Integrating Regex, Greedy Algorithms, and Levenshtein Distance for Enhanced Search Functionality on Stack Overflow

Filbert - 13522021
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522021@std.stei.itb.ac.id

***Abstract*— In the digital age, the efficiency of search mechanisms in technical forums such as Stack Overflow is crucial for enhancing user experience and providing quick access to relevant information. This paper introduces a sophisticated approach that combines regular expressions (regex), greedy algorithms, and Levenshtein Distance to refine search query results on Stack Overflow. Regular expressions are employed to parse and extract meaningful patterns from user queries, while greedy algorithms optimize the selection process for the most relevant answers by maximizing matching criteria. Furthermore, the Levenshtein Distance is utilized to correct and suggest potential query refinements, thereby accommodating common typographical errors and improving the accuracy of search results. Our methodology demonstrates significant improvements in retrieving precise and contextually relevant information, thereby reducing search time and enhancing user satisfaction. The integration of these three techniques provides a robust framework for improving search functionalities in community-driven Q&A websites, where quick access to accurate information is highly valued.**

***Keywords*—Regular Expressions; Greedy Algorithms; Levenshtein Distance; Search Optimization; Stack Overflow**

I. Introduction

In the ever-expanding domain of software development and IT, forums like Stack Overflow have become indispensable resources for professionals and enthusiasts alike. Stack Overflow, in particular, stands out as one of the most prominent technical forums globally, offering a vast repository of questions and answers that cater to a wide range of programming and software development issues. As of the latest statistics, Stack Overflow hosts millions of questions and has become a critical tool for developers seeking to resolve technical challenges swiftly.

However, despite its comprehensive database and user-friendly design, finding the most relevant information on Stack Overflow can sometimes be a daunting task. Users often rely on the platform's search functionality to find answers to their specific problems. This process involves typing queries into the search bar and sifting through the results presented. The effectiveness of this search is crucial as it directly impacts the user's ability to find helpful information quickly. Current search mechanisms primarily depend on keyword matching, which can lead to two significant issues: the omission of relevant results due to the rigid nature of keyword searches and the failure to retrieve correct information if the query contains typographical errors.

The first issue arises from the inherent limitations of keyword-based search algorithms, which rely heavily on the exact terms entered by the user. If a user does not know the precise terminology used in relevant discussions or if the query is not perfectly aligned with the language in the database, the search results may not be entirely pertinent. This problem is exacerbated by the diverse ways in which a single technical concept can be described, leading to variations in phrasing and terminology across different posts.

The second issue pertains to the sensitivity of keyword searches to typographical errors. Stack Overflow's current search engine, like many others, lacks a robust mechanism for handling misspellings or slightly incorrect queries. Consequently, a user making a small typo or not knowing the exact correct spelling of technical terms might end up with no useful results. In a field where precise terminology is often crucial, the ability to recognize and correct user input errors or suggest closely related terms can significantly enhance search effectiveness.

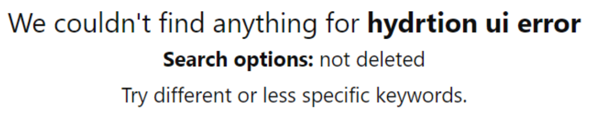

We couldn't find anything for **hydrtion ui error**
Search options: not deleted
Try different or less specific keywords.

Figure 1 Example searching not found

Source : https://stackoverflow.com/search?q=hydrtion+ui+error

To address these challenges, this paper proposes a novel approach that integrates regular expressions (regex), greedy algorithms, and the Levenshtein Distance(LD) into the search functionality of Stack Overflow. The use of regular expressions allows for flexible pattern matching, which can accommodate variations in phrasing and terminology. Greedy algorithms improve the efficiency and relevance of the search results by prioritizing matches that meet the most criteria first.

Meanwhile, the Levenshtein Distance provides a method for handling typographical errors by measuring the difference between strings and suggesting the nearest correct terms.

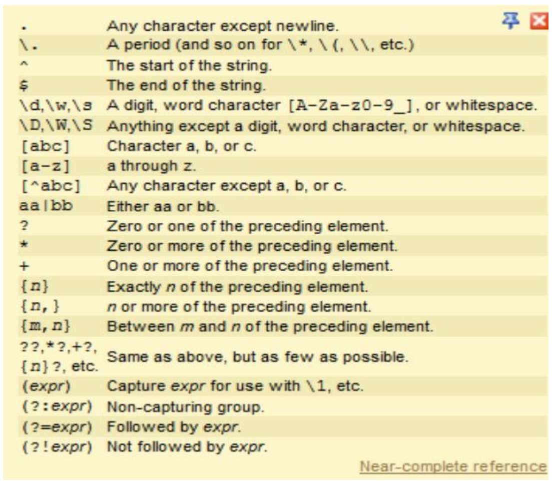
This integrated approach aims to revolutionize the search process on Stack Overflow, ensuring that users not only find what they are looking for more quickly and accurately but also discover valuable content that they may not have uncovered through conventional search techniques. By enhancing the platform's ability to handle varied input and correct errors, This program can significantly improve user satisfaction and access to information. This paper will detail the methodology, implementation, and testing of these enhancements, alongside a comprehensive analysis of their impact on search quality and user experience on Stack Overflow.

II. THEORY AND CONCEPTS

A. *Regular Expressions*

Regular expressions, commonly abbreviated as regex, serve as a robust tool for efficient pattern matching and manipulation of strings. They are built on a syntactic framework that allows for precise identification of string patterns through a composed sequence of characters. This sequence, termed a pattern, can match various combinations of characters within a text, making regex invaluable for tasks like searching, replacing, and parsing data in text processing or programming environments.

The functionality of regex is derived from its use of a mixture of symbols and characters that denote specific meanings. For example, character classes like [a-z] match any lowercase letter, while quantifiers like * (indicating zero or more repetitions of the preceding element) and + (one or more repetitions) adjust the breadth of matches. Anchors (^ and $) are employed to signify the start and end of a string, respectively, ensuring that the pattern matches in specific positions. Beyond simple matches, regex can capture subsets of strings using parentheses for grouping, which facilitates complex substitutions and rearrangements of data within strings.



`.`	Any character except newline.
`\.`	A period (and so on for `\*`, `\(`, `\\`, etc.)
`^`	The start of the string.
`$`	The end of the string.
`\d,\w,\s`	A digit, word character `[A-Za-z0-9_]`, or whitespace.
`\D,\W,\S`	Anything except a digit, word character, or whitespace.
`[abc]`	Character a, b, or c.
`[a-z]`	a through z.
`[^abc]`	Any character except a, b, or c.
`aa|bb`	Either aa or bb.
`?`	Zero or one of the preceding element.
`*`	Zero or more of the preceding element.
`+`	One or more of the preceding element.
`{n}`	Exactly *n* of the preceding element.
`{n,}`	*n* or more of the preceding element.
`{m,n}`	Between *m* and *n* of the preceding element.
`??,*?,+?,` `{n}?`, etc.	Same as above, but as few as possible.
(*expr*)	Capture *expr* for use with `\1`, etc.
(?:*expr*)	Non-capturing group.
(?=*expr*)	Followed by *expr*.
(?!*expr*)	Not followed by *expr*.

Near-complete reference

Figure 2. Regex attribute
Source: [4]

Regex also get several function to manipulate text, including:

- **Matching**: Identifying specific patterns within text using functions like match() or findall(). These functions allow for the detection and retrieval of patterns that meet predefined criteria, aiding in tasks such as data extraction and keyword detection.
- **Validation**: Ensuring that a text conforms to a specified pattern using functions such as search() or match(). Validation is crucial in scenarios like form input where the correctness of data, such as phone numbers or email addresses, needs to be confirmed against a standardized format.
- **Replacement**: Substituting matched patterns with new text using functions such as sub() or subn(). This is often used in editing texts to update specific information or to anonymize sensitive data within a body of text by replacing names, identifiers, or other personal information with generic placeholders.
- **Splitting**: Dividing text into segments based on a particular pattern using functions like split(). This method is particularly useful for parsing data, such as extracting words, phrases, or other components from structured or semi-structured text formats.

