Approximate and Exact String Matching Implementation in a Mandarin Dictionary Retrieval System

Ray Owen Martin - 13524033
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail:rayowenmartin2018@gmail.com , 13524033@std.stei.itb.ac.id

***Abstract*—In this modern era, Internet-based platforms have been essential to learn new skills, including learning new languages. However, learning a new language which does not use the Latin alphabet, such as Mandarin with its Hanzi as the characters and Pinyin as their pronunciation, presents their own level of challenge. Relying on external applications or physical dictionaries disrupts user experience. One solution is to develop a browser extension that acts as a pocket dictionary and can tolerate typographical errors in phonetic inputs. Brute Force, Knuth-Morris-Pratt (KMP), and Weighted Levenshtein algorithms are utilized to explore through the dictionary with all Hanzi, Pinyin and definition as possible queries. This paper will present how these algorithms can be effectively implemented as a solution and optimally routed for each challenge.**

***Keywords*—KMP, Weighted Levenshtein, Mandarin, Chromium Extension, String Matching**

I. Introduction

The Internet has proven to be an immensely useful tool in today's modern era, acting as a primary source of information. People utilize the Internet for various aspects of life, including work, education, and entertainment. One factor that makes the Internet so dominant is how individuals heavily rely on it to acquire knowledge and learn important skills.

Skills are a primary necessity for securing employment, or starting a business. Both hard skills and soft skills are sought after in the job market, and relying on a single particular skill is no longer sufficient in the eyes of a human resource manager. Other than the primary skills for a profession, individuals must also possess secondary skills to support them. These include time management, problem solving, critical thinking and one that proves to be one of the most challenging to learn, speaking multiple languages.

Countries dominating the global economy necessitate effective communication across different countries. Consequently, language barriers often create wedges between international parties. Even though artificial intelligence and translating apps have shown significance in resolving this problem, differences in context and culture might still note uncaught mistakes and miscommunications. Human translations and understandings remain more trustworthy than tools in this case.

Learning a new language often poses a significant problem for independent learners, emphasizing the relevance of easily accessible tools like pocket dictionaries. Integrating them into browsers would boost the efficiency of one’s learning process, yet typographical errors, partial definitions and limited keyboard or alphabet pose a challenge for it. Hence, implementing robust string matching algorithms is required to ensure a dictionary retrieval system trustworthy enough to support one’s language learning.

II. Theoretical Frameworks

A. String Matching

String or pattern matching is a process of finding a smaller string (pattern) inside a larger text [1]. The main task is to find the locations in text matching with the pattern, sometimes only the first location, sometimes all locations. Other than using brute force to try to match the characters from text and pattern one by one, there are few string matching algorithms generally used, divided into exact matching and approximate matching. Exact matching includes Knuth-Morris-Pratt (KMP) algorithm, Boyer Moore algorithm and Rabin Karp algorithm, while approximate matching includes Weighted Levenshtein and Jaro-Winkler distance. Exact matching improves mainly speed of the matching process, while approximate matching is mainly used to tolerate typographical errors.

The importance of string matching can be seen in many of its applications, including searching in text editors, plagiarism detection, spelling checker, web search engine, image processing, even bioinformatics.

B. The Brute Force Approach

String matching is simple when complexity is not a factor to be weighed in. The algorithm can be stated as: The pattern moves over the text one position at a time and characters are compared. If all characters match, the index is stored; otherwise, the next position is checked. This approach is not a problem when both pattern and text are short, but when either one of those, or maybe both, happens to be long enough, this approach will surely create a major time problem. The best case scenario is $O(n)$, just like any other algorithm, but the worst case scenario will create the time complexity of $O(m \times (n - m + 1))$ with n as the length of text and m as the length of pattern.

