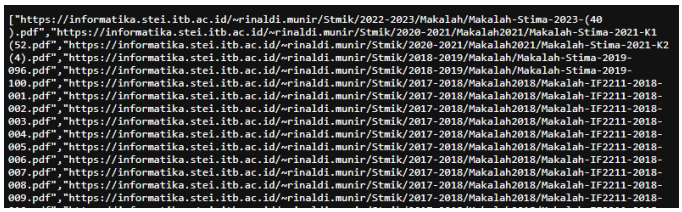
N = Total file pdf yang akan didownload

Dari metode ini kita menemukan bahwa total waktu yang dibutuhkan untuk mendownload 6413 file pdf dan maksimal 16 thread adalah sekitar 1 jam 15 menit. Jika ada sebuah file pdf yang berbeda dibanding yang lain, maka rata-rata akan kurang akurat. Semakin banyak pdf yang sudah didownload maka akan semakin akurat aproksimasi waktunya.



Gambar 9. Total File PDF, approximasi waktu, dan error selama scrapping. (Sumber: dari penulis)

Selama proses web scrapping, terdapat 485 file yang tidak dapat disimpan. Salah satu contohnya terdapat pada gambar 9. Ada url yang salah pengetikan, ada yang format pdf kurang wajar sehingga tidak bisa dibaca, ada juga yang bisa dibaca tapi mengakibatkan program halt. Ini membuat scrapping harus memiliki sistem *time out* jika parsing file pdf terlalu lama dan semua proses harus diulang dari awal.



Gambar 10. Kumpulan url yang mengakibatkan error (Sumber: https://github.com/DhafinFawwaz/Plagiarism-Checker-Makalah/blob/main/app/dataset/json/not-found.json)

B. *Data Cleaning & Preprocessing*

Salah satu tahapan pada algoritma string matching yang cukup umum adalah tahap *tokenizing*. Dalam tahap ini, dilakukan pembersihan data dengan memanfaatkan Regular Expression berikut:

- Penghapusan new line (\n) dan carriage return (\r). Regex: [\r\n]
- Penghapusan page break (-- Page (<angka>) Break --). Regex: ----------Page \(\w+\) Break----------
- Semua spasi lebih dari diubah satu menjadi hanya satu. Regex: \s+

Format data hasil scraping berupa file json dengan menyimpan judul, url, dan hasil ekstraksi dan pembersihan dari makalah seperti sebagai berkut.

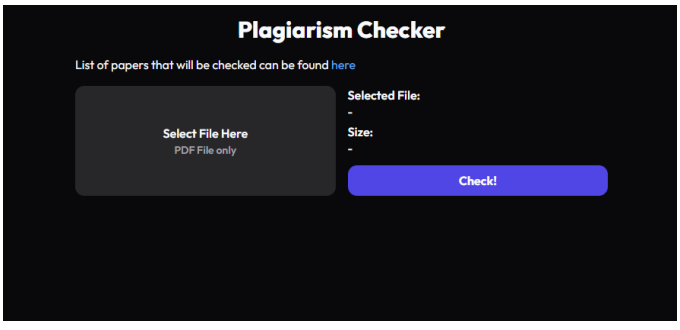
```
[{“title”:””, “href”:””,” content”:””}]
```

Dalam implementasi sebenarnya, data hasil scraping juga harus dibagi menjadi 2 file terpisah. Ini dilakukan karena restriksi dari commit github yang membatasi maksimum file sebesar 100 MB sementara file hasil scrapping sebesar 134 MB.

Perhatikan pula bahwa preprocessing *w-shingles* tidak dilakukan di sini. Setelah dicoba, preprocessing ini akan membuat ukuran file mencapai 884 MB sementara optimasi yang dilakukan hanya mempercepat kalkulasi sekitar 8 detik.

C. *Web Development*

Seperti yang dijelaskan sebelumnya, website akan dikembangkan dengan memanfaatkan framework Next.js. Pengguna akan bisa memasukkan file pdf dan melakukan pengecekan plagiarisme pada file yang diinginkan tersebut. Website kemudian akan melakukan *API Call* ke server dan akan dikembalikan persentase plagiarisme keseluruhan, persentase plagiarisme tiap makalah, dan kumpulan bagian makalah yang berkemungkinan plagiarisme untuk setiap makalah yang dicek. Pengguna akan bisa memilih bagian dari tiap makalah yang berkemungkinan plagiarisme. Berikut ini tampilan menu awal website.