Internally, regex patterns are processed by regex engines that can implement either deterministic or nondeterministic finite automata to efficiently find matches. These engines parse the regex pattern and execute the search across the targeted string, applying the pattern rules sequentially to determine matches. The sophistication of regex in handling complex string operations, combined with its integration in many programming languages and tools, underscores its utility in diverse computational tasks.

B. *Greedy Algorithm*

The greedy algorithm is a straightforward yet powerful approach used primarily in optimization problems where a local optimum is selected at each step with the hope of finding a global optimum. It builds the solution incrementally, making the locally optimal choice at each stage without regard for future consequences. This method is particularly effective in scenarios where the local optimum at each stage aligns with the global optimum, such as in the cases of constructing a minimal spanning tree or solving the coin change problem.

In practice, the greedy algorithm follows a systematic process defined by several key elements:

- Candidate Set (C): This is the set containing all the options available for selection at each step. It provides the pool from which choices are made in the pursuit of an optimal solution.

- Solution Set (S): This set accumulates the chosen candidates that collectively form the evolving solution to the problem.

- Solution Function: This function determines whether a chosen subset of candidates successfully constitutes a complete solution to the problem.
- Selection Function: This function is crucial for implementing the greedy strategy. It selects the most promising candidate based on a specific heuristic that typically aims to maximize or minimize certain attributes immediately.
- Feasibility Function: This function evaluates whether a candidate, once selected, can feasibly be added to the solution set without violating problem constraints.
- Objective Function: This function seeks to maximize or minimize a variable or set of variables within the problem, thus defining the ultimate goal of the algorithm.

The greedy algorithm continues to evaluate until no further selections can be made. The key to a greedy algorithm's success lies in its selection strategy, which must guarantee the attainment of an optimal solution. However, its main drawback is that it doesn't always produce the optimal solution for all problems, particularly those requiring future considerations overlooked by a purely local perspective.

A classic example of a greedy algorithm is the problem of making change using the fewest coins possible. Imagine you are a cashier and need to give someone change using the least number of coins. Suppose you have an unlimited supply of quarters (25 cents), dimes (10 cents), nickels (5 cents), and pennies (1 cent), and you need to give 99 cents in change. The greedy approach would start by selecting the coin with the highest value that does not exceed the remaining amount of change needed. You would first use three quarters (75 cents), reducing the remaining change to 24 cents. Next, you would use two dimes (20 cents), bringing the remaining change down to 4 cents. Then, you would use four pennies to make up the remaining amount. This method quickly and effectively minimizes the number of coins, as at each step, it opts for the largest possible denomination that can be used, which is a hallmark of greedy algorithms. This approach works perfectly for currencies like the U.S. dollar, where smaller denominations are always fractions of larger ones, ensuring that the greedy solution is also the optimal solution.

C. *Levenshtein Distance Algorithm*

The Levenshtein Distance Algorithm quantifies the similarity between two strings by measuring the minimum number of single-character edits required to transform one string into the other. This metric is particularly useful in fields such as natural language processing, where it aids in tasks like spell checking and phonetic comparisons, and in bioinformatics, particularly in DNA analysis.

The algorithm operates by constructing a matrix where each cell (i, j) represents the Levenshtein distance between the first i characters of one string and the first j characters of another. Initialization of this matrix starts with setting the first row and column to represent the incremental transformation of each string from an empty string to the full string. As the matrix populates, each cell is filled based on the operations required to match the characters of both strings up to that point, choosing the lowest cost among insertion, deletion, and substitution. The cost is determined by whether the characters at the current position in the two strings match. If they do, the cost is zero (no operation needed); otherwise, it is one (one operation needed). The final value at the bottom-right corner of the matrix provides the total number of edits needed.

For instance, let's consider the strings "kitten" and "sitting". The Levenshtein Distance between these two strings can be calculated as follows:

1. Substitution: Change 'k' in "kitten" to 's' to get "sitten".
2. Substitution: Change 'e' in "sitten" to 'i' to get "sittin".
3. Insertion: Add 'g' at the end of "sittin" to get "sitting".

Thus, it takes three operations to transform "kitten" into "sitting". The Levenshtein Distance between these two words is 3.

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

Figure 3 LD definition
Source : https://en.wikipedia.org/wiki/Levenshtein_distance

This algorithm's complexity is generally O(mn), where m and n are the lengths of the two strings, making it computationally intensive for very long strings without optimization strategies like trimming equal substrings from the start or end of the strings.

D. *Caching Mechanisms*

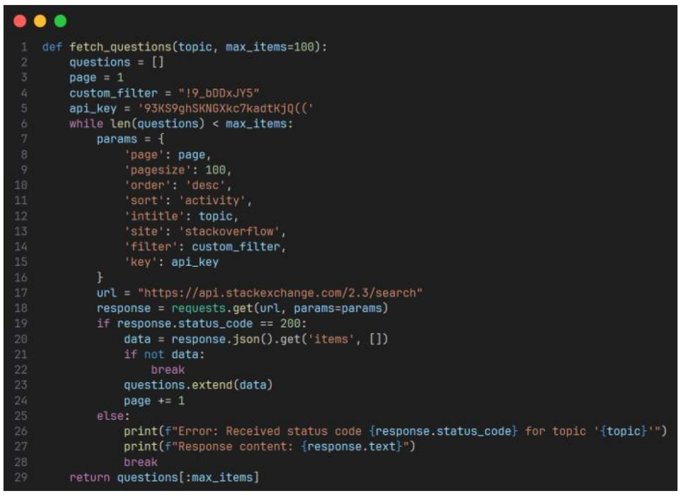
Caching is a technique to temporarily store copies of data in locations of faster access. Caching is critical in performance optimization, especially where operations involve expensive or frequently accessed data. In computational algorithms, caching can significantly reduce the time complexity by avoiding redundant recalculations.

In the context of the Levenshtein Distance and other computationally intensive operations, caching previously computed results of function calls (memoization) prevents the need to recompute them for every query, thereby enhancing performance. Using a Least Recently Used (LRU) cache mechanism can be particularly effective in environments like Stack Overflow, where certain queries may be more common than others. The cache stores a limited number of the most recently used queries and their results, discarding the least recently used if the cache exceeds its capacity limit.

III. IMPLEMENTATION

The process of enhancing Stack Overflow's search functionality can be abstracted into several stages, mirroring the complexity of the problem and requiring a systematic approach to its solution. Initially, the challenge is translated into a programmable format. This implementation involves multiple components: the core algorithms (Regex, Greedy, and Levenshtein Distance) and their integration within a cohesive system designed to handle and improve search queries effectively. Each component serves a unique role in refining the search process, from initial query processing to the final delivery of optimized results. Below, we detail the implementation stages and describe how each algorithm is utilized within the broader system architecture.

