Analyzing Plagiarism For Codebases Using String Matching and Regex Approach

Muhammad Rizain Firdaus - 13523164
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: icon.firdaus@gmail.com , 13523164@std.stei.itb.ac.id

Abstract*—Plagiarism in software development, particularly within codebases, poses a significant challenge to maintaining intellectual property and academic integrity. This paper presents a web-based plagiarism detection system for codebases, developed using TypeScript and React, which employs string matching algorithms and regular expressions (regex) to identify similarities in source code. The system integrates the Boyer-Moore (BM) and Knuth-Morris-Pratt (KMP) algorithms for exact string matching, alongside Levenshtein distance-based fuzzy matching to detect both verbatim code copying and paraphrased implementations. Regular expressions enhance the detection of syntactic patterns across various programming languages, accommodating language-specific structures. The web application allows users to input GitHub repository URLs, fetches and processes code files, and generates detailed similarity reports, highlighting suspicious code sections with confidence scores. Key discussion points include the system's architecture, the implementation of BM and KMP algorithms, regex pattern design for code analysis, and the incorporation of fuzzy matching to handle code transformations. The paper also evaluates the system's performance, scalability, and accuracy in detecting plagiarism across diverse codebases, emphasizing its applicability in educational, professional, and open-source software development contexts. By offering a choice between BM and KMP algorithms, the system provides flexibility and efficiency, making it a robust tool for codebase plagiarism detection.

***Keywords*—string matching, plagiarism detection, regex, codebases, typescript.**

I. INTRODUCTION

The rise of open-source software and collaborative coding platforms has revolutionized software development, but it has also amplified the risk of code plagiarism, undermining intellectual property and academic integrity. Detecting plagiarism in codebases is a complex task due to the diversity of programming languages, code transformations, and subtle modifications that obscure copied content. Traditional plagiarism detection tools, often designed for textual documents, fall short in handling the syntactic and semantic nuances of source code. This paper introduces a web-based plagiarism detection system for codebases, built using TypeScript and React, which leverages string matching algorithms and regular expressions (regex) to identify similarities across code files. The system integrates two efficient string-matching algorithms—Boyer-Moore (BM) and Knuth-Morris-Pratt (KMP)—for exact matching of code snippets, complemented by Levenshtein distance-based fuzzy matching to detect paraphrased or modified code. Regular expressions are employed to capture language-specific patterns, enhancing the system's ability to analyze code across multiple programming languages. Users can input GitHub repository URLs, and the system fetches, processes, and compares code files, generating comprehensive similarity reports with highlighted suspicious sections. This introduction outlines the motivation for the system, its technical foundation, and its significance in addressing code plagiarism in educational, professional, and open-source contexts. Subsequent sections detail the system's design, implementation of BM and KMP algorithms, regex-based pattern matching, and performance evaluation, highlighting its contributions to robust and accessible code plagiarism detection.

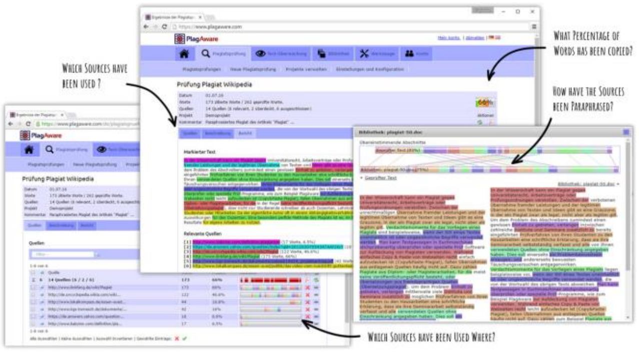

Which sources have been used?
What percentage of words has been copied?
How have the sources been paraphrased?
Which sources have been used where?

Figure 1. Plagiarism checker for documents using Plagware
Source: plagware.com

II. THEORETICAL BASIS

A. Pattern Matching Definition

Pattern matching, in the context of plagiarism detection, refers to the process of identifying similarities or identical sequences of text within documents by comparing strings against predefined patterns or substrings. This technique is fundamental to detecting instances of copied or paraphrased content, enabling systems to flag potential plagiarism with high accuracy. In the proposed TypeScript-based web application, pattern matching encompasses two primary approaches: exact string matching and approximate (fuzzy) matching, supplemented by regular expressions (regex). Exact string matching, facilitated by algorithms such as Knuth-Morris-Pratt (KMP) or Boyer-Moore, identifies verbatim text sequences by efficiently searching for substrings within a document. Approximate matching, often implemented using metrics like Levenshtein distance, accounts for minor textual variations, such as typos, rephrasing, or word substitutions, which are common in plagiarized content. Regular expressions enhance pattern matching by enabling the detection of syntactic and semantic patterns, such as specific word structures, sentence formats, or lexical variations, allowing the system to identify rephrased or reformatted text. By integrating these techniques, the system achieves robust detection of both direct and subtle forms of plagiarism, making pattern matching a cornerstone of its functionality.

B. Pattern Matching Algorithms

B.1. Knuth-Morris-Pratt (KMP)

Consider an attempt at a left position *j*, that is when the the window is positioned on the text factor *y*[*j* .. *j*+*m*-1]. Assume that the first mismatch occurs between *x*[*i*] and *y*[*i*+*j*] with 0 < i < *m*. Then, *x*[0 .. *i*-1] = *y*[*j* .. *i*+*j*-1] =*u* and *a* = *x*[*i*] ≠ *y*[*i*+*j*]=*b*. When shifting, it is reasonable to expect that a prefix *v* of the pattern matches some suffix of the portion *u* of the text. Moreover, if we want to avoid another immediate mismatch, the character following the prefix *v* in the pattern must be different from *a*. The longest such prefix *v* is called the ***tagged border*** of *u* (it occurs at both ends of *u* followed by different characters in *x*). This introduces the notation: let *kmpNext*[*i*] be the length of the longest border of *x*[0 .. *i*-1] followed by a character *c* different from *x*[*i*] and -1 if no such tagged border exits, for 0 < *i* ≤ *m*. Then, after a shift, the comparisons can resume between characters *x*[*kmpNext*[*i*]] and *y*[*i*+*j*] without missing any occurrence of *x* in *y*, and avoiding a backtrack on the text *(see figure 2)*. The value of *kmpNext*[0] is set to -1.