C. Knuth-Morris-Pratt Algorithm

One of the most widely used algorithms for string matching is Knuth-Morris-Pratt (KMP) algorithm, which aids us significantly with its linear-time processing speed. Rather than directly match characters, this algorithm initially precomputes a prefix function, also known as border function and failure function. The function finds the size of the largest prefix that is also a suffix of the same pattern. This particular function will then be used in order to avoid testing useless shifts as in the brute force attempt, which undoubtedly speeds up the search. The pseudocode for this algorithm can be seen as follows:

```
KMP-MATCHER(T, P)
1   n = T.length
2   m = P.length
3   π = COMPUTE-PREFIX-FUNCTION(P)
4   q = 0                                      // number of characters matched
5   for i = 1 to n                             // scan the text from left to right
6       while q > 0 and P[q + 1] ≠ T[i]
7           q = π[q]                           // next character does not match
8       if P[q + 1] == T[i]
9           q = q + 1                          // next character matches
10      if q == m                              // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                           // look for the next match
```

Fig 1. KMP algorithm pseudocode *(Source: [3])*

```
COMPUTE-PREFIX-FUNCTION(P)
1   m = P.length
2   let π[1 . . m] be a new array
3   π[1] = 0
4   k = 0
5   for q = 2 to m
6       while k > 0 and P[k + 1] ≠ P[q]
7           k = π[k]
8       if P[k + 1] == P[q]
9           k = k + 1
10      π[q] = k
11  return π
```

Fig 2. KMP border function pseudocode *(Source: [3])*

KMP algorithm improves worst case complexity to O(m+n), with O(m) for computing the border function and O(n) to match the string. It is important to note that this approach is for finding only the initial match found in the text.

D. Standard Levenshtein Distance

The Levenshtein Distance is a number that shows how different two strings are. The equation can be written as follows:

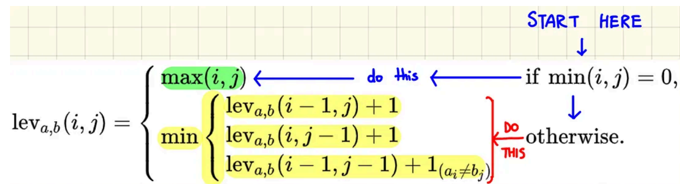

Fig 3. Levenshtein distance function *(Source: https://www.geeksforgeeks.org/dsa/naive-algorithm-for-pattern-searching/)*

with a as first string, b as second string, i as terminal character position of a and j as terminal character position as b, with all indexes starting from 1. To put in simple terms, when a certain index of both strings are different, it adds a point to the Levenshtein distance. The point stands for how many edits a string needs to be done to be the same as the other. The bigger the point is, the further it is from string a to string b.

E. Weighted Levenshtein Distance

Taking into account how the standard Levenshtein algorithm penalizes every difference fairly as 1 point, the function can be modified such that certain differences can be tolerated according to the needs. For example if two strings are compared and there is a need to tolerate a typographical error of 'k' and 'l' due to both characters being side by side on a keyboard, the penalty cost can be reduced to 0.25. This technique would be beneficial to calculate more accurate distances in certain contexts, nevertheless the implementation and punishment points are important aspects to support this benefit.

F. Dynamic Programming in Weighted Levenshtein

To computationally implement the modified weighted Levenshtein algorithm, Dynamic Programming (DP) is often used due to its versatility. The DP approach constructs a two-dimensional matrix where the algorithm iteratively populates the cells by calculating the minimum cost among three possible operations: an insertion, a deletion, or a substitution. In a Weighted Levenshtein DP implementation, instead of adding a constant integer of 1 for a substitution, the algorithm evaluates the specific characters involved and adds the customized fractional penalty cost to the matrix calculation. The final minimal edit distance is then systematically extracted from the bottom-rightmost cell of the matrix, ensuring an optimal and highly accurate evaluation of the string similarity.