Data Gathering



```python
def fetch_questions(topic, max_items=100):
    questions = []
    page = 1
    custom_filter = "!9_bDDxJY5"
    api_key = '93KS9ghSKNGXkc7kadtKjQ(('
    while len(questions) < max_items:
        params = {
            'page': page,
            'pagesize': 100,
            'order': 'desc',
            'sort': 'activity',
            'intitle': topic,
            'site': 'stackoverflow',
            'filter': custom_filter,
            'key': api_key
        }
        url = "https://api.stackexchange.com/2.3/search"
        response = requests.get(url, params=params)
        if response.status_code == 200:
            data = response.json().get('items', [])
            if not data:
                break
            questions.extend(data)
            page += 1
        else:
            print(f"Error: Received status code {response.status_code} for topic '{topic}'")
            print(f"Response content: {response.text}")
            break
    return questions[:max_items]
```

Figure 4. Code Snippet of gathering data from API

The data gathering process for the enhanced search system on Stack Overflow is conducted through a meticulous approach utilizing the Stack Overflow API. This API provides a robust platform for retrieving detailed question data directly from Stack Overflow, which includes titles, content, answer counts, comment counts, and other engagement metrics relevant to the search queries.

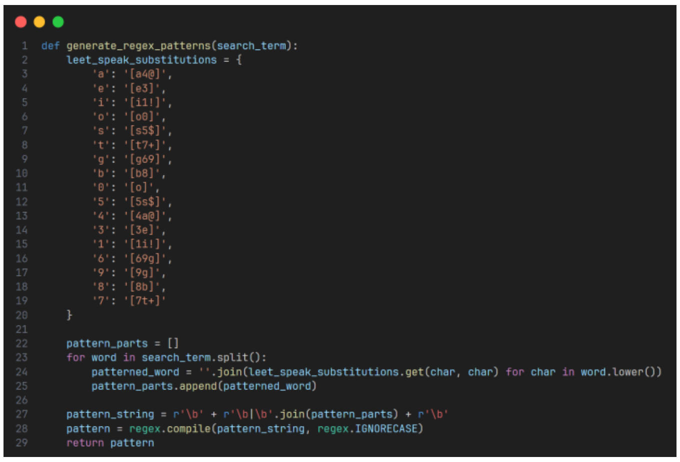
To fetch the data, a Python function `fetch_questions` is employed, which makes HTTP GET requests to the Stack Overflow API's search endpoint. This function accepts two parameters: `topic`, which specifies the subject or keywords to search for, and `max_items`, determining the maximum number of questions to retrieve. The function utilizes a loop to handle pagination, ensuring that if more than 100 items are needed (the maximum number of items retrievable in a single API call due to pagination limits), additional requests are made until the desired number of questions is gathered or no more relevant questions are available.

The API requests are strategically customized with a variety of parameters to optimize and refine the search results effectively. The `page` parameter is utilized to control the pagination of the results, ensuring that all relevant data can be accessed sequentially. The `pagesize` parameter specifies how many results appear per page, with a limit of up to 100 items, maximizing the data retrieved per request. Sorting and ordering are managed by the `order` and `sort` parameters, which prioritize the questions fetched by their relevance and recent activity, thus ensuring that the most currently pertinent questions are retrieved first. The `intitle` parameter narrows the search to include only those questions that contain the specified keywords within their titles, enhancing the relevance of the search results to the user's query. The `site` parameter specifically targets the Stack Overflow site, maintaining the focus of the data retrieval. A custom `filter` is used to obtain detailed question attributes that are crucial for subsequent processing and evaluation, tailoring the fetched data to the needs of the application. Additionally, an API key (`key`) is provided to enhance the request limit and expand access capabilities within the Stack Overflow API, allowing for more robust and extensive data interaction.

Responses from the API are parsed as JSON, and the relevant data is extracted and added to a list of questions. The function handles errors and status codes by printing appropriate messages, which assists in troubleshooting and maintaining the robustness of the data gathering process.

This method of data gathering via the Stack Overflow API not only ensures access to real-time and comprehensive data but also aligns with efficient programming practices by handling pagination and maximizing data retrieval through optimized API calls. This setup is crucial for supporting the advanced search functionalities of the system, where having access to a wide range of up-to-date questions and their metadata significantly enhances the relevance and accuracy of the search results.

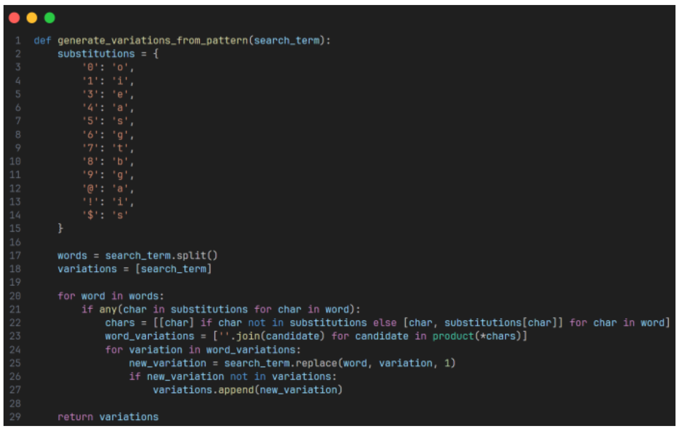
A. *Regular Expressions (Regex)*



```python
def generate_regex_patterns(search_term):
    leet_speak_substitutions = {
        'a': '[a4@]',
        'e': '[e3]',
        'i': '[i1!]',
        'o': '[o0]',
        's': '[s5$]',
        't': '[t7+]',
        'g': '[g69]',
        'b': '[b8]',
        '0': '[o]',
        '5': '[5s$]',
        '4': '[4a@]',
        '3': '[3e]',
        '1': '[1i!]',
        '6': '[69g]',
        '9': '[9g]',
        '8': '[8b]',
        '7': '[7t+]'
    }

    pattern_parts = []
    for word in search_term.split():
        patterned_word = ''.join(leet_speak_substitutions.get(char, char) for char in word.lower())
        pattern_parts.append(patterned_word)

    pattern_string = r'\b' + r'\b|\b'.join(pattern_parts) + r'\b'
    pattern = regex.compile(pattern_string, regex.IGNORECASE)
    return pattern
```

Figure 5. Code Snippet of regex function

The generate_regex_patterns function is designed to construct a regex pattern capable of recognizing a wide range of variations for each word in a given search term. This is achieved through a dictionary, leet_speak_substitutions, which maps standard alphabetic characters to regex patterns that include their common leetspeak or symbolic representations. For instance, the letter 'a' might be represented as '4' or '@' in leetspeak, thus the mapping 'a' to [a4@]. This allows the regex engine to match any of these characters, making the search robust against variations in user input. The function processes each word in the search term separately, replacing each character with its corresponding pattern from the dictionary. These patterns are then concatenated into a single regex pattern using word boundaries (\b) to ensure matches occur at the boundaries of words. The resulting pattern is compiled with case-insensitivity to broaden the search's scope, making it more likely to match relevant results irrespective of the case used in the query.



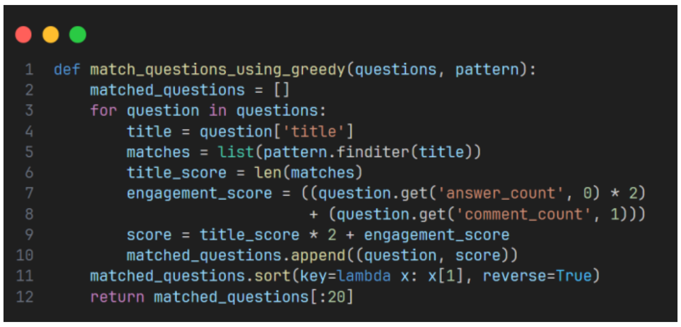
```python
def generate_variations_from_pattern(search_term):
    substitutions = {
        '0': 'o',
        '1': 'i',
        '3': 'e',
        '4': 'a',
        '5': 's',
        '6': 'g',
        '7': 't',
        '8': 'b',
        '9': 'g',
        '@': 'a',
        '!': 'i',
        '$': 's'
    }

    words = search_term.split()
    variations = [search_term]

    for word in words:
        if any(char in substitutions for char in word):
            chars = [[char] if char not in substitutions else [char, substitutions[char]] for char in word]
            word_variations = [''.join(candidate) for candidate in product(*chars)]
            for variation in word_variations:
                new_variation = search_term.replace(word, variation, 1)
                if new_variation not in variations:
                    variations.append(new_variation)

    return variations
```

Figure 6. Code Snippet of generating variations from input

Conversely, the generate_variations_from_pattern function creates a comprehensive list of potential variations of the search term, considering common substitutions and errors. This is handled through a substitutions dictionary, which, like the regex function, maps characters to possible substitutions but is utilized here for generating direct string variations. The function examines each word in the search term, identifying characters with potential substitutions. For words containing substitutable characters, it uses the product function from the itertools module to generate all possible combinations of the original and substituted characters. These combinations are then joined to form new word variations, which are added to the list of variations if not already included. This approach ensures the generation of a broad set of query variations, enhancing the search system's ability to accommodate and correct user errors or unconventional spelling.