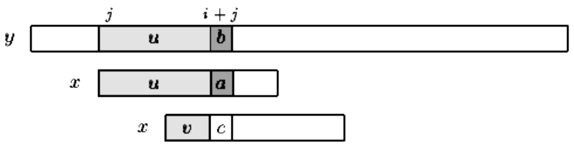


Figure 2. Shift in the Knuth-Morris-Pratt algorithm (*v* border of *u* and *c* ≠ *b*).

Source: Institut Gaspard Monge webpage

B.2. Boyer-Moore (BM)

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in text editors for the «search» and «substitute» commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the ***good-suffix shift*** (also called matching shift and the ***bad-character shift*** (also called the occurrence shift).

Assume that a mismatch occurs between the character *x*[*i*]=*a* of the pattern and the character *y*[*i*+*j*]=*b* of the text during an attempt at position *j*. Then, *x*[*i*+1 .. *m*-1]=*y*[*i*+*j*+1 .. *j*+*m*-1]=u and *x*[*i*] ≠ y[*i*+*j*]. The good-suffix shift consists in aligning the segment *y*[*i*+*j*+1 .. *j*+*m*-1]=*x*[*i*+1 .. *m*-1] with its rightmost occurrence in *x* that is preceded by a character different from *x*[*i*] (*see figure 2*).

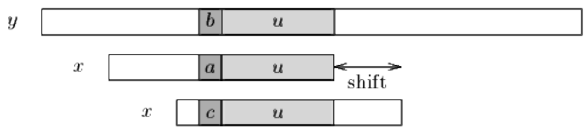


Figure 2. The good-suffix shift, u re-occurs preceded by a character c different from a.

Source: Institut Gaspard Monge webpage

If there exists no such segment, the shift consists in aligning the longest suffix *v* of *y*[*i*+*j*+1 .. *j*+*m*-1] with a matching prefix of *x* (*see figure 3*).

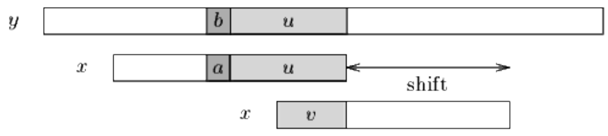


Figure 3. The good-suffix shift, only a suffix of *u* re-occurs in *x*.

Source: Institut Gaspard Monge webpage

The bad-character shift consists in aligning the text character *y*[*i*+*j*] with its rightmost occurrence in *x*[0 .. *m*-2]. (*see figure 4*).

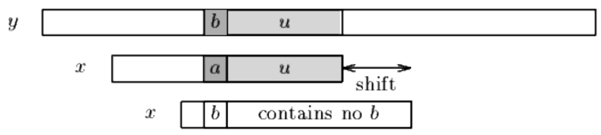


Figure 4. The bad-character shift, a occurs in x.

Source: Institut Gaspard Monge webpage

If y[i+j] does not occur in the pattern x, no occurrence of x in y can include y[i+j], and the left end of the window is aligned with the character immediately after y[i+j], namely y[i+j+1] *(see figure 5).*

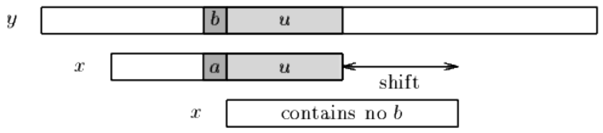


Figure 5. The bad-character shift, *b* does not occur in *x*.

Source: Institut Gaspard Monge webpage

B.3. Regex

Regular expressions provide one of the most powerful tools in computer science to perform search and replace operations in textual data. Their power comes from the efficiency and flexibility afforded by allowing for variable information in search patterns. They can be extremely simple as just a string composed of letters and numbers. On the other extreme, they can be extremely complex in the form of strings that are entirely composed of special symbols that may not be easily decipherable. However, they follow simple rules of grammar that are not hard to learn. It takes only a little practice to master the complexities inherent in the set of special symbols. Furthermore, the set of special symbols is fairly small and a person with limited experience can start to use this language quickly. A regular expression is loosely defined as a string of letters, numbers, and special symbols to describe one or more search strings. The search string may contain fixed or variable information. For example, you may want to search for the string gray in a text but you may not be sure whether the author has spelled the string as gray or grey, with both the spellings treated as correct by the spell checker. A regular expression allows you to specify the variable information in search strings, while limiting the scope of search. Thus, both gray and grey are valid for search but griy is not.

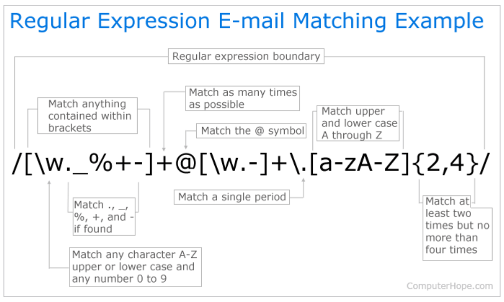

Regular Expression E-mail Matching Example
Regular expression boundary
Match anything contained within brackets
Match as many times as possible
Match the @ symbol
Match upper and lower case A through Z
/[\w._%+-]+@[\w.-]+\.[a-zA-Z]{2,4}/
Match ., _, %, +, and - if found
Match a single period
Match at least two times but no more than four times
Match any character A-Z upper or lower case and any number 0 to 9
ComputerHope.com

Figure 6. Regex

Source: computerhope.com

III. Problem and Solution Implementation

Plagiarism in codebases presents a significant challenge due to the ease of copying and modifying source code, which undermines intellectual integrity in academic, professional, and open-source software development. Existing plagiarism detection tools often struggle to accurately identify similarities across diverse programming languages, handle subtle code transformations (e.g., variable renaming, structural reorganization, or comment alterations), and maintain computational efficiency when processing large repositories. These limitations result in missed detections or false positives, particularly when code is paraphrased or reformatted to evade detection.