G. Mandarin Language

Mandarin refers to the standardized form of spoken Chinese. It is the official language of Mainland China. It uses the Simplified Chinese character system, which uses Hanzi as the character and Pinyin as a system to write out Chinese phonetically using the Latin alphabet. For example, the Hanzi 中文 can also be written as zhōngwén or zhong1wen2. Although the first annotation is more popular and formal, the usage of tones is not popular with the Latin keyboard which results in the need of the second one. It is important to note that the same Pinyin can refer to different Hanzi and the same Hanzi can also have different Pinyin, hence a full context is often needed to understand the meaning of their usages.

Most Hanzi also have radical, for example 的 (de; a possessive pronoun) has the radical 白 (bái) also read as 白字旁 (báizìpáng). Each radical shows different meaning, in this case 白字旁 is usually used for words with meaning related to purity or light. For example, 的 had an original meaning of bright or bullseye. Understanding radicals can improve one's understanding of Hanzi for this reason.

H. Chromium Extension

Chromium is an open-source browser architecture that serves as the foundation for most modern web browsers, including Google Chrome, Brave and Microsoft Edge. The system operates utilizing the Manifest V3 architecture, which

enforces strict security protocols and highly optimized background processing. A browser extension is a small, lightweight software that can be installed into the browser to customize its appearance or function. It can do almost everything the browser permits, including encrypt emails, store passwords, check spelling and more.

Contemporary Chromium extensions operate utilizing the Manifest V3 architecture, a standardized framework designed to enforce strict security protocols and ensure highly optimized background processing. Furthermore, this architecture facilitates Document Object Model (DOM) manipulation, which allows the extension to dynamically interact with the active webpage. Through this mechanism, the extension can seamlessly read, modify, or highlight specific elements—such as textual data and characters—directly on the user's screen without permanently altering the original source code of the website.

III. Proposed Method

The proposed method to make this Mandarin Dictionary Retrieval System is to implement string matching algorithms in a TypeScript-based backend aided by HTML interface. The compile process will use Vite. The method can be divided into few parts as follows:

A. System Architecture

The project is based on Manifest V3 architecture with two primary environments, including the popup user interface and the content script. The user interface is programmed with HTML and CSS and is used to receive the queries typed in by users alongside the parameter preferred.

The whole extension execution will be done within the extension's isolated environment. Afterwards, the extension utilizes the *chrome.tabs.sendMessage* API to transmit the matched Hanzi messages to the Content Script (*content.ts*). The Content Script is persistently injected into all active web URLs in order to traverse and manipulate the Document Object Model (DOM) of the active webpage. This communication pipeline enables the real-time visual isolation and highlighting of the queried characters directly on the user's screen.

From a high level perspective, the extension can be divided into levels:

- Presentation layer
  The popup interface handled by mainly *index.html* and *main.ts* which handle query selections, results rendering and benchmark display, as well as *style.css* to manage styles using CSS
- Data layer
  The dataset used as the main dictionary is loaded with *load.ts*
- Search layer
  search.ts serves as the primary file handling all searches, whilst *pinyinsearch.ts*, *defsearch.ts* and *radicalsearch.ts* each handle their own algorithm of searching their parameters.
- Messaging layer
  messaging.ts sends messages from backend to the interface in the active tab, while *content.ts* handles receiving the messages and acts accordingly. This layer is also responsible for highlighting matching Hanzi as in *messaging.ts*.

B. Presentation Layer

This layer is responsible for managing the whole popup interface. There are few files related to the presentation layer:

- *index.html*
  This file manages the base look of the extension. This file is then controlled and modified by other files to show search results and benchmark display, hence it is designed to be highly responsive
- *style.css*
  Just like CSS in general, this file handles interface decorators, including interface element sizes, font sizes, margin, padding, etc.
- *main.ts*
  Upon initialization, the main file calls the dataset loading and searching function. To ensure a highly responsive user experience, the program is event-driven, that is implemented by doing search for each keystroke, whether the input is complete or not. For further uses, change in algorithm selection will also redo search accordingly. It is important to note that this program does not handle search algorithm routing explicitly as it is handled in the search layer.