These functions, when combined, provide a robust mechanism for handling and refining search queries. The regex patterns generated by generate_regex_patterns are utilized to scan and match content within Stack Overflow, allowing for the retrieval of relevant information that matches various user-input patterns. Meanwhile, the variations produced by generate_variations_from_pattern are used to formulate multiple query strings, which can be sent to the Stack Overflow API. This method ensures that the search covers a wide array of potential user intents, increasing the effectiveness of the search system in providing accurate and useful results.

B. *Greedy Algorithms*



```python
def match_questions_using_greedy(questions, pattern):
    matched_questions = []
    for question in questions:
        title = question['title']
        matches = list(pattern.finditer(title))
        title_score = len(matches)
        engagement_score = ((question.get('answer_count', 0) * 2)
                            + (question.get('comment_count', 1)))
        score = title_score * 2 + engagement_score
        matched_questions.append((question, score))
    matched_questions.sort(key=lambda x: x[1], reverse=True)
    return matched_questions[:20]
```

Figure 7. Code Snippet of greedy function

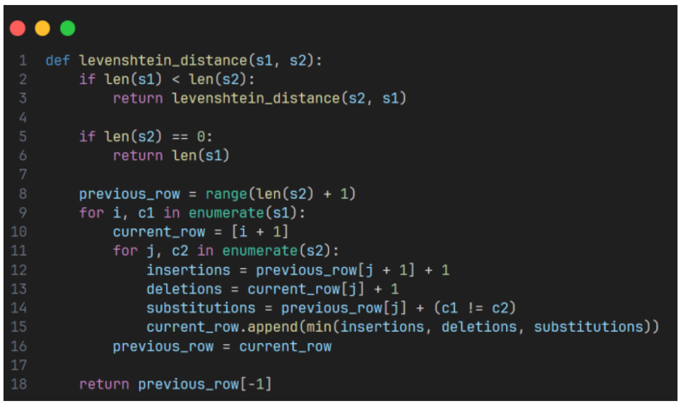
The match_questions_using_greedy function plays a pivotal role in optimizing the search process by efficiently ranking questions based on their relevance to the user's query and their engagement metrics on Stack Overflow. This function embodies the principles of a greedy algorithm, which strives to find a locally optimal solution at each step with the hope that these local optima will lead to a global optimum.

The function begins by iterating over a list of questions that have been preliminarily fetched based on the user's search term. For each question, it utilizes a compiled regex pattern to find and count matches of the search term within the question's title. This pattern matching step is crucial as it assesses the relevance of each question to the search term based on the presence and frequency of matching terms in the title.

After identifying matches, the function calculates a score for each question by considering both the number of matches (title_score) and the question's engagement metrics, such as the number of answers and comments it has received. The title_score is given a higher weighting by doubling its value, emphasizing the importance of relevance over engagement. However, the engagement score, calculated by doubling the answer count and adding the comment count (with a base value of one to ensure a minimum score for engagement), also plays a significant role. This composite score helps in prioritizing questions that are not only relevant but also deemed valuable by the community.

Once all questions have been evaluated and scored, the function sorts them in descending order based on their computed scores. This sorting step is where the greedy nature of the algorithm is most evident, as it selects the top questions based on the highest scores without re-evaluating or backtracking. The function then returns the top 20 questions, making the assumption that these represent the best match to the user's query based on the combined criteria of relevance and community engagement.

C. *Levenshtein Distance Algorithm*



```python
def levenshtein_distance(s1, s2):
    if len(s1) < len(s2):
        return levenshtein_distance(s2, s1)

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

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

    return previous_row[-1]
```

Figure 8. Code Snippet of Levenshtein Distance function

The levenshtein_distance function implements the Levenshtein Distance algorithm, which is a well-known method for measuring the difference between two strings. This algorithm calculates the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into another, providing a quantitative basis for comparing string similarity. This function is particularly useful in applications such as spell checking, error correction in text inputs, and anywhere else where measuring how similar two strings are is useful.

The implementation starts by ensuring that the first string (s1) is not shorter than the second string (s2), as this standardizes the calculation process. If s1 is shorter, the function calls itself with the order of the parameters switched. This adjustment guarantees that the algorithm always processes the shorter string in terms of the number of columns in the dynamic programming table, optimizing space complexity.

The core of the Levenshtein Distance algorithm lies in its use of dynamic programming. This method involves building up a solution using previously solved subproblems. The function initializes a list (previous_row) with a sequence from 0 to the length of s2, representing the initial cost of deleting characters from s2 to match an empty s1.

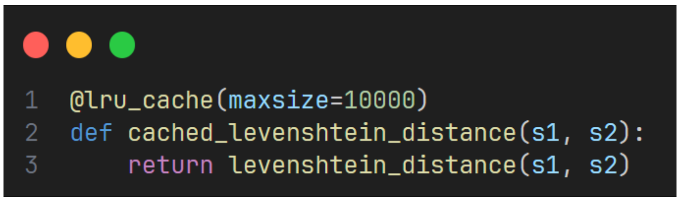
For each character in s1, the function iterates over the characters in s2, computing the costs of insertions, deletions, and substitutions. Specifically:

- Insertions are calculated by adding one to the previous row's cost at the same index, reflecting the cost of inserting a character from s2 into s1.
- Deletions are accounted for by adding one to the current row's previous cost, representing the removal of a character from s1 to match s2.
- Substitutions are calculated based on whether the current characters in s1 and s2 match; if they don't, the cost is the previous diagonal cost plus one; otherwise, it is just the previous diagonal cost.

The minimum of these three values is then determined for each character pair, and this minimum cost is added to the current_row, which tracks the computation for the current character of s1.

After processing all characters of both strings, the last value in previous_row, which now represents the full cost of transforming s1 into s2, is returned. This value is the Levenshtein Distance, indicating the minimum edit distance between the two strings.