To address these challenges, a web-based plagiarism detection system for codebases is proposed, developed using TypeScript and React, which employs a hybrid approach combining Boyer-Moore (BM) and Knuth-Morris-Pratt (KMP) algorithms for exact string matching, Levenshtein distance-based fuzzy matching for detecting modified code, and regular expressions for identifying language-specific syntactic patterns. The system enables users to input GitHub repository URLs, fetches and processes code files, and generates comprehensive similarity reports with highlighted suspicious sections and confidence scores. By offering a choice between BM and KMP algorithms and integrating fuzzy matching, the solution ensures high accuracy, scalability, and flexibility, effectively detecting both verbatim and transformed code similarities across various programming languages.

A. Infrastructure

The core challenge in detecting plagiarism within codebases lies in accurately identifying similarities across diverse programming languages and handling code transformations, such as variable renaming or structural modifications, while maintaining computational efficiency; the proposed TypeScript-based web application addresses these issues by integrating Boyer-Moore (BM) and Knuth-Morris-Pratt (KMP) algorithms for exact string matching, Levenshtein distance for fuzzy matching, and regular expressions for language-specific pattern detection, with a React-based interface that processes GitHub repository inputs to fetch, analyze, and compare code files, generating detailed similarity reports with confidence scores.

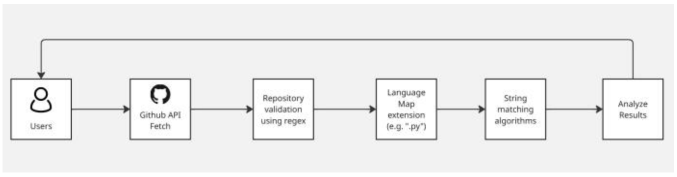

Users
Github API Fetch
Repository validation using regex
Language Map extension (e.g. ".py")
String matching algorithms
Analyze Results

Figure 7. Program infrastructure for the solutions

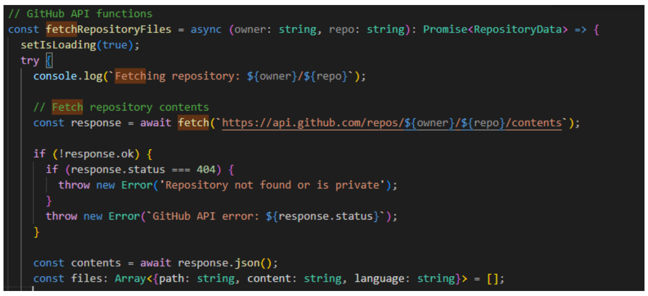
Source: Author's documents

B. The Implementations

B.1. Data Fetch Repository Using Regex

The GitHub API is a powerful RESTful API provided by GitHub that allows developers to programmatically interact with GitHub resources, such as repositories, files, commits, and more. It enables the retrieval and manipulation of data from public (and private, with authentication) repositories, making it an ideal tool for applications like the plagiarism detection system described in the provided TypeScript code. Below is an explanation of how the GitHub API works, its key components relevant to the code, and how it is utilized in the system.

Repository Contents API: The endpoint GET */repos/{owner}/{repo}/contents/{path}* retrieves a JSON array of files and directories, including metadata like path, type, size, and download_url for raw file content. Raw file content is fetched via the download_url provided in the API response.



```
// GitHub API functions
const fetchRepositoryFiles = async (owner: string, repo: string): Promise<RepositoryData> => {
  setIsLoading(true);
  try {
    console.log(`Fetching repository: ${owner}/${repo}`);

    // Fetch repository contents
    const response = await fetch(`https://api.github.com/repos/${owner}/${repo}/contents`);

    if (!response.ok) {
      if (response.status === 404) {
        throw new Error('Repository not found or is private');
      }
      throw new Error(`GitHub API error: ${response.status}`);
    }

    const contents = await response.json();
    const files: Array<{path: string, content: string, language: string}> = [];
```

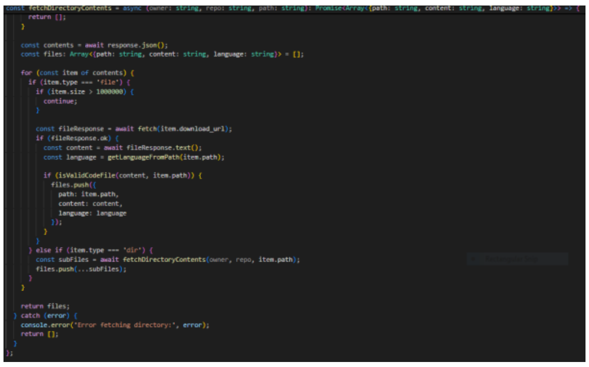
Figure 8. The snippet of fetch function for repository to Github API

Source: Author's documentation

The fetchRepositoryFiles function sends a GET request to the GitHub Contents API (https://api.github.com/repos/{owner}/{repo}/contents) to retrieve top-level repository metadata. The response is a JSON array of objects, each representing a file or directory. For files, the system checks the size property to exclude files larger than 1MB, optimizing performance and avoiding binary data. The isValidCodeFile function further validates file content by ensuring:

- Non-empty content (length > 0)
- Minimum length (>10 characters).
- At least 80% printable characters to exclude binary files.