Gambar 11. Tampilan website (Sumber: dari penulis)

D. Pencarian Plagiarisme

Dalam implementasi algoritmanya, input akan berupa string yang merupakan text hasil ekstraksi file pdf dari input pengguna. Output berupa persentase plagiarisme keseluruhan, bertipe double, lalu larik yang isinya persentase plagiarisme tiap makalah, dan larik jangkauan plagiarism. Larik jangkauan ini berupa larik integer yang untuk setiap elemennya merupakan larik dengan panjang 2 yaitu elemen pertama merupakan index mulai plagiarisme, elemen ke kedua merupakan index berhenti plagiarisme. Larik ini akan menyatakan jangkauan/range/interval dari text yang berkemungkinan plagiarisme. Agar lebih jelas, lihat table berikut

Input	Text: string

Output	{
	Persentase keseluruhan: double,
	{
	Persentase tiap Makalah: double
	Larik jangkauan plagiarisme: integer[][2]
	}
	}

Dari frontend website, pengguna akan memasukkan file pdf. Lalu akan di terima oleh backend server. Di sini file pdf akan ekstraksi dulu menjadi sebuah text. Lalu ambil juga semua data hasil scrapping sebelumnya untuk dilakukan pengecekan.

```
export async function POST(req: Request) {
    const file = await req.formData();
    const pdfFile = file.get('file') as File;
    const pdfBuffer = await pdfFile.arrayBuffer();

    const content = await extractPdfTextContentFromBuffer(Buffer.from(pdfBuffer));
    if(!content || content === "") return NextResponse.json({ error: "Not a valid pdf file!" });
    const dataClean = loadDataClean();

    const result = findPlagiarism({content: content, title: pdfFile.name, href: ""}, dataClean);
    return NextResponse.json(result);
}
```

Gambar 12. Pembacaan file pdf dari pengguna dan yang akan dicek (Sumber: dari penulis)

Saat melakukan pencarian, pertama-tama dilakukan pembersihan data dari masukan pengguna. Lalu dilakukan juga pembersihan data yang bersifat destruktif pada datanya. Maksudnya di sini adalah dilakukan pembersihan yang akan merubah isi dari output. Dalam hal ini, dilakukan pengubahan setiap huruf dari data menjadi huruf kecil. Setelah itu dimulailah pencarian bagian makalah yang berkemungkinan plagiarisme.

```
export function findPlagiarism(paper: Paper, data: Paper[]): PlagiarizeResult{
    const startTime = performance.now();
    const dirtyContent = paper.content;
    const content = clean(dirtyContent);
    const contentDestructed = cleanDestruct(content);

    const plagiarizedPaper = getPlagiarizedPaper(contentDestructed, data);
    const result: PlagiarizeResult = {
        selectedPaper: { title: paper.title, href: paper.href, content: content},
        plagiarizedPaper: plagiarizedPaper,
        percentage: calculateAllPercentage(contentDestructed, plagiarizedPaper),
        executionTime: 0, // must be set at the end to ensure the correct time
    };

    result.executionTime = performance.now() - startTime;
    return result;
}
```

Gambar 13. Pembersihan data pengguna dan mulai pencarian plagiarisme (Sumber: dari penulis)

Pertama, bangkitkan larik *w-shingling* dengan jumlah token sebesar 5 (SHINGLES_AMOUNT = 5). Lalu iterasi setiap makalah yang akan dicek dan simpan index saat ini. Bangkitkan larik jangkauan plagiarisme berdasarkan *w-shingling* tadi dan isi text dari makalah yang akan dicek saat ini. Jika panjangnya 0, maka skip saja ke iterasi berikutnya. Jika tidak, maka masukkan ke output dan kalkulasikan persentase plagiarisme berdasarkan makalah pengguna dan larik jangkauan plagiarism. Setelah semuanya selesai, urutkan hasilnya dari persentase terbesar ke persentasi terkecil.

```
function getPlagiarizedPaper(content: string, data: Paper[]){
    const plagiarizedPaper: PlagiarizedPaper[] = [];

    const len = data.length;
    const userShingles = shingles(content, SHINGLES_AMOUNT);
    for(let i = 0; i < len; i++){
        const plagiarizedContent = data[i].content;
        const similiarTextList = getSimilarTextList(userShingles, plagiarizedContent);
        if(similiarTextList.length === 0) continue;

        console.log("(%d/%d)", i+1, len)

        plagiarizedPaper.push({
            title: data[i].title,
            href: data[i].href,
            content: plagiarizedContent,
            plagiarizedList: similiarTextList,
            percentage: calculatePercentage(content, similiarTextList),
        });
    }

    plagiarizedPaper.sort((a, b) => b.percentage - a.percentage);
    return plagiarizedPaper;
}
```

Gambar 14. Pembangkitan larik jangkauan plagiarism dan persentase plagiarisme terhadap setiap makalah yang dicek (Sumber: dari penulis)

Dalam proses pembangkitan larik jangkauan plagiarism, pertama-tama diawali dengan pembangkitan *w-shingling* dengan jumlah token sebesar 5 untuk makalah yang sedang dicek. Lalu iterasi setiap elemen dari *w-shingling* pada makalah pengguna. Jika shingle tersebut ada atau mirip pada *w-shingling* dari makalah yang dicek, maka index saat itu merupakan *starting index* dari larik jangkauan plagiarisme, dan index + jumlah token merupakan *ending index* dari larik jangkauan plagiarism. Setelah selesai, setiap elemen pada larik jangkauan plagiarism akan di gabungkan jika ada yang overlap.

```
function getSimilarTextList(userShingles: string[], plagiarizedText: string): number[][] {
    const plagiarizedShingles = shingles(plagiarizedText, SHINGLES_AMOUNT);

    const similarText: number[][] = [];

    for(let i = 0; i < userShingles.length; i++){
        if(isShinglesListContainsShingles(plagiarizedShingles, userShingles[i])){
            similarText.push([i, i+SHINGLES_AMOUNT]);
        }
    }

    return combineOverlapping(similarText);
}
```