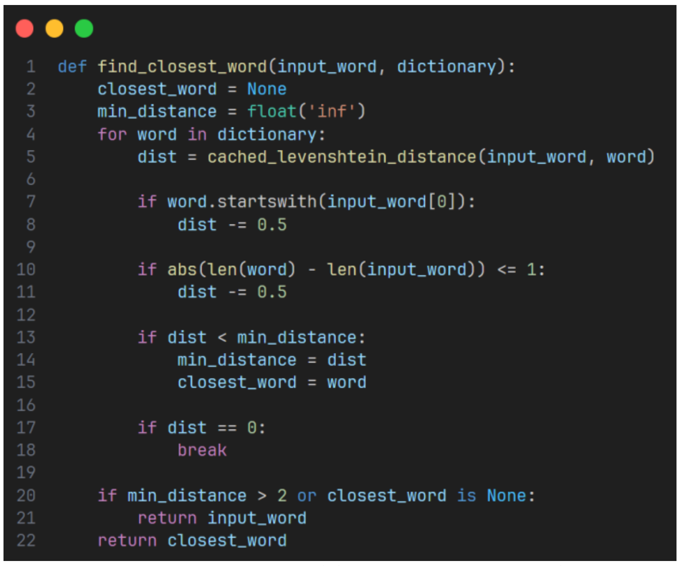
D. Caching



```python
@lru_cache(maxsize=10000)
def cached_levenshtein_distance(s1, s2):
    return levenshtein_distance(s1, s2)
```

Figure 9. Code Snippet of caching implementation

The cached_levenshtein_distance function leverages Python’s functools.lru_cache decorator, a powerful feature that implements memoization to optimize the performance of function calls that perform computationally intensive operations. By caching the results of previous invocations of the levenshtein_distance function, it avoids redundant calculations by storing and retrieving the results of function calls based on the function's input arguments. The cache can hold up to 10,000 unique function calls, which significantly reduces the processing time, especially when the function is used frequently in contexts like checking a large number of words against a user's input. This caching strategy is particularly effective when dealing with a finite set of potential matches, such as a dictionary of words, where many words might be repeatedly compared against different input terms during the course of a user session.



```python
def find_closest_word(input_word, dictionary):
    closest_word = None
    min_distance = float('inf')
    for word in dictionary:
        dist = cached_levenshtein_distance(input_word, word)

        if word.startswith(input_word[0]):
            dist -= 0.5

        if abs(len(word) - len(input_word)) <= 1:
            dist -= 0.5

        if dist < min_distance:
            min_distance = dist
            closest_word = word

        if dist == 0:
            break

    if min_distance > 2 or closest_word is None:
        return input_word
    return closest_word
```

Figure 10. Code Snippet of finding closest word function

The find_closest_word function is designed to identify the word in a given dictionary that most closely matches an input word, based on the Levenshtein Distance. This function iterates over each word in the dictionary, computing the distance to the input word using the cached_levenshtein_distance to ensure efficiency.

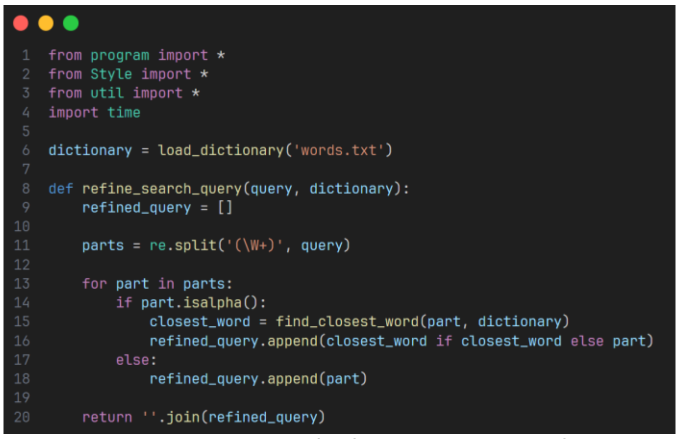
The function incorporates a few intelligent tweaks to refine its search for the closest match:

- **Initial Letter Bias**: The function provides a small heuristic bonus (reducing the distance by 0.5) if the word in the dictionary starts with the same letter as the input word. This heuristic is based on the observation that words that start with the same letter are more likely to be related in meaning or more likely to be the intended word if there's a typographical error in the input.
- **Length Consideration**: Another heuristic adjustment reduces the distance by 0.5 if the word in the dictionary is within one character in length of the input word. This adjustment is useful for catching common typos like

missing letters, extra letters, or substituted letters, which might not significantly alter the length of the word.

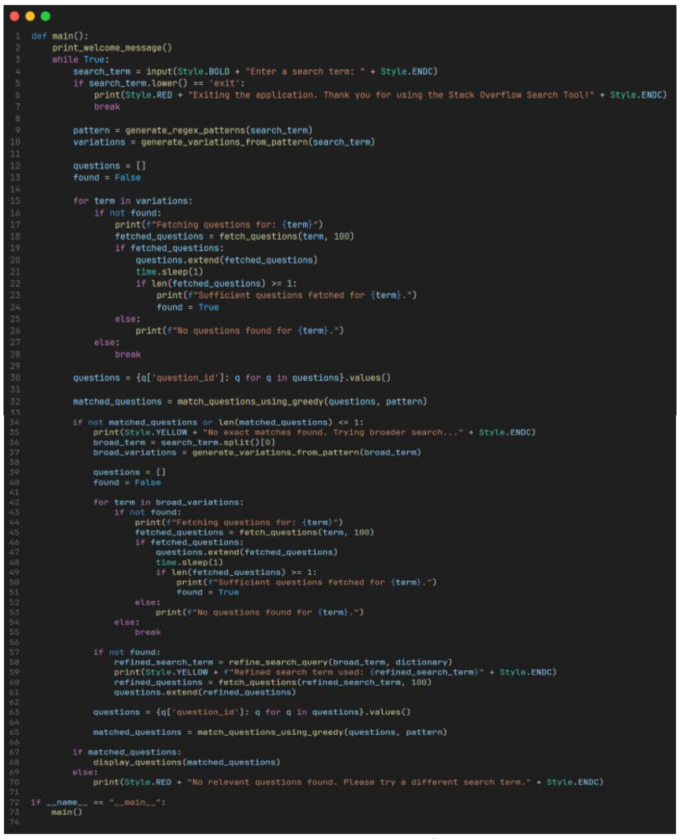
The function keeps track of the word with the minimum distance found during the iteration. If this minimum distance is greater than a threshold (in this case, 2), or if no word has been identified as a close match (closest_word is None), the function defaults to returning the original input word, suggesting that no sufficiently close correction was found within the dictionary. Otherwise, it returns the word that was identified as having the closest Levenshtein Distance to the input.

E. Code flow



```python
from program import *
from Style import *
from util import *
import time

dictionary = load_dictionary('words.txt')

def refine_search_query(query, dictionary):
    refined_query = []

    parts = re.split('(\W+)', query)

    for part in parts:
        if part.isalpha():
            closest_word = find_closest_word(part, dictionary)
            refined_query.append(closest_word if closest_word else part)
        else:
            refined_query.append(part)

    return ''.join(refined_query)
```

Figure 11. Code Snippet of refining search query function



```python
def main():
    print_welcome_message()
    while True:
        search_term = input(Style.BOLD + "Enter a search term: " + Style.ENDC)
        if search_term.lower() == 'exit':
            print(Style.RED + "Exiting the application. Thank you for using the Stack Overflow Search Tool!" + Style.ENDC)
            break

        pattern = generate_regex_patterns(search_term)
        variations = generate_variations_from_pattern(search_term)

        questions = []
        found = False

        for term in variations:
            if not found:
                print(f"Fetching questions for: {term}")
                fetched_questions = fetch_questions(term, 100)
                if fetched_questions:
                    questions.extend(fetched_questions)
                    time.sleep(1)
                    if len(fetched_questions) >= 1:
                        print(f"Sufficient questions fetched for {term}.")
                        found = True
                else:
                    print(f"No questions found for {term}.")
            else:
                break

        questions = {q['question_id']: q for q in questions}.values()

        matched_questions = match_questions_using_greedy(questions, pattern)

        if not matched_questions or len(matched_questions) <= 1:
            print(Style.YELLOW + "No exact matches found. Trying broader search..." + Style.ENDC)
            broad_term = search_term.split()[0]
            broad_variations = generate_variations_from_pattern(broad_term)

            questions = []
            found = False

            for term in broad_variations:
                if not found:
                    print(f"Fetching questions for: {term}")
                    fetched_questions = fetch_questions(term, 100)
                    if fetched_questions:
                        questions.extend(fetched_questions)
                        time.sleep(1)
                        if len(fetched_questions) >= 1:
                            print(f"Sufficient questions fetched for {term}.")
                            found = True
                    else:
                        print(f"No questions found for {term}.")
                else:
                    break

            if not found:
                refined_search_term = refine_search_query(broad_term, dictionary)
                print(Style.YELLOW + f"Refined search term used: {refined_search_term}" + Style.ENDC)
                refined_questions = fetch_questions(refined_search_term, 100)
                questions.extend(refined_questions)

            questions = {q['question_id']: q for q in questions}.values()

            matched_questions = match_questions_using_greedy(questions, pattern)

        if matched_questions:
            display_questions(matched_questions)
        else:
            print(Style.RED + "No relevant questions found. Please try a different search term." + Style.ENDC)

if __name__ == "__main__":
    main()
```