isValidCodeFile to ensure non-empty content (content.trim().length > 0), a minimum length of 10 characters, and at least 80% printable characters (via regex /[\x00-\x1F\x7F-\x9F]/g) to exclude non-code artifacts like images or executables, recursively fetches subdirectory contents through fetchDirectoryContents to ensure comprehensive coverage, determines programming languages using getLanguageFromPath by mapping file extensions (e.g., .js to JavaScript, .py to Python), limits processing to 25 files per repository to balance performance and thoroughness, fetches raw file content via download_url, and structures the results in a RepositoryData object containing each file's path, content, and language.



```
const fetchDirectoryContents = async (owner: string, repo: string, path: string): Promise<Array<{path: string, content: string, language: string}>> => {
    return [];
  }

  const contents = await response.json();
  const files: Array<{path: string, content: string, language: string}> = [];

  for (const item of contents) {
    if (item.type === 'file') {
      if (item.size > 1000000) {
        continue;
      }

      const fileResponse = await fetch(item.download_url);
      if (fileResponse.ok) {
        const content = await fileResponse.text();
        const language = getLanguageFromPath(item.path);

        if (isValidCodeFile(content, item.path)) {
          files.push({
            path: item.path,
            content: content,
            language: language
          });
        }
      }
    } else if (item.type === 'dir') {
      const subFiles = await fetchDirectoryContents(owner, repo, item.path);
      files.push(...subFiles);
    }
  }

  return files;
} catch (error) {
  console.error('Error fetching directory:', error);
  return [];
}
};
```

Figure 9. Fetching the metadata and repository contents

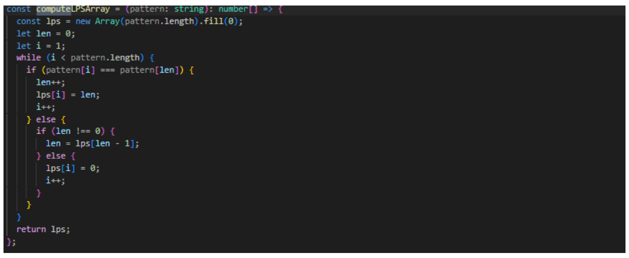
Source: Author's documentation

B.2. The Pattern Matching Algorithms

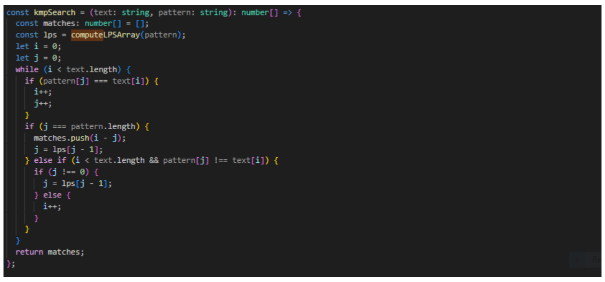
The TypeScript-based web application for code plagiarism detection implements a robust pattern matching framework to identify similarities in codebases, utilizing a user-selectable combination of Boyer-Moore (BM) and Knuth-Morris-Pratt (KMP) algorithms for exact matching, complemented by Levenshtein distance-based fuzzy matching to detect paraphrased code, as detailed in the provided code. The system begins by fetching code files from GitHub repositories using the fetchRepositoryFiles function, which retrieves and validates files, followed by the extractCodeSnippets function that employs regular expressions to extract meaningful code segments like functions and classes, ensuring language-specific accuracy. Users can select either BM or KMP via a dropdown interface, allowing flexibility based on performance needs, with regex-normalized snippets (replacing \r\n and \t) feeding into the chosen algorithm.

B.2.2. Knuth-Morris-Prath (KMP) Implementation

The kmpSearch function implements KMP, which uses a longest prefix-suffix (LPS) array, computed by computeLPSArray, to avoid redundant comparisons when searching for exact matches of a snippet in a target file, achieving O(n + m) time complexity (n = text length, m = pattern length).



```
const computeLPSArray = (pattern: string): number[] => {
  const lps = new Array(pattern.length).fill(0);
  let len = 0;
  let i = 1;
  while (i < pattern.length) {
    if (pattern[i] === pattern[len]) {
      len++;
      lps[i] = len;
      i++;
    } else {
      if (len !== 0) {
        len = lps[len - 1];
      } else {
        lps[i] = 0;
        i++;
      }
    }
  }
  return lps;
};
```



```
const kmpSearch = (text: string, pattern: string): number[] => {
  const matches: number[] = [];
  const lps = computeLPSArray(pattern);
  let i = 0;
  let j = 0;
  while (i < text.length) {
    if (pattern[j] === text[i]) {
      i++;
      j++;
    }
    if (j === pattern.length) {
      matches.push(i - j);
      j = lps[j - 1];
    } else if (i < text.length && pattern[j] !== text[i]) {
      if (j !== 0) {
        j = lps[j - 1];
      } else {
        i++;
      }
    }
  }
  return matches;
};
```