Gambar 15. Proses pembangkitan larik jangkauan plagiarisme (Sumber: dari penulis)

Dalam pengecekan apakah shingle ada atau mirip pada *w-shingling* dari makalah yang dicek, diimplementasikan algoritma exact string matching yaitu Knuth Morris Prat dan approximate string matching yaitu Levenshtein Distance. Prosesnya yaitu lakukan iterasi untuk setiap elemen *w-*

shingling, Jika ada yang *exact match* menggunakan algoritma Knuth Morris Prat, maka langsung kembalikan benar. Jika tidak, maka lakukan iterasi ulang dari awal menggunakan Levenshtein Distance. Dalam konteks ini, diberikan *treshold* sebesar 0.2 (LEVENSHTEIN_THRESHOLD). Maksud dari *threshold* ini yaitu, jika kedua string memiliki kemiripan sebesar 80%, maka string tersebut mirip. Cara menghitung kemiripan dari output Levenshtein Distance adalah dengan mencari terlebih dahulu panjang dari string yang lebih panjang. Ini merupakan nilai maksimum dari Levenshtein Distance. Lalu bandingkan apakah output dari Levenshtein Distance lebih kecil dari panjang string tadi dikali dengan tresholdnya. Jika lebih kecil, langsung kembalikan benar. Jika tidak ada satu pun yang mirip, maka kembalikan salah.

```
function isShinglesListContainsShingles(shinglesList: string[], shingles: string): boolean {

    // Exact
    for(let i = 0; i < shinglesList.length; i++){
        if(kmp(shinglesList[i], shingles)) return true;
    }

    // Approximate
    for(let i = 0; i < shinglesList.length; i++){
        const longest = Math.max(shinglesList[i].length, shingles.length);
        if(levenshteinDistance(shinglesList[i], shingles) <= longest * LEVENSHTEIN_THRESHOLD) return true;
    }

    return false;
}
```

Gambar 16. Implementasi Pencocokan String. (Sumber: dari penulis)

Pembangkitan *w-shingling* dilakukan dengan cukup sederhana. Split terlebih dahulu string yang ingin dikonversi menjadi larik string yang dipisah berdasarkan spasi. Iterasi setiap elemen hasil split dikurang jumlah token kurang 1. Masukkan 3 elemen pertama dari index iterasi sekarang ke sebuah larik output.

```
export function shingles(content: string, n: number): string[] {
    const shingle: Set<string> = new Set();

    const split = content.split(' ');
    const lenMinusN = split.length - n + 1;
    for(let i = 0; i < lenMinusN; i++){
        const strList = split.slice(i, i+n)
        shingle.add(strList.join(' '));
    }

    return Array.from(shingle);
}
```

Gambar 17. Pembangkitan w-shingling dari sebuah string. (Sumber: dari penulis)

Implementasi algoritma Knuth Morris Prat dimulai dari pembangkitan larik Longest Prefix Suffix (lps). Pada fungsi computeLPSArray, len merupakan variable untuk melakukan *tracking* panjang proper prefix yang juga merupakan panjang suffix terpanjang saat ini. Maksud dari proper prefix di sini adalah prefix dari sring tapi tidak termasuk string itu sendiri. i sebagai variable index yang dimulai dari 1 untuk pemrosesan lebih lanjut. len diset 0, dan elemen pertama lps juga sudah pasti 0.

Dalam pengulangan while loop, jika karakter di index i sama dengan karakter pada index len, ini artinya ditemukan proper prefix yang juga merupakan suffix. Maka increment len sebanyak 1, isi index ke i pada lps dengan nilai len saat ini. Lalu increment i setelah yang tadi untuk ke iterasi berikutnya. Tetapi jika karakter di index i tidak sama dengan karakter pada index len maka akan ada 2 kemungkinan. Jika len tidak sama dengan 0, kembali ke lps sebelumnya dengan mengubah nilai len menjadi elemen ke len-1 dari lps. Tujuannya adalah untuk menemukan proper prefix yang lebih pendek yang bisa jadi juga merupakan suffix. Kemungkina ke-2 yaitu jika nilai len saat ini sama dengan 0. Ini artinya tidak ada prefix yang cocok dengan suffix saat ini. Maka set elemen ke i pada lps menjadi 0 dan increment i untuk lanjut ke index berikutnya.