C. Data Layer

The whole extension will be useless without a proper dataset and the dataset loading algorithm. This project uses two datasets to complement each other, which can be explored in the *public/data/* folder of the project–*hanziDB.csv* handles most single words while *zerotohero-zh-vocabulary.csv* handles phrases and some other single words. As this is a small project, the dataset may not be complete.

- *load.ts*
  This program is responsible for defining the data structure of the dataset transformed, as well as loading, parsing and processing the datasets.
  As we are interested in Hanzi, Pinyin, definition and radical, we will be saving these fields of the dataset. Note that there is no radical field in *zerotohero-zh-vocabulary.csv*, but since it is used to complement phrases, in which radicals are not as useful in phrases, it would not affect the program negatively. The data type for the records in both datasets is then represented as:

```
export interface HanziRecord {
    character: string;
    pinyin: string;
    definition: string;
    radical: string;
}
```

In parsing the datasets, due to differences in fields, it is required to build two different functions for

different datasets in order to distinguish them and avoid problems. Loading the dataset will require the function *loadDatasets* which returns an array of *HanziRecord*.

D. *Search Layer*

The backbone of this whole project is how string matching algorithms are used to retrieve each field of *HanziRecord* as stated above. This is handled by the search layer. The main idea is using optimal algorithms for certain usage, which can be summarized as follows:

- Knuth-Morris-Pratt algorithm for exact matching

  KMP algorithm is used for queries searching for definite Hanzi, pinyin or definition. Its fast processing time is preferred for responsive search, if and only if the query is already exact and definite, meaning it does not tolerate typographical errors. The algorithm is defined in *defsearch.ts*.

  Similar to KMP algorithm in general, this algorithm starts with computing the border function, in this program annotated as Longest Prefix Suffix (LPS) table. Note that this step is performed after converting query to lowercase to ensure case-insensitive searching.

```
// Build longest prefix suffix table
function computeLPSArray(pattern: string):
number[] {
    const m = pattern.length;
    const lps = new Array(m).fill(0);
    let len = 0;
    let i = 1;

    while (i < m) {
        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;
}
```

  Afterwards, the program iterates through all the datasets performing KMP search as stated in the function as follows. Note that the definitions and Pinyin are also normalized to lower case before calling this function.

```
// Search with KMP
function kmpSearch(text: string, pattern:
string, lps: number[]): boolean {
    if (!pattern) return true;
    if (!text) return false;

    const n = text.length;
    const m = pattern.length;
    let i = 0; // text index
    let j = 0; // pattern index

    while (i < n) {
        if (pattern[j] === text[i]) {
            j++;
            i++;
        }

        if (j === m) {
            return true; // Match found!
        } else if (i < n && pattern[j] !==
text[i]) {
            if (j !== 0) {
                j = lps[j - 1]; // use
longest prefix table to determine next
character to be compared
            } else {
                i++;
            }
        }
    }
    return false;
}
```

  The overall function for this algorithm is then wrapped in the function *performExactMatching*, which includes preprocessing, normalizing, searching, then the top 20 most similar records will be sent to the frontend to be shown in the interface.

- Weighted Levenshtein algorithm for approximate or fuzzy Pinyin matching

  The previous case shows to be ineffective for Pinyin-related queries typed with Latin characters which do not contain tones. To tolerate this, approximate string matching has to be used. In this particular case, a Weighted Levenshtein algorithm is chosen. The algorithm is defined in *pinyinsearch.ts*.