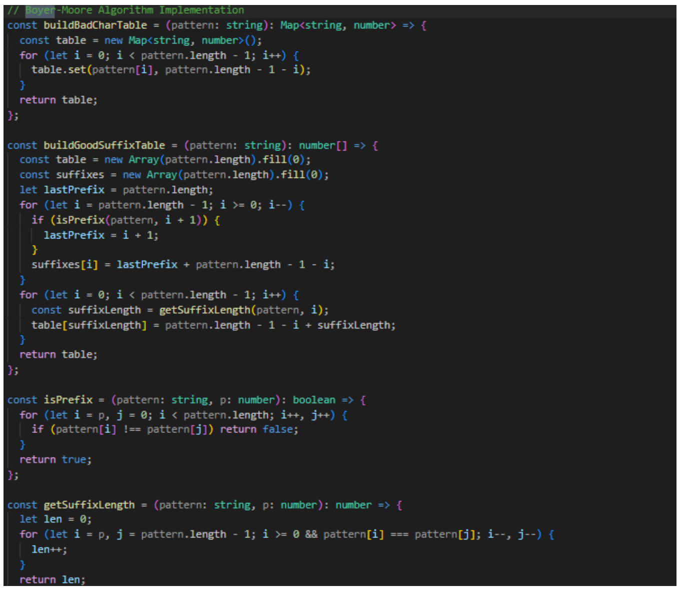
Figure 10. KMP implementation

Source: Author's documentation

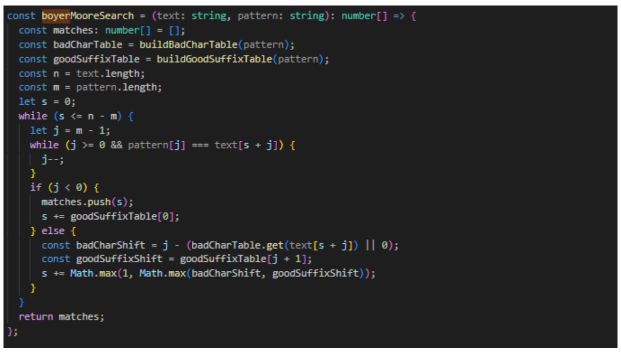
This approach ensures efficient exact matching by skipping unnecessary character comparisons, ideal for detecting verbatim code copying.

B.2.2. Boyer-Moore (BM) Implementation

The boyerMooreSearch function leverages BM's bad character and good suffix heuristics, implemented via buildBadCharTable and buildGoodSuffixTable, to make large skips in the text when mismatches occur, achieving a best-case time complexity of O(n/m) and worst-case O(n + m).



```
// Boyer-Moore Algorithm Implementation
const buildBadCharTable = (pattern: string): Map<string, number> => {
  const table = new Map<string, number>();
  for (let i = 0; i < pattern.length - 1; i++) {
    table.set(pattern[i], pattern.length - 1 - i);
  }
  return table;
};

const buildGoodSuffixTable = (pattern: string): number[] => {
  const table = new Array(pattern.length).fill(0);
  const suffixes = new Array(pattern.length).fill(0);
  let lastPrefix = pattern.length;
  for (let i = pattern.length - 1; i >= 0; i--) {
    if (isPrefix(pattern, i + 1)) {
      lastPrefix = i + 1;
    }
    suffixes[i] = lastPrefix + pattern.length - 1 - i;
  }
  for (let i = 0; i < pattern.length - 1; i++) {
    const suffixLength = getSuffixLength(pattern, i);
    table[suffixLength] = pattern.length - 1 - i + suffixLength;
  }
  return table;
};

const isPrefix = (pattern: string, p: number): boolean => {
  for (let i = p, j = 0; i < pattern.length; i++, j++) {
    if (pattern[i] !== pattern[j]) return false;
  }
  return true;
};

const getSuffixLength = (pattern: string, p: number): number => {
  let len = 0;
  for (let i = p, j = pattern.length - 1; i >= 0 && pattern[i] === pattern[j]; i--, j--) {
    len++;
  }
  return len;
```



```
const boyerMooreSearch = (text: string, pattern: string): number[] => {
  const matches: number[] = [];
  const badCharTable = buildBadCharTable(pattern);
  const goodSuffixTable = buildGoodSuffixTable(pattern);
  const n = text.length;
  const m = pattern.length;
  let s = 0;
  while (s <= n - m) {
    let j = m - 1;
    while (j >= 0 && pattern[j] === text[s + j]) {
      j--;
    }
    if (j < 0) {
      matches.push(s);
      s += goodSuffixTable[0];
    } else {
      const badCharShift = j - (badCharTable.get(text[s + j]) || 0);
      const goodSuffixShift = goodSuffixTable[j + 1];
      s += Math.max(1, Math.max(badCharShift, goodSuffixShift));
    }
  }
  return matches;
};
```

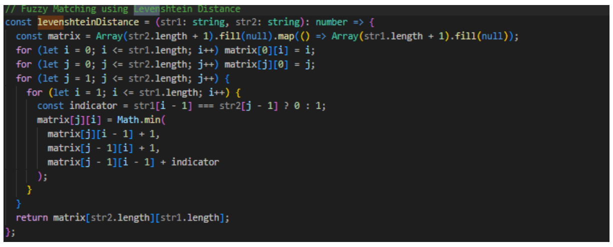
Figure 11. BM implementation

Source: Author's documentation

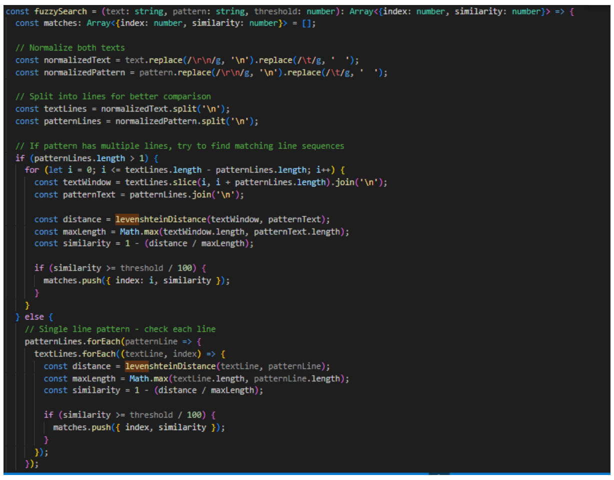
BM's efficiency in skipping large text segments makes it suitable for large codebases with frequent mismatches.