Dalam loop utama di fungsi kmp, inisialisasi variabel j dengan 0, lalu lakukan iterasi sampai panjang string a. Jika nilai dari karakter ke i dari a sama dengan karakter ke j dari b, maka increment j lalu cek apakah nilai dari j sama dengan m. Jika benar, maka string telah ditemukan. Tapi jika karakter ke i dari a beda dengan karakter ke j dari b, maka selama j lebih dari 0 dan karakter ke i beda dengan karakter ke j, geser indeks j ke nilai yang lebih kecil berdasarkan larik lps. Caranya adalah dengan set nilai j menjadi elemen ke j-1 dari larik lps. Dengan cara ini, karena larik lps menyimpan informasi panjang proper prefix yang juga merupakan suffix, maka kita bisa menghindari perbandingan karakter yang tidak perlu.

```
export function kmp(a: string, b: string): boolean {
    const n = a.length;
    const m = b.length;
    const lps = new Array(m).fill(0);
    let j = 0;

    computeLPSArray(b, m, lps);

    for(let i = 0; i < n; i++){
        if(a[i] === b[j]){
            j++;
        } else {
            while(j > 0 && a[i] !== b[j]){
                j = lps[j-1];
            }
        }

        if(j === m) return true;
    }

    return false;
}

function computeLPSArray(b: string, m: number, lps: number[]){
    let len = 0;
    lps[0] = 0;
    let i = 1;

    while(i < m){
        if(b[i] === b[len]){
            len++;
            lps[i] = len;
            i++;
        } else {
            if(len !== 0){
                len = lps[len-1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}
```

Gambar 18. Implementasi Algoritma KMP (Sumber: dari penulis)

Implementasi algoritma Levenshtein Distance dimulai dengan menginisialisasi larik mat dengan ukuran baris yaitu panjang string a, dan ukuran kolom yaitu panjang string b. Isi matriks pada baris pertama dengan nilai dari 0 sampai n dan kolom pertama dengan nilai dari 0 sampai m. Sisanya diisi dengan 0. Ini akan menunjukkan nilai Levenshtein Distance dari string yang di hilangkan hingga kosong menjadi string yang diinginkan. Kemudian mulai iterasi setiap elemen matriks kecuali elemen pertama. Untuk setiap elemen, maka mat[i][j] akan diset menjadi nilai yang paling kecil diantara *delete* yaitu nilai di atas elemen tersebut tambah satu, *insert* yaitu nilai di kiri elemen tersebut tambah 1, dan *substitute* yaitu nilai di kiri atas tambah 0 jika a[i-1] sama dengan b[i-1] atau 1 jika a[i-1] beda dengan b[i-1]. Nilai dari setiap iterasi akan seakan-akan menjalar dan memengaruhi sampai iterasi terakhir. Kemudian nilai pada elemen paling kanan bawah adalah output dari Levenshtein Distance tersebut.

```
export function levenshteinDistance(a: string, b: string): number {
    const n = a.length;
    const m = b.length;
    const mat: number[][] = new Array(n+1).fill(0).map(() => new Array(m+1).fill(0));

    for(let i = 0; i <= n; i++) mat[i][0] = i;
    for(let i = 0; i <= m; i++) mat[0][i] = i;

    for(let i = 1; i <= n; i++){
        for(let j = 1; j <= m; j++){
            mat[i][j] = Math.min(
                mat[i-1][j] + 1,
                mat[i][j-1] + 1,
                mat[i-1][j-1] + (a[i-1] === b[j-1] ? 0 : 1)
            );
        }
    }

    return mat[n][m];
}
```

Gambar 19. Algoritma Levenshtein Distance

Setelah larik jangkauan plagiarisme didapatkan, akan ada beberapa elemen yang index awal dan akhirnya *overlap* dengan elemen yang lain sehingga akan mempersulit saat ingin menggambarnya dari frontend. Ilustrasinya sebagai berikut.

Overlap: [[3,9], [13,23], [17,28], [30,33]]

Terjadi *overlap* pada elemen ke 2 dan 3. Sehingga perlu digabungkan

Combined: [[3,9], [13,28], [30,33]]

Untuk mengatasi ini maka kita perlu mengimplementasikan fungsi combineOverlapping ini. Kita mulai dulu dengan mengambil elemen pertama pada larik jangkauan plagiarisme sebagai jangkauan saat ini. Mulai dari elemen kedua, ambil *starting index* dan *ending index* pada indeks iterasi saat ini. Jika interval saat ini overlap dengan elemen jangkauan saat ini, maka geser jangkauan sekarang sehingga *ending index*nya menjadi nilai maksimum antara nilainya sendiri dengan *ending index* pada iterasi saat ini. Tapi jika tidak, maka sudah tidak ada overlap dan kita bisa memasukkan jangkauan tersebut ke larik output. saat keluar dari pengulangan, jangan lupa masukkan juga jangkauan saat ini ke larik output. Maka larik ini adalah outputnya.

```
function combineOverlapping(plagiarizedList: number[][]): number[][] {
    const result = [];
    const len = plagiarizedList.length;
    if(len === 0) return [];

    let [start, end] = plagiarizedList[0];
    for(let i = 1; i < len; i++) {
        const [s, e] = plagiarizedList[i];
        if(s <= end) {
            end = Math.max(end, e);
        } else {
            result.push([start, end]);
            [start, end] = [s, e];
        }
    }
    result.push([start, end]);
    return result;
}
```

Gambar 20. Implementasi penggabungan larik jangkauan plagirisme yang overlap (Sumber: dari penulis)