  This algorithm starts with defining groups of similar Pinyin and Latin characters, as shown below:

```
// group for levenshtein fuzzy
const PINYIN_GROUPS = [
    ['a', 'ā', 'á', 'ǎ', 'à'],
    ['e', 'ē', 'é', 'ě', 'è'],
    ['i', 'ī', 'í', 'ǐ', 'ì'],
    ['o', 'ō', 'ó', 'ǒ', 'ò'],
    ['u', 'ū', 'ú', 'ǔ', 'ù'],
    ['v', 'ü', 'ǖ', 'ǘ', 'ǚ', 'ǜ', 'u']
// 'v' is often used to substitute 'ü'
];
```

  Afterwards, declaring costs for tone difference, substitution, insertion and deletion is the backbone of this algorithm, as it tolerates errors. This is also the reason this algorithm is classified as fuzzy. In this particular case, it is defined as follows, but it can be improved or modified accordingly:

```
// fuzzy costs
const COST_TONE_DIFFERENCE = 0.1; //
difference in note should always be low
cost
const COST_SUBSTITUTION = 1;      //
difference in letter is costly
const COST_INSERT_DELETE = 1;     // as
much as insertion and deletion of the
letters
```

With those definitions cleared, the similarity can be calculated with the following function:

```
// calculate the similarity cost of the
words
function calculatePinyinSimilarity(source:
string, target: string): number {
    const s = Array.from(source);
    const t = Array.from(target);
    const m = s.length;
    const n = t.length;

    const d: number[][] = Array.from({
length: m + 1 }, () => Array(n +
1).fill(0));

    for (let i = 0; i <= m; i++) d[i][0] =
i * COST_INSERT_DELETE;
    for (let j = 0; j <= n; j++) d[0][j] =
j * COST_INSERT_DELETE;

    for (let i = 1; i <= m; i++) {
        for (let j = 1; j <= n; j++) {
            const cost =
getPinyinSubstitutionCost(s[i - 1], t[j -
1]);
            d[i][j] = Math.min(
                d[i - 1][j] +
COST_INSERT_DELETE,        // Deletion
                d[i][j - 1] +
COST_INSERT_DELETE,        // Insertion
                d[i - 1][j - 1] + cost
// Substitution
            );
        }
    }

    const distance = d[m][n];
    const maxLength = Math.max(m, n, 1);
    return Math.max(0, 1 - (distance /
maxLength));
}
```

with the definition of *getPinyinSubstitutionCost* as:

```
// helper function, checks how similar a
letter is to the ones in dataset
function getPinyinSubstitutionCost(left:
string, right: string): number {
    const l = left.toLowerCase();
    const r = right.toLowerCase();

    if (l === r) return 0;

    for (const group of PINYIN_GROUPS) {
        if (group.includes(l) &&
group.includes(r)) {
            return COST_TONE_DIFFERENCE;
        }
    }

    return COST_SUBSTITUTION;
}
```

This function will normalize the similarity cost (capped at minimum of 0.0 and maximum of 1.0) so that every record can be evaluated fairly. The cost is calculated at 1 - (distance/maxLength) or 0 if query is empty, which results in higher costs meaning the query is more similar to the record.

The main function is the function *performPinyinFuzzySearch.* The query will first be normalized to lower case and no spaces. A threshold is then declared to filter results to show only possible relevant matches. The function calculatePinyinSimilarity is then called for each normalized record being traversed, then the top 20 most similar records will be sent to the frontend to be shown in the interface.

- Brute force algorithm for radical matching
  The first two algorithms are used for big datasets. For radical matching, due to its small size, it is still efficient enough to use brute force for matching the radicals. After all, the radical field in HanziRecord is always only one character. Using an "optimized" algorithm will only result in redundant preprocessing hence decreases performance. The algorithm is implemented in *radicalsearch.ts*.
  The brute force approach is simple, implemented as follows:

```
// Brute force for radical search
function bruteForceSearch(text: string,
pattern: string): boolean {
    if (!pattern) return true;
    if (!text) return false;

    const n = text.length;
    const m = pattern.length;

    for (let i = 0; i <= n - m; i++) {
        let j = 0;

        while (j < m && text[i + j] ===
pattern[j]) {
            j++;
        }

        // pattern is inside text if the
loop managed to reach the length of the
pattern
        if (j === m) {
            return true;
        }
    }

    return false;
}
```

The main function is the function *performRadicalSearch.* Because this uses radical as a query which is still Hanzi, no normalization is needed. Due to its small range of results, no record limit is needed as well. The function just calls bruteForceSearch for every record, then the relevant records will be sent to the frontend to be shown in the interface.