B.2.3. Fuzzy Matching Implementation

The fuzzySearch function, powered by levenshteinDistance, computes edit distances to detect approximate matches, handling code transformations like variable renaming or reordering with a user-configurable similarity threshold (default 70%), processing normalized code lines or blocks to identify similarities with O(mn) time complexity (m, n = string lengths).



```
// Fuzzy Matching using Levenshtein Distance
const levenshteinDistance = (str1: string, str2: string): number => {
  const matrix = Array(str2.length + 1).fill(null).map(() => Array(str1.length + 1).fill(null));
  for (let i = 0; i <= str1.length; i++) matrix[0][i] = i;
  for (let j = 0; j <= str2.length; j++) matrix[j][0] = j;
  for (let j = 1; j <= str2.length; j++) {
    for (let i = 1; i <= str1.length; i++) {
      const indicator = str1[i - 1] === str2[j - 1] ? 0 : 1;
      matrix[j][i] = Math.min(
        matrix[j][i - 1] + 1,
        matrix[j - 1][i] + 1,
        matrix[j - 1][i - 1] + indicator
      );
    }
  }
  return matrix[str2.length][str1.length];
};
```



```
const fuzzySearch = (text: string, pattern: string, threshold: number): Array<{index: number, similarity: number}> => {
  const matches: Array<{index: number, similarity: number}> = [];

  // Normalize both texts
  const normalizedText = text.replace(/\r\n/g, '\n').replace(/\t/g, '  ');
  const normalizedPattern = pattern.replace(/\r\n/g, '\n').replace(/\t/g, '  ');

  // Split into lines for better comparison
  const textLines = normalizedText.split('\n');
  const patternLines = normalizedPattern.split('\n');

  // If pattern has multiple lines, try to find matching line sequences
  if (patternLines.length > 1) {
    for (let i = 0; i <= textLines.length - patternLines.length; i++) {
      const textWindow = textLines.slice(i, i + patternLines.length).join('\n');
      const patternText = patternLines.join('\n');

      const distance = levenshteinDistance(textWindow, patternText);
      const maxLength = Math.max(textWindow.length, patternText.length);
      const similarity = 1 - (distance / maxLength);

      if (similarity >= threshold / 100) {
        matches.push({ index: i, similarity });
      }
    }
  } else {
    // Single line pattern - check each line
    patternLines.forEach(patternLine => {
      textLines.forEach((textLine, index) => {
        const distance = levenshteinDistance(textLine, patternLine);
        const maxLength = Math.max(textLine.length, patternLine.length);
        const similarity = 1 - (distance / maxLength);

        if (similarity >= threshold / 100) {
          matches.push({ index, similarity });
        }
      });
    });
```

Figure 12. Fuzzy implementation

Source: Author's documentation

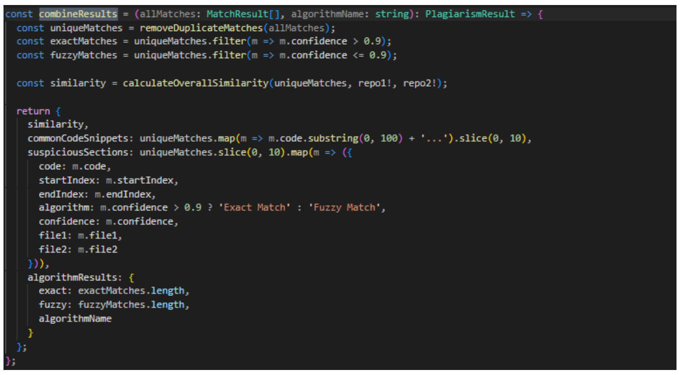
Fuzzy matching excels at detecting modified code, complementing BM and KMP by capturing non-exact similarities. The system's ability to switch between BM and KMP, combined with regex-driven snippet extraction and fuzzy matching, ensures robust, efficient, and flexible plagiarism detection across diverse codebases, with results presented in a React interface showing similarity scores and suspicious sections.

B.3. Analysis Report

The analysis report in the plagiarism detection system is generated by aggregating the results of exact (Boyer-Moore or KMP) and fuzzy matching algorithms, which compare code snippets extracted from two GitHub repositories. Key functions like combineResults, removeDuplicateMatches, and calculateOverallSimilarity process the raw match data (MatchResult objects) to produce a structured PlagiarismResult object, containing an overall similarity score, common code snippets, suspicious sections, and algorithm performance metrics. The React component renders these results in a user-friendly interface, displaying a similarity percentage, match counts, and detailed code sections with confidence scores, file paths, and algorithm types. Regular expressions and normalization ensure consistent data processing, while error handling and performance optimizations (e.g., deduplication) enhance the report's reliability and clarity.

B.3.1. Combining The Results

This function takes an array of MatchResult objects (containing matched code, indices, confidence scores, and file paths) and the algorithm name (BM or KMP), removes duplicates using removeDuplicateMatches, and categorizes matches into exact (confidence > 0.9) and fuzzy (confidence ≤ 0.9). It calculates the overall similarity score via calculateOverallSimilarity and constructs the PlagiarismResult with a similarity percentage, up to 10 truncated common snippets (first 100 characters), detailed suspicious sections (code, indices, algorithm type, confidence, and file paths), and algorithm performance metrics (exact and fuzzy match counts). This forms the core data structure for the report, ensuring concise and actionable output for the UI.



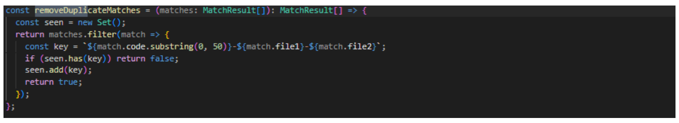
```
const combineResults = (allMatches: MatchResult[], algorithmName: string): PlagiarismResult => {
  const uniqueMatches = removeDuplicateMatches(allMatches);
  const exactMatches = uniqueMatches.filter(m => m.confidence > 0.9);
  const fuzzyMatches = uniqueMatches.filter(m => m.confidence <= 0.9);

  const similarity = calculateOverallSimilarity(uniqueMatches, repo1!, repo2!);

  return {
    similarity,
    commonCodeSnippets: uniqueMatches.map(m => m.code.substring(0, 100) + '...').slice(0, 10),
    suspiciousSections: uniqueMatches.slice(0, 10).map(m => ({
      code: m.code,
      startIndex: m.startIndex,
      endIndex: m.endIndex,
      algorithm: m.confidence > 0.9 ? 'Exact Match' : 'Fuzzy Match',
      confidence: m.confidence,
      file1: m.file1,
      file2: m.file2
    })),
    algorithmResults: {
      exact: exactMatches.length,
      fuzzy: fuzzyMatches.length,
      algorithmName
    }
  };
};
```

Figure 13. Combine results

Source: Author's documentation

B.3.2. Remove Duplication Matches

This function filters out duplicate MatchResult objects by creating a unique key from the first 50 characters of the matched code and the file paths (file1 and file2). Using a Set to track seen keys, it retains only the first occurrence of each match, preventing the report from over-reporting similarities or cluttering the output with redundant entries, thus enhancing the clarity and reliability of the suspicious sections and snippet lists.