Setelah mendapatkan larik jangkauan plagiarisme, maka kita sudah bisa menghitung persentase plagiarisme sebuah text. Pastikan larik tadi sudah digabungkan jika ada yang overlap. Implementasi dari fungsi ini cukup sederhana. Pisah terlebih dahulu string yang akan dikalkulasi menjadi larik string berdasarkan spasi dan ambil panjangnya. Larik jangkauan plagiarisme tadi menyatakan setiap area dari text yang berkemungkinan plagiarisme. Jadi kita hanya perlu membandingkan total dari panjang bagian yang plagiarisme dengan panjang dari keseluruhan text. Cara menghitung total panjangnya hanya dengan mencari selisih dari elemen ke-2 dan pertama dari tiap elemennya, lalu totalnya semuanya. Outputnya adalah perbandingan tersebut dikali 100.

```
function calculatePercentage(content: string, plagiarizedList: number[][]): number {
    const length = content.split(' ').length;
    let total = 0;
    for(let i = 0; i < plagiarizedList.length; i++){
        total += plagiarizedList[i][1] - plagiarizedList[i][0];
    }
    return total / length * 100;
}
```

Gambar 21. Implementasi kalkulasi persentasi terhadap makalah (Sumber: dari penulis)

Setelah semua kalkulasi selesai, ada satu lagi yang perlu dihitung yaitu persentase plagiarisme secara keseluruhan. Perhitungan ini cukup sederhana. Kita hanya perlu menggabungkan setiap larik jangkauan plagiarisme dari setiap makalah sehingga tidak overlap. Lalu hitung persentasenya dengan cara yang sama jika dianggap sebagai satu buah makalah.

```
function calculateAllPercentage(content: string, plagiarizeResult: PlagiarizedPaper[]): number {
    if(plagiarizeResult.length === 0) return 0;

    let result = plagiarizeResult[0].plagiarizedList;
    for(let i = 0; i < plagiarizeResult.length; i++){
        result = combineOverlapping(result.concat(plagiarizeResult[i].plagiarizedList));
    }
    return calculatePercentage(content, result);
}
```

Gambar 22. Implementasi kalkulasi persentase keseluruhan (Sumber: dari penulis)

Dengan ini semua kalkulasi telah selesai dan bisa dikembalikan ke frontend. Namun masih ada tahapan ekstra yaitu frontend harus memberikan penanda bagian dari text yang ini ditandai sebagai plagiarisme. Implementasi ini

diterapkan pada fungsi getPlagiarizedBlock berikut. Tujuan fungsi ini adalah menyisipkan bagian dari text yang terindikasi plagiarisme. Ilustrasinya sebagai berikut:

Teks = “Halo nama saya adalah Budi. Saya suka bermain bola”

larik: [[1,4], [6,8]]

Hasil = “Halo <**penanda**>nama saya adalah</**penanda**> Budi. Saya <**penanda**>suka bermain</**penanda**> bola”

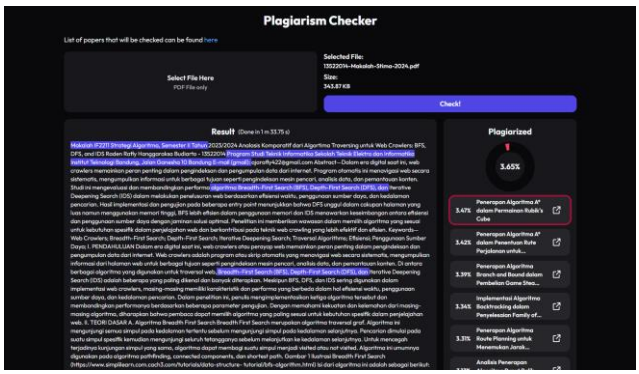
Implementasinya dimulai dari memisah teks menjadi larik kata berdasarkan spasi. Inisialisasi larik yang akan menyimpan output. Masukkan dulu elemen pertamanya yaitu kata-kata sampai ditemukan bagian plagiarisme pertama kali. Kecuali jika bagian plagiarisme langsung di elemen pertama. Pada implementasinya sudah otomatis karena menggunakan fungsi slice pada typescript. Lalu untuk setiap elemen dari larik jangkauan plagiarisme, masukkan string dari dari *starting index* sampai *ending index*nya dengan mengelilinginya dalam penanda. Lalu cek apakah elemen berikutnya belum elemen terakhir. Jika belum, masukkan string dari *ending index* jangkauan saat ini sampai *starting index* jangkauan berikutnya. Tapi jika sudah, maka masukkan string dari *ending index* jangkauan saat ini sampai akhir. Jika iterasi selesai, maka sudah didapatkan hasilnya.

```
function block(content: string){
  return <span id="marked" className="inline bg-indigo-600"> {content} </span>;
}
function getPlagiarizedBlock(content: string, plagiarizedList: number[][]): JSX.Element {
  const split = content.split(" ");
  const result = [];

  result.push(split.slice(0, plagiarizedList[0][0]).join(" "));
  for(let i = 0; i < plagiarizedList.length; i++){
    result.push(block(split.slice(plagiarizedList[i][0], plagiarizedList[i][1]).join(" ")));
    if(i+1 < plagiarizedList.length) {
      result.push(split.slice(plagiarizedList[i][1], plagiarizedList[i+1][0]).join(" "));
    } else {
      result.push(split.slice(plagiarizedList[i][1]).join(" "));
    }
  }

  return <p className="text-left text-sm font-normal">{result}</p>;
}
```