E. Messaging Layer

The final layer of this extension is responsible for communication between the user interface as implemented in *main.ts* and the host webpage's Document Object Model (DOM) in *content.ts* and *messaging.ts*. The extension utilizes Chromium's Message Passing API to transmit serialized JSON payloads.

- *main.ts*
  main.ts acts as payload emitter. It initializes the whole application computation, wrapping all functions as stated above, while monitoring user interactions. When any changes happen in the interface–user clicks a result, type in query, *main.ts* handles them. For example, when a user selects a specific dictionary result, the script extracts the target Hanzi character and queries the browser to identify the currently active tab using *chrome.tabs.query*. Once the target tab is verified, main.ts invokes the *chrome.tabs.sendMessage* protocol to securely transmit a structured command—such as { action: 'focus', hanzi: hanzi }—across the environment boundary.
- *content.ts*
  content.ts acts as payload consumer and executor. It initializes *chrome.runtime.onMessage.addListener* persistently to intercept incoming messages from the extension popup. For each payload received, the listener evaluates the action flag to determine the appropriate response logic. *content.ts* bridges the messaging layer into the DOM Manipulation phase, for example and mainly by executing the *highlightText* function, translating the backend payload into physical visual feedback on the user's screen.
- *messaging.ts*
  messaging.ts covers a single function *sendHighlightCommand* which, as the name states, sends a message to execute highlighting for the input it receives. This is used to show which words the user searches.

IV. Results and Discussion

Few results of this project will be shown and analyzed by comparing its results visually. All test cases will try all three algorithms and the results shown in some webpages will be observed and analyzed.

Fig 4. Project Extension Base UI *(Source: Author's document)*

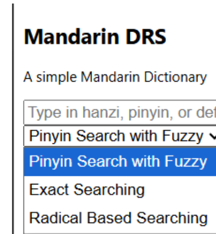

Fig 5. Project Extension Base UI - Algorithm Choices *(Source: Author's document)*

- Pinyin Search with Fuzzy

1. Website: https://baike.baidu.com/
   Input: Bai
   Result:

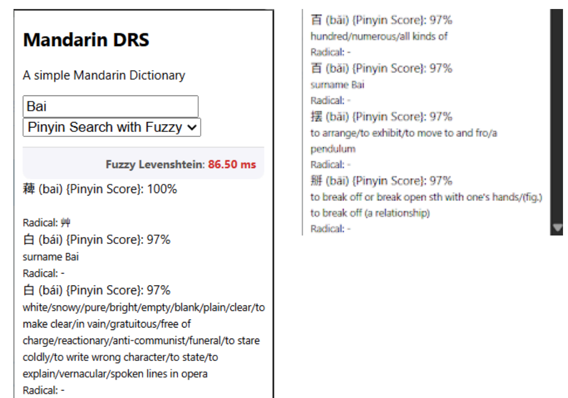

Fig 6. Pinyin Search Test Case 1 - Extension Result *(Source: Author's document)*

Fig 7. Pinyin Search Test Case 1 - Result 1 *(Source: Author's document)*

Fig 8. Pinyin Search Test Case 1 - Result 2 *(Source: Author's document)*

The extension result shows the search does normalization, for the uppercase 'B' in 'Bai' does not affect results shown. The Pinyin scores are also computed perfectly, for the first result ('bai') with no tone shows a perfect 100% score, while the others with difference in tone of 'a' shows a slightly lower score. Result 1 shows success in the code capturing the Hanzi 百 (bǎi; as shown in the fourth result in the extension UI) in the website, however for images like in the red-blue logo, the code fails to capture Hanzi in it. This is expected as this project does not involve image checking, however Result 2 shows an instance of failure in capturing the Hanzi as queried. This failure is marked with the red-colored doodle.