Figure 12 & 13. Code Snippet of main program

The author now will explain how the program is working. Given the search term is “hyperparameter tuning makes accuracy lower”. When the search term is inputted, the program first invokes the generate_regex_patterns function. This function creates a regular expression pattern that can recognize variations in the input term. It does so by mapping each alphabetic character in the search term to a regex pattern that includes possible substitutions based on common typos or leetspeak variations. For example, "hyperparameter" could be distorted to "hyp3rparamet3r", but the regex pattern will still match it because of the substitutions defined (e.g., 'e' to '[e3]').

Next, the generate_variations_from_pattern function produces a list of possible variations of the search term to account for different ways users might misspell or vary their queries. This list helps in broadening the search to include queries that might not exactly match the original input but are close variations of it.

The program then attempts to fetch questions from the Stack Overflow API using these variations. It starts with the exact input term and searches for questions that match this term. If it finds questions, it proceeds to process them; if not, it iterates through other generated variations, sending requests to the API until it either finds questions or exhausts the variations.

In this specific instance, no questions were found matching the complete search phrase "hyperparameter tuning makes accuracy lower". This triggers a broader search mechanism where the program isolates the first significant term "hyperparameter" and regenerates regex patterns and variations for this narrower term. It then repeats the fetch process. If broadening the search term still does not yield the fetched question, the program will check for similar words using the Levenshtein Distance algorithm to find the closest match between the input and the words in the dictionary. If refining the search query still does not retrieve the question, the program will stop and end with no solutions retrieved.

Once questions are fetched for the term "hyperparameter", the match_questions_using_greedy function sorts these questions based on a scoring system that prioritizes the relevance (number of regex matches in the title) and engagement metrics (answers and comments). This greedy algorithm ensures that the questions most likely to be useful are selected and ranked at the top of the results list.

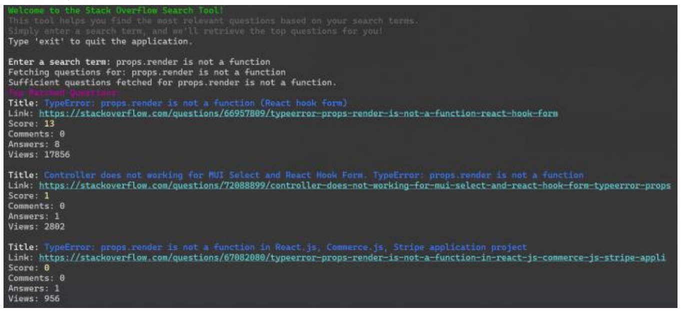
Finally, the selected questions are displayed, with details such as title, link, score, number of comments, answers, and views. This list provides users with direct access to discussions and solutions relevant to their query about "hyperparameter tuning," even though the original, more complex query did not directly yield results.

IV. TESTING AND ANALYSIS

For the enhanced search tool developed for Stack Overflow, we have conducted detailed testing using multiple test cases that simulate real-world usage. These test cases help illustrate how the system handles different types of user queries, ranging from straightforward to complex and error-prone inputs.

A. *Testing*

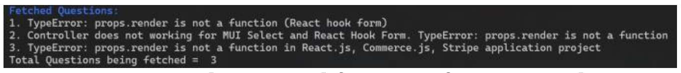
1. Test Case 1



```
Welcome to the Stack Overflow Search Tool!
This tool helps you find the most relevant questions based on your search terms.
Simply enter a search term, and we'll retrieve the top questions for you!
Type 'exit' to quit the application.

Enter a search term: props.render is not a function
Fetching questions for: props.render is not a function
Sufficient questions fetched for props.render is not a function.
Top Matched Questions:
Title: TypeError: props.render is not a function (React hook form)
Link: https://stackoverflow.com/questions/66957809/typeerror-props-render-is-not-a-function-react-hook-form
Score: 13
Comments: 0
Answers: 8
Views: 17856

Title: Controller does not working for MUI Select and React Hook Form. TypeError: props.render is not a function
Link: https://stackoverflow.com/questions/72088899/controller-does-not-working-for-mui-select-and-react-hook-form-typeerror-props
Score: 1
Comments: 0
Answers: 1
Views: 2802

Title: TypeError: props.render is not a function in React.js, Commerce.js, Stripe application project
Link: https://stackoverflow.com/questions/67082080/typeerror-props-render-is-not-a-function-in-react-js-commerce-js-stripe-appli
Score: 0
Comments: 0
Answers: 1
Views: 956
```

Figure 14. Case Study 1

In the first case study, the user input precisely matched a question posted on Stack Overflow. This accuracy in the input enabled the system to quickly locate and fetch only three articles that were exactly relevant to the search query.

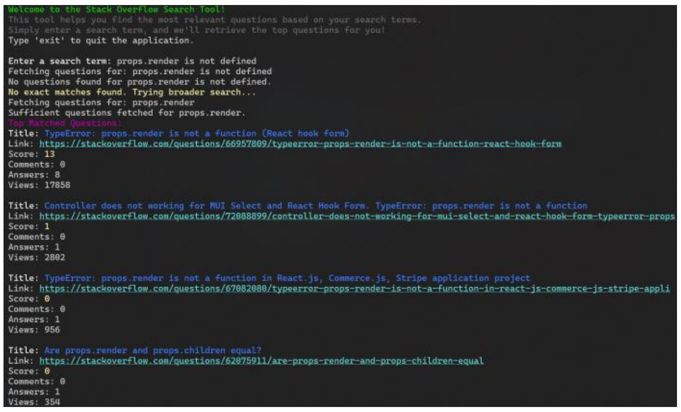


```
Fetched Questions:
1. TypeError: props.render is not a function (React hook form)
2. Controller does not working for MUI Select and React Hook Form. TypeError: props.render is not a function
3. TypeError: props.render is not a function in React.js, Commerce.js, Stripe application project
Total Questions being fetched =  3
```

Figure 15. Title retrieved from API for Case Study 1

Once these articles were retrieved from the API, the next step involved applying a regular expression (regex) to match the user's input against the titles of these articles. This ensured that only the most relevant articles were considered further. Subsequently, a greedy algorithm was implemented to rank these articles. The greedy approach prioritized articles based on a calculated score, which assessed both the number of matches in the regex process and the engagement metrics (like views and answers) of each article. The article with the highest composite score was deemed the most relevant. This efficient filtering and ranking process highlight the system's capability to handle precise queries with high accuracy, ensuring that the user is provided with the most directly applicable and useful content.