Gambar 23. Implementasi penanda bagian teks yang berkemungkinan plagiarisme (sumber: dari penulis)

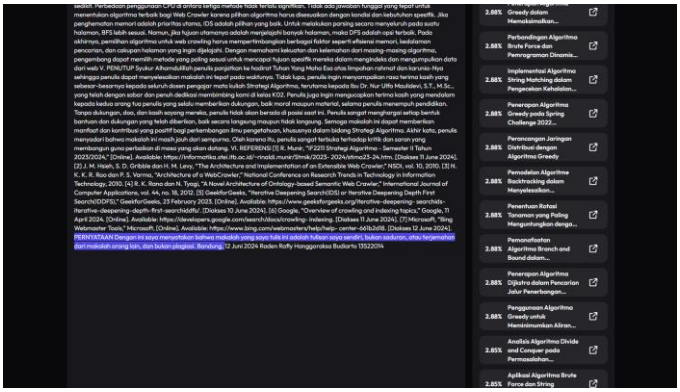
E. Pengujian Plagiarisme

Pengujian file pdf yang saat penulisan makalah ini tidak ada pada website Rinaldi Munir [10]:



Gambar 24. Pengujian makalah unik bagian 1 (Sumber: dari penulis)

Gambar 20. Pengujian unik bagian 1 (Sumber: dari penulis)



Gambar 25. Pengujian unik bagian 2 (Sumber: dari penulis)

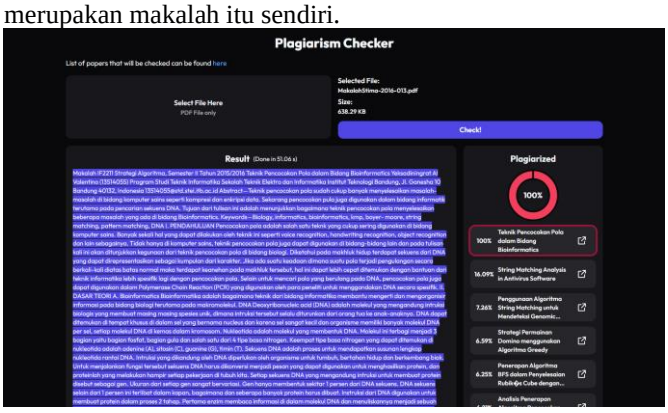
Semua text yang ditandai plagiarisme pada gambar 19, 20, dan 21 merupakan teks yang umum digunakan sehingga makalah ini tidak melakukan plagiarisme.

Selanjutnya pengujian Makalah yang memang ada pada website Rinaldi Munir. Dapat dilihat terdapat kemiripan dibagian code KMP. Ini dapt diartikan kedua makalah menggunakan code dari sumber yang sama.

Makalah Input:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2015-2016/Makalah-2016/MakalahStima-2016-013.pdf [11]

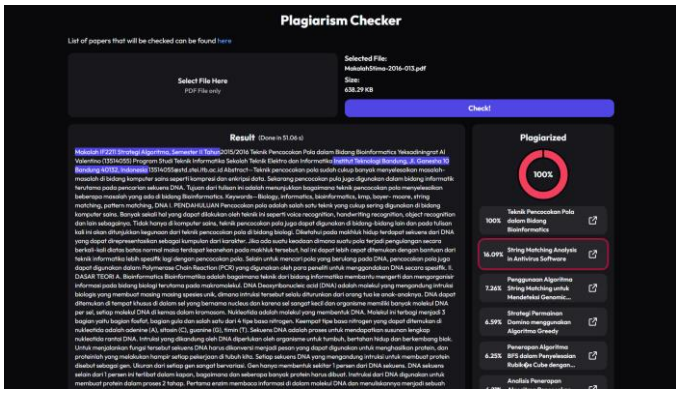
Makalah paling mirip:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2016-2017/Makalah2017/Makalah-IF2211-2017-064.pdf [12]

Untuk gambar 19, plagiarisme 100% karena memang merupakan makalah itu sendiri.