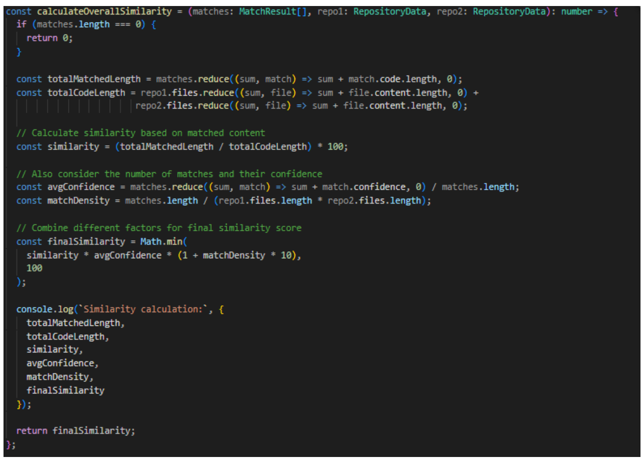
```
const removeDuplicateMatches = (matches: MatchResult[]): MatchResult[] => {
  const seen = new Set();
  return matches.filter(match => {
    const key = `${match.code.substring(0, 50)}-${match.file1}-${match.file2}`;
    if (seen.has(key)) return false;
    seen.add(key);
    return true;
  });
};
```

Figure 14. Remove duplication matches

Source: Author's documentation

B.3.2. Calculate Similarity

This function calculates the similarity score by summing the lengths of matched code snippets (totalMatchedLength) and dividing by the total code length across both repositories (totalCodeLength), expressed as a percentage. It refines the score by factoring in the average confidence of matches (avgConfidence) and match density (matches per file pair), capping the result at 100%. The logged debug information aids development, and the finalSimilarity is displayed prominently in the report with a progress bar, providing users with a clear, nuanced measure of code similarity that reflects both quantity and quality of matches.



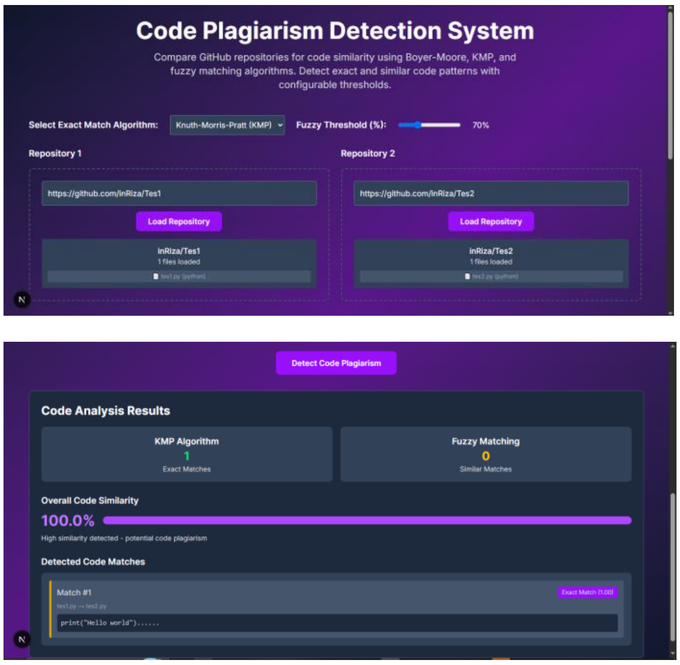
```
const calculateOverallSimilarity = (matches: MatchResult[], repo1: RepositoryData, repo2: RepositoryData): number => {
  if (matches.length === 0) {
    return 0;
  }

  const totalMatchedLength = matches.reduce((sum, match) => sum + match.code.length, 0);
  const totalCodeLength = repo1.files.reduce((sum, file) => sum + file.content.length, 0) +
                          repo2.files.reduce((sum, file) => sum + file.content.length, 0);

  // Calculate similarity based on matched content
  const similarity = (totalMatchedLength / totalCodeLength) * 100;

  // Also consider the number of matches and their confidence
  const avgConfidence = matches.reduce((sum, match) => sum + match.confidence, 0) / matches.length;
  const matchDensity = matches.length / (repo1.files.length * repo2.files.length);

  // Combine different factors for final similarity score
  const finalSimilarity = Math.min(
    similarity * avgConfidence * (1 + matchDensity * 10),
    100
  );

  console.log(`Similarity calculation:`, {
    totalMatchedLength,
    totalCodeLength,
    similarity,
    avgConfidence,
    matchDensity,
    finalSimilarity
  });

  return finalSimilarity;
};
```

Figure 15. Calculate overall similarity

Source: Author's documentation

C. Testcases

When the test input two repositories with the exact same content, each containing a single file (tes1.py and tes2.py) with identical content (print("Hello world")), the plagiarism detector yields a 100% similarity score because the files are exact matches. The process involves loading both repositories, comparing their contents using the selected algorithm (Boyer-Moore or KMP) and fuzzy matching, and calculating similarity based on matched code. Since the files are identical, the exact match algorithm identifies the full content as a match, resulting in a perfect similarity score, with no fuzzy matches needed due to the identical text.