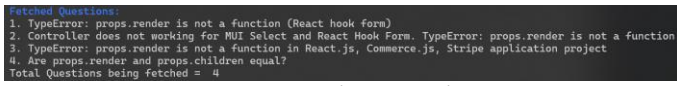
2. Test Case 2



```
Welcome to the Stack Overflow Search Tool!
This tool helps you find the most relevant questions based on your search terms.
Simply enter a search term, and we'll retrieve the top questions for you!
Type 'exit' to quit the application.

Enter a search term: props.render is not defined
Fetching questions for: props.render is not defined
No questions found for props.render is not defined.
No exact matches found. Trying broader search...
Fetching questions for: props.render
Sufficient questions fetched for props.render.
Top Matched Questions:
Title: TypeError: props.render is not a function (React hook form)
Link: https://stackoverflow.com/questions/66957809/typeerror-props-render-is-not-a-function-react-hook-form
Score: 13
Comments: 0
Answers: 8
Views: 17858

Title: Controller does not working for MUI Select and React Hook Form. TypeError: props.render is not a function
Link: https://stackoverflow.com/questions/72088899/controller-does-not-working-for-mui-select-and-react-hook-form-typeerror-props
Score: 1
Comments: 0
Answers: 1
Views: 2802

Title: TypeError: props.render is not a function in React.js, Commerce.js, Stripe application project
Link: https://stackoverflow.com/questions/67082080/typeerror-props-render-is-not-a-function-in-react-js-commerce-js-stripe-appli
Score: 0
Comments: 0
Answers: 1
Views: 956

Title: Are props.render and props.children equal?
Link: https://stackoverflow.com/questions/62075911/are-props-render-and-props-children-equal
Score: 0
Comments: 0
Answers: 1
Views: 354
```

Figure 16. Case Study 2

The second case study addressed a scenario where the user input did not exactly match any existing keywords or phrases found in Stack Overflow articles, leading initially to zero results. To overcome this, the system performed a secondary search focusing solely on the first word of the user's input, "props.render."

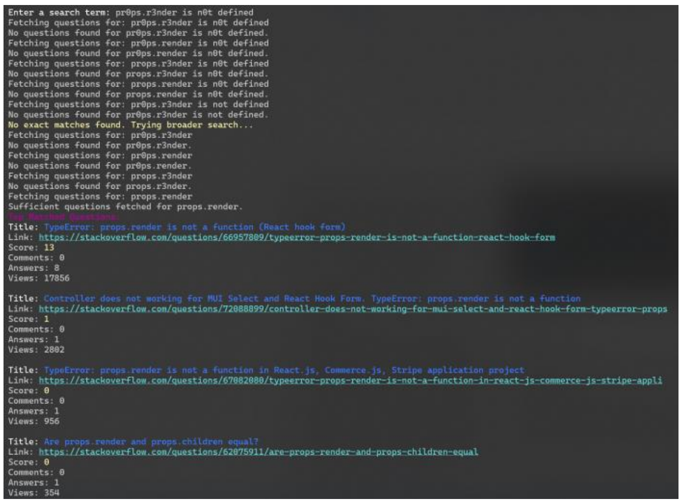


```
Fetched Questions:
1. TypeError: props.render is not a function (React hook form)
2. Controller does not working for MUI Select and React Hook Form. TypeError: props.render is not a function
3. TypeError: props.render is not a function in React.js, Commerce.js, Stripe application project
4. Are props.render and props.children equal?
Total Questions being fetched =  4
```

Figure 17. Title retrieved from API for Case Study 2

This strategy significantly broadened the search scope, fetching a larger array of articles. Once these articles were retrieved, the same regex matching was applied to ensure the articles had relevance to the initial query. The articles were then ranked by relevance using the greedy algorithm, which assessed the articles based on the alignment of their content with the user's intended query and the engagement they had received. This case demonstrates the system's ability to adaptively widen the search parameters when faced with initial failures, ensuring that the user still receives useful results by leveraging broader keywords.

3. Test Case 3

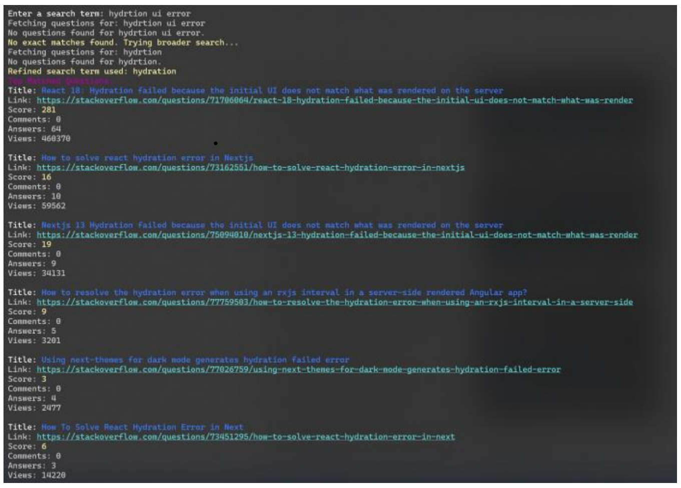


```
Enter a search term: pr0ps.r3nder is n0t defined
Fetching questions for: pr0ps.r3nder is n0t defined
No questions found for pr0ps.r3nder is n0t defined.
Fetching questions for: pr0ps.render is n0t defined
No questions found for pr0ps.render is n0t defined.
Fetching questions for: props.r3nder is n0t defined
No questions found for props.r3nder is n0t defined.
Fetching questions for: props.render is n0t defined
No questions found for props.render is n0t defined.
Fetching questions for: pr0ps.r3nder is not defined
No questions found for pr0ps.r3nder is not defined.
No exact matches found. Trying broader search...
Fetching questions for: pr0ps.r3nder
No questions found for pr0ps.r3nder.
Fetching questions for: pr0ps.render
No questions found for pr0ps.render.
Fetching questions for: props.r3nder
No questions found for props.r3nder.
Fetching questions for: props.render
Sufficient questions fetched for props.render.
Top Matched Questions:
Title: TypeError: props.render is not a function (React hook form)
Link: https://stackoverflow.com/questions/66957809/typeerror-props-render-is-not-a-function-react-hook-form
Score: 13
Comments: 0
Answers: 8
Views: 17856

Title: Controller does not working for MUI Select and React Hook Form. TypeError: props.render is not a function
Link: https://stackoverflow.com/questions/72088899/controller-does-not-working-for-mui-select-and-react-hook-form-typeerror-props
Score: 1
Comments: 0
Answers: 1
Views: 2802

Title: TypeError: props.render is not a function in React.js, Commerce.js, Stripe application project
Link: https://stackoverflow.com/questions/67082080/typeerror-props-render-is-not-a-function-in-react-js-commerce-js-stripe-appli
Score: 0
Comments: 0
Answers: 1
Views: 956

Title: Are props.render and props.children equal?
Link: https://stackoverflow.com/questions/62075911/are-props-render-and-props-children-equal
Score: 0
Comments: 0
Answers: 1
Views: 354
```

Figure 18. Case Study 3

In the third scenario, the user entered a query with common typographical errors where letters were substituted with numerals, such as "pr0ps" instead of "props." The system's robust design addressed this by first attempting to fetch a range of variations on the user's input, correcting "pr0ps" to "props" and "n0t" to "not." This process involved multiple fetch attempts to find a match. If these attempts were unsuccessful, the system defaulted to focusing on the initial segment of the query, fetching variations on "pr0ps.r3nder" to maximize the chances of retrieving relevant data. This approach underscores the system's flexibility and its sophisticated handling of user errors, utilizing intelligent fallback strategies to salvage useful results from potentially unsuccessful queries.