Gambar 26. Pengujian file makalah terhadap makalah itu sendiri (Sumber: dari penulis)

Untuk gambar 20, ditemukan plagiarisme sebesar 16.09%

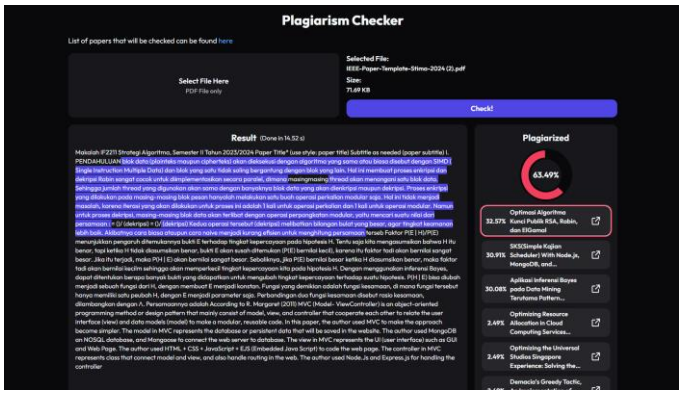


Gambar 27. Pengujian file makalah yang paling mirip tapi bukan makalah itu sendiri (Sumber: dari penulis)

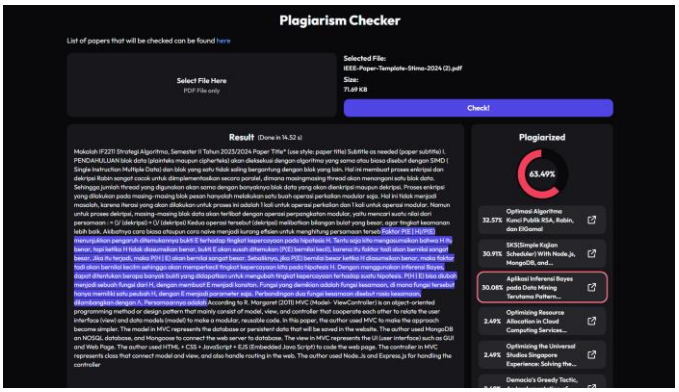
Untuk gambar 21, ditemukan bagian code yang sangat mirip dengan cukup panjang. Kemungkinan kedua makalah menggunakan code dari sumber yang sama termasuk bagian komentar.

Gambar 28. Pengujian file ditemukan bagian code mirip (Sumber: dari penulis)

Berikutnya pengujian file pdf buatan dengan copy paste secara manual beberapa bagian dari makalah lain:



Gambar 29. Pengujian manual 1 (Sumber: dari penulis)

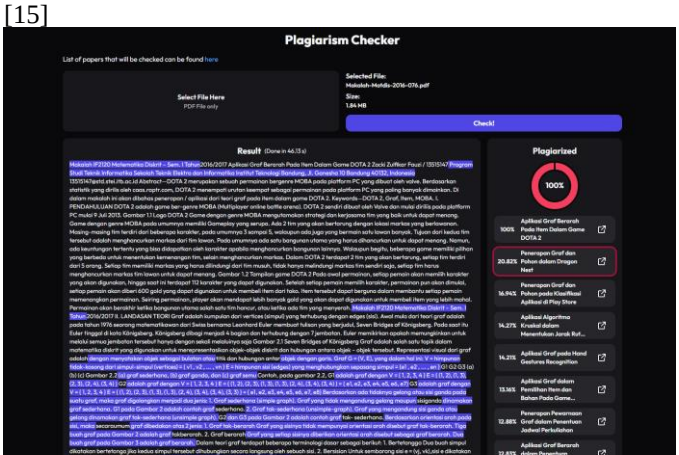


Gambar 30. Pengujian manual 2 (Sumber: dari penulis)

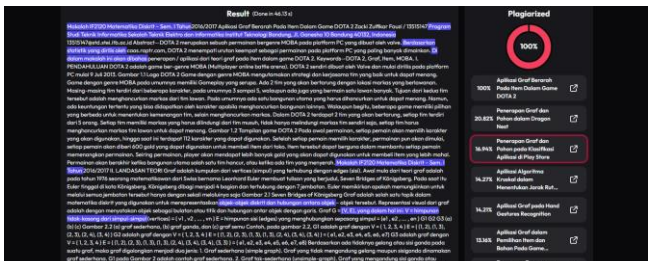
Terakhir pengujian file Makalah yang memang ada pada website Rinaldi Munir.

Makalah Input:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2016-2017/Makalah2016/Makalah-Matdis-2016-076.pdf [13]

2 Makalah paling mirip:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2013-2014/Makalah2013/MakalahIF2120-2013-024.pdf [14]
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2015-2016/Makalah-Matdis-2015/Makalah-IF2120-2015-018.pdf [15]



Gambar 31. Pengujian file ditemukan bagian dasar teori mirip (Sumber: dari penulis)



Gambar 32. Pengujian file ditemukan bagian dasar teori mirip (Sumber: dari penulis)

Dapat dilihat terdapat kemiripan dibagian dasar teori yang artinya kedua makalah menggunakan sumber dasar teori yang sama.