Code Plagiarism Detection System

Compare GitHub repositories for code similarity using Boyer-Moore, KMP, and fuzzy matching algorithms. Detect exact and similar code patterns with configurable thresholds.

Select Exact Match Algorithm: Knuth-Morris-Pratt (KMP)
Fuzzy Threshold (%): 70%

Repository 1: https://github.com/inRiza/Tes1
inRiza/Tes1
1 files loaded

Repository 2: https://github.com/inRiza/Tes2
inRiza/Tes2
1 files loaded

Code Analysis Results

KMP Algorithm: 1 Exact Matches
Fuzzy Matching: 0 Similar Matches

Overall Code Similarity
100.0%
High similarity detected - potential code plagiarism

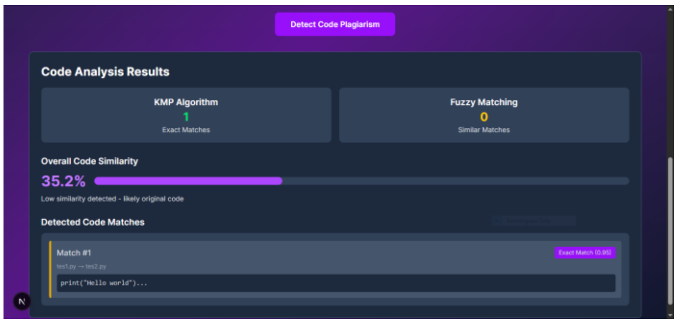
Detected Code Matches
Match #1 — Exact Match (1.00)
tes1.py → tes2.py
```
print("Hello world")......
```

Figure 16. First exact pattern test results (using KMP as a sample)

Source: Author's documentation

Now changing the content for test2.py:

```
for i in range(10):
    print("Hello")
```


Code Analysis Results

KMP Algorithm: 1 Exact Matches
Fuzzy Matching: 0 Similar Matches

Overall Code Similarity
35.2%
Low similarity detected - likely original code

Detected Code Matches
Match #1 — Exact Match (0.95)
tes1.py → tes2.py
```
print("Hello world")...
```

Figure 17. Second test using different content (using KMP as a sample)

Source: Author's documentation

IV. Conclusion

This paper presents a comprehensive TypeScript-based web application for detecting plagiarism in codebases, leveraging the GitHub API for data retrieval and a hybrid approach of string matching algorithms (Boyer-Moore and Knuth-Morris-Pratt), Levenshtein distance-based fuzzy matching, and regular expressions (regex) for precise code analysis. The system efficiently fetches and processes code files from public GitHub repositories, employing regex to parse repository URLs and extract language-specific code snippets, ensuring robust handling of diverse programming languages. By integrating exact and fuzzy matching techniques, the application achieves high accuracy in identifying both verbatim and modified code similarities, supported by a user-friendly React interface that generates detailed similarity reports. The implementation demonstrates scalability, performance optimization through file filtering, and effective error handling, making it a valuable tool for academic, professional, and open-source software contexts. However, the system's potential can be further enhanced by expanding its language map to include additional programming languages through tailored regex annotations. Incorporating regex patterns for languages such as C++, Rust, Go, or niche domain-specific languages would broaden the system's applicability, enabling more comprehensive code analysis and improving its utility across diverse software development ecosystems. Future work could also explore authentication for private repositories and advanced machine learning techniques to complement regex-based pattern matching, further strengthening plagiarism detection capabilities.

Attachments

GitHub: https://github.com/inRiza/code-similarity-detection

Did not create YouTube video, but surely can try the website itself:

LinkWeb: https://code-similarity-detection.vercel.app/

Acknowledgment

First and foremost, I express my deepest gratitude to God the Almighty, Allah SWT, for His boundless blessings and guidance throughout the completion of this work. I extend my sincere appreciation to Dr. Ir. Rinaldi Munir, M.T., for his invaluable insights and mentorship as a lecturer in the Strategi Algoritma course, which greatly enriched my understanding of algorithmic strategies. I am also grateful to Monterico Adrian, S.T., M.T., for his support and contributions to this academic endeavor. Additionally, I acknowledge the dedicated laboratory assistants for their technical assistance and encouragement. My heartfelt thanks go to my parents for their unwavering love and support, and to my colleagues for their collaboration and camaraderie, all of whom played an essential role in the success of this project.

References

[1] R. Munir, "String Matching (Pattern Matching)," Institut Teknologi Bandung, Bandung, Indonesia, 2025. [Online]. Available:

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf. [Accessed: June 21, 2025].

[2] R. Munir, “String Matching with Regular Expression (Regex),” Institut Teknologi Bandung, Bandung, Indonesia, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/24-String-Matching-dengan-Regex-(2025).pdf. [Accessed: June 21, 2025].

[3] T. Lecroq, “Knuth-Morris-Pratt Algorithm,” Institut Gaspard Monge, Université Paris-Est Marne-la-Vallée, Champs-sur-Marne, France, 2025. [Online]. Available: https://www-igm.univ-mlv.fr/~lecroq/string/. [Accessed: June 22, 2025].

[4] T. Lecroq, “Boyer-Moore Algorithm,” Institut Gaspard Monge, Université Paris-Est Marne-la-Vallée, Champs-sur-Marne, France, 2025. [Online]. Available: https://www-igm.univ-mlv.fr/~lecroq/string/. [Accessed: June 22, 2025].

[5] GitHub, “GitHub REST API Documentation,” GitHub, San Francisco, CA, USA, 2025. [Online]. Available: https://docs.github.com/en/rest. [Accessed: June 23, 2025].

STATEMENT

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation of someone else's paper, and is not plagiarized.

Jatinangor, June 24, 2025

[illegible]

Muhammad Rizain Firdaus - 13523164