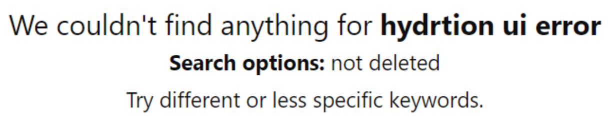
4. Test Case 4



```
Enter a search term: hydrtion ui error
Fetching questions for: hydrtion ui error
No questions found for hydrtion ui error.
No exact matches found. Trying broader search...
Fetching questions for: hydrtion
No questions found for hydrtion.
Refined search term used: hydration
Top Matched Questions:
Title: React 18: Hydration failed because the initial UI does not match what was rendered on the server
Link: https://stackoverflow.com/questions/71706064/react-18-hydration-failed-because-the-initial-ui-does-not-match-what-was-render
Score: 281
Comments: 0
Answers: 64
Views: 460370

Title: How to solve react hydration error in Nextjs
Link: https://stackoverflow.com/questions/73162551/how-to-solve-react-hydration-error-in-nextjs
Score: 16
Comments: 0
Answers: 10
Views: 59562

Title: Nextjs 13 Hydration failed because the initial UI does not match what was rendered on the server
Link: https://stackoverflow.com/questions/75094010/nextjs-13-hydration-failed-because-the-initial-ui-does-not-match-what-was-render
Score: 19
Comments: 0
Answers: 9
Views: 34131

Title: How to resolve the hydration error when using an rxjs interval in a server-side rendered Angular app?
Link: https://stackoverflow.com/questions/77759503/how-to-resolve-the-hydration-error-when-using-an-rxjs-interval-in-a-server-side
Score: 9
Comments: 0
Answers: 5
Views: 3201

Title: Using next-themes for dark mode generates hydration failed error
Link: https://stackoverflow.com/questions/77026759/using-next-themes-for-dark-mode-generates-hydration-failed-error
Score: 3
Comments: 0
Answers: 4
Views: 2477

Title: How To Solve React Hydration Error in Next
Link: https://stackoverflow.com/questions/73451295/how-to-solve-react-hydration-error-in-next
Score: 6
Comments: 0
Answers: 3
Views: 14220
```

Figure 19. Case Study 4

The final case study revolved around a user input that included a typo, "hydrtion" instead of "hydration." This typographical error could potentially derail the search process, but the system's implementation of the Levenshtein Distance (LD) algorithm preemptively countered this issue. The LD algorithm was employed after an initial broader search failed to yield results, due to the dictionary not recognizing the misspelled word. By comparing the misspelled word to similar terms in an English dictionary, the system identified the correct spelling and re-executed the search with the corrected term. This not only salvaged the query but also ensured that the results were relevant and accurate, demonstrating the system’s advanced capability in handling and correcting user input errors through intelligent algorithmic interventions.



We couldn't find anything for **hydrtion ui error**

Search options: not deleted

Try different or less specific keywords.

Figure 20. Example searching not found

Source : https://stackoverflow.com/search?q=hydrtion+ui+error

B. *Analysis*

The analysis of the four case studies conducted reveals a robust and adaptive search system capable of handling a wide array of query scenarios on Stack Overflow, from precise to imprecise inputs. The first case study showcases the system's efficiency in swiftly retrieving relevant results when user inputs closely match database entries, reflecting its effectiveness in straightforward scenarios. Conversely, the subsequent case studies demonstrate the system's adeptness in managing more complex interactions, such as vague queries or inputs marred by typographical errors.

In particular, the system's use of regex for generating variations and the Levenshtein Distance algorithm for correcting errors highlights its sophisticated approach to ensuring user queries—no matter how inaccurately inputted—result in useful outcomes. This capability is crucial for enhancing user satisfaction, as it minimizes potential frustrations associated with unsuccessful searches. Moreover, the greedy algorithm's role in prioritizing search results based on relevance and engagement metrics ensures that users are presented with the most useful and high-quality content.

However, the necessity of broadening searches and the iterative fetching process indicate potential areas for refinement, such as improving initial keyword extraction to reduce reliance on broader searches. Integrating more advanced natural language processing could enhance the system's understanding of query context, further improving accuracy and efficiency.

Overall, the system proves highly effective in adapting to user needs and maintaining a high level of performance across diverse querying conditions, underscoring its utility in a dynamic search environment like Stack Overflow. This analysis not only affirms the system's current capabilities but also sets the stage for future enhancements that could leverage emerging technologies for even better performance.

V. CONCLUSION

In conclusion, the development and implementation of the enhanced search system for Stack Overflow have successfully demonstrated the integration of complex algorithms such as regular expressions, greedy ranking mechanisms, and the Levenshtein Distance algorithm significantly improve the handling of user queries. This system is designed to be robust and adaptive, capable of managing a wide range of input scenarios from exact matches to typographical errors and vague queries.

Throughout the testing and analysis phases, the system consistently showcased its ability to efficiently process and respond to user inputs, delivering relevant and high-quality results even under challenging conditions. The implementation of regex has proven effective for pattern recognition and variation handling, allowing the system to capture a broad spectrum of potential user inputs. Meanwhile, the greedy algorithm ensures that the most pertinent responses are prioritized based on relevance and user engagement, enhancing the overall utility of the search results.

Moreover, the integration of the Levenshtein Distance algorithm has added a sophisticated layer of error correction, which is crucial for accommodating common user mistakes and ensuring that these do not hinder the search experience. This feature not only improves the accuracy of the search results but also enhances user satisfaction by reducing the frequency and impact of unsuccessful searches.

Future enhancements could focus on incorporating advanced natural language processing techniques to further refine the understanding of user queries, potentially reducing the need for broad searches and improving the system’s efficiency. Additionally, optimizing the caching mechanism to prioritize frequently encountered queries could enhance response times and reduce computational overhead.

This project has not only improved the search functionality on Stack Overflow but has also set a benchmark for how search systems can evolve to meet the demands of real-world users in dynamic and information-rich environments. By continuing to adapt and integrate cutting-edge technologies, such systems can provide even greater support to users, helping them navigate vast databases with ease and efficiency.

VI. APPENDIX

The complete program of this paper can be found below
https://github.com/Filbert88/StackOverflow-Search-Enhancer

VIDEO LINK AT YOUTUBE

https://youtu.be/4itEYRlNb1A?si=-5c1eOOrP8k5T5pT

ACKNOWLEDGMENT

The author would like to express deep gratitude to the following individuals for their invaluable support:

1. The Almighty God - The author acknowledges the divine blessings and guidance that have been crucial in completing this paper. The author's faith and reliance on divine support have been fundamental throughout the writing process.

2. The Author's Parents - The constant support, encouragement, and belief in the author's capabilities from their parents have been indispensable. Their unwavering presence and motivation have been a significant driving force behind the completion of this work.

3. Dr. Ir. Rinaldi Munir, M.T., Dr. Ir. Rila Mandala, and Dr. Nur Ulfa Maulidevi - The author extends a heartfelt thank you to these esteemed professors from the IF2211 Algorithm Strategies course. Their exceptional mentorship, insightful guidance, and profound knowledge have greatly enhanced the author's understanding of the subject. Their dedication and expertise have had a profound impact.

The author is deeply grateful for the immeasurable contributions of these individuals, whose support and guidance have been essential to the successful completion of this paper.

REFERENCES

[1] Câmpeanu, C., Salomaa, K., & Yu, S. (2003). A FORMAL STUDY OF PRACTICAL REGULAR EXPRESSIONS. International Journal of Foundations of Computer Science, 14(06), 1007–1018. https://doi.org/10.1142/s012905410300214x

[2] Baeldung. (n.d.). Levenshtein Distance Computation. Retrieved from https://www.baeldung.com/cs/levenshtein-distance-computation

[3] Munir, R. (2021). *Algoritma Greedy (Bagian 1)*. Retrieved from https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf

[4] Munir, R. (2019). *String Matching dengan Regex*. Retrieved from https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2022-2023/String-Matching-dengan-Regex-2019.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

[illegible]

Filbert (13522021)

.