IV. Kesimpulan

Algoritma pencocokan string dapat digunakan untuk melakukan pengecekan plagiarisme pada sebuah makalah di website Rinaldi Munis. Algoritma utama yang digunakan diantaranya adalah W-Shingling, KMP, dan Levensthein Distance. Dari hasil pengujian juga dapat dilihat bahwa algoritma yang dilakukan cukup akurat karena dapat menemukan hal hal yang memang mirip seperti code yang dicopy dari sumber sama, ataupun bagian dasar teori yang lasung copy paste dari sumber yang sama. Jarang ditemukan bahwa mahasiswa ITB melakukan plagiarisme kepada mahasiswa lain. Namun dalam konteks dasar teori, cukup sering ditemukan ada yang mengcopy langsung dan ini merupakan hal yang wajar dan mungkin seharusnya tidak dianggap plagiarisme. Untuk pengembangan selanjutnya dapat diterapkan pencarian frekuensi unik dan umum juga agar dapat dicek bahwa beberapa kata memang merupakan kata yang umum digunakan dan tidak bisa dianggap sebagai plagiarisme. Namun agar implementasi string matching dapat terlihat berhasil, penulis memilih untuk tetap menganggapnya sebagai bagian yang ditemukan/plagiarisme (walaupun sebenarnya tidak) oleh hasil algoritma pencocokan string yang digunakan.

Dapat ditunjukkan juga bahwa dengan memanfaatkan algoritma string matching, kita dapat membuat sebuah sistem yang efisien untuk melakukan pengecekan plagiarisme secara signifikan dibandingkan melakukan pengecekan secara manual. Sistem yang telah dibuat ini juga dapat digunakan sebagai alat pendukung dalam menjaga integritas secara akademik

V. Source Code at Github

https://github.com/DhafinFawwaz/Plagiarism-Checker-Makalah

References

[1] Menteri Pendidikan Nasional, "Peraturan Menteri Pendidikan Nasional Nomor 17 Tahun 2010 tentang Pencegahan dan Penanggulangan Plagiat di Perguruan Tinggi" Diakses pada 12 Juni 2024, dari https://lpm.uin-malang.ac.id/wp-content/uploads/2018/12/Permendiknas_Pencegahan_Plagiat_2010.pdf.

[2] Dham, Mayank (29 Juni 2023), "String Matching Algorithm" Diakses pada 12 Juni 2024, dari https://www.prepbytes.com/blog/strings/string-matching-algorithm/#:~:text=String%20matching%20algorithms%20are%20fundamental,a%20larger%20text%20or%20string.

[3] Geeksforgeeks (7 November 2022), "Applications of String Matching Algorithms" Diakses pada 12 Juni 2024, dari https://www.geeksforgeeks.org/applications-of-string-matching-algorithms/

[4] Tong, Zachary (16 January 2013) "Searching with Shingles" diakses pada 12 Juni 2024, dari https://www.elastic.co/blog/searching-with-shingles

[5] Munir, Rinaldi "Pencocokan String (String/Pattern Matching)" diakses pada 12 Juni 2024, dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

[6] Grashchenko, Sergey (27 Juli 2023) "Levenshtein Distance Computation" diakses pada 12 Juni 2024, dari https://www.baeldung.com/cs/levenshtein-distance-computation#:~:text=Levenshtein%20distance%20is%20the%20smallest,insertions%2C%20deletions%2C%20and%20substitutions.

[7] Hillier, Will (13 Agustus 2021) "What Is Web Scraping? A Complete Beginner's Guide" diakses pada 12 Juni 2024, dari https://careerfoundry.com/en/blog/data-analytics/web-scraping-guide/

[8] Burns, Bill "What Is Multithreading: A Guide to Multithreaded Applications" diakses pada 12 Juni 2024, dari https://totalview.io/blog/multithreading-multithreaded-applications#what-is-a-thread-in-programming

[9] Hope, Computer (18 Oktober 2022) "Regex" diakses pada 12 Juni 2024, dari https://www.computerhope.com/jargon/r/regex.htm

[10] Budiarto, Rafly (12 Juni 2024) "Analasis Komparatif dari Algortima Traversing untuk Web Crawlers: BFS, DFS, and IDS" diakses pada 12 Juni 2024 dari https://drive.google.com/drive/u/1/folders/1-ZZnCHDg_I0K8eO0GqVQ0ZkuY6uzn59k

[11] Valentino, Yeksadiningrat (8 Mei 2016) "Teknik Pencocokan Pola dalam Bidang Bioinformatics" diakses pada 12 Juni 2024, dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2015-2016/Makalah-2016/MakalahStima-2016-013.pdf

[12] Davida, Bethea (17 Mei 2017) "String Matching Analysis in Antivirus Software" diakses pada 12 Juni 2024, dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2016-2017/Makalah2017/Makalah-IF2211-2017-064.pdf

[13] Sembodo, Ichwan (17 Desember 2013) "Penerapan Graf dan Pohon dalam Dragon Nest" diakses pada 12 Juni 2024, dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2016-2017/Makalah2016/Makalah-Matdis-2016-076.pdf

[14] Fauzi, Zacki (9 Desember 2016) "Aplikasi Graf Berarah Pada Item Dalam Game DOTA 2" diakses pada 12 Juni 2024, dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2013-2014/Makalah2013/MakalahIF2120-2013-024.pdf

[15] Qurany, Amal (10 Desember 2015) "Penerapan Graf dan Pohon pada Klasifikasi Aplikasi di Play Store" diakses pada 12 Juni 2024 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2015-2016/Makalah-Matdis-2015/Makalah-IF2120-2015-018.pdf

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, 12 Juni 2024

Dhafin Fawwaz Ikramullah dan 13522084