2. Website: https://baike.baidu.com/item/%E6%9C%89%E7%82

%B9%E7%94%9C/7122483
Input: ge
Result:

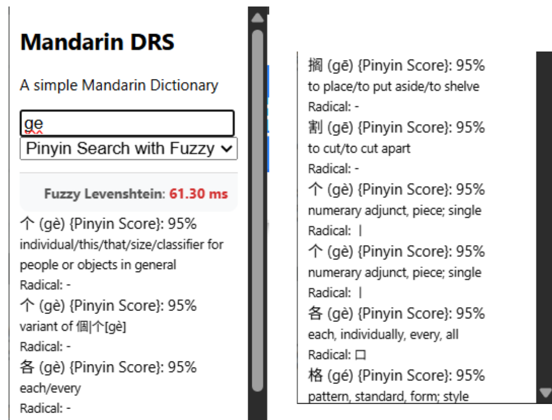

Fig 9. Pinyin Search Test Case 2 - Extension Result *(Source: Author's document)*

Fig 10. Pinyin Search Test Case 2 - Result 1 *(Source: Author's document)*

Fig 11. Pinyin Search Test Case 2 - Result 2 *(Source: Author's document)*

The extension result shows the Pinyin scores are also computed perfectly, for all results, which has difference in tone of 'e' shows a slightly lower score. Result 1 shows success in the code capturing the multiple Hanzi with Pinyin similar to 'ge', including 格 (gé) and 歌 (gē). Result 2 shows success on the filter system, whereas clicking on 歌 filters only itself and not 格, as doodled in red, anymore.

- Exact Searching

3. Website: https://baike.baidu.com/
   Input: 白
   Result:

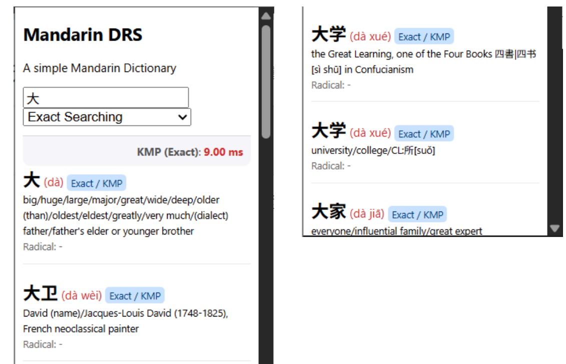

Fig 12. Pinyin Search Test Case 3 - Extension Result *(Source: Author's document)*

Fig 13. Pinyin Search Test Case 3 - Result 1 *(Source: Author's document)*

The extension result shows the search does its job perfectly, showing only the Hanzi 大 and phrases containing it. It also highlights the correct Hanzi in the webpage.

4. Website: https://baike.baidu.com/item/MIT%E5%8D%9A%E7%89%A9%E9%A6%86?fromModule=lemma_search-box
   Input:学, xué
   Result:

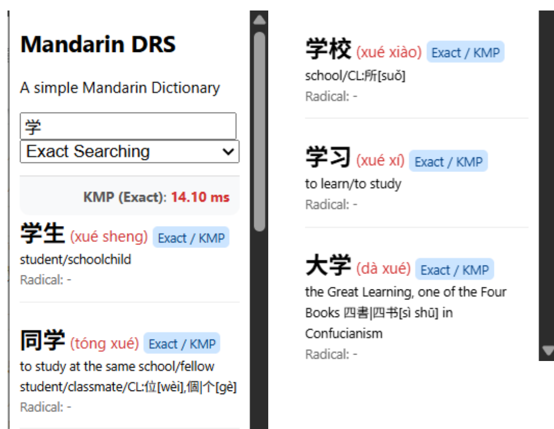

Fig 14. Pinyin Search Test Case 4 - Extension Result *(Source: Author's document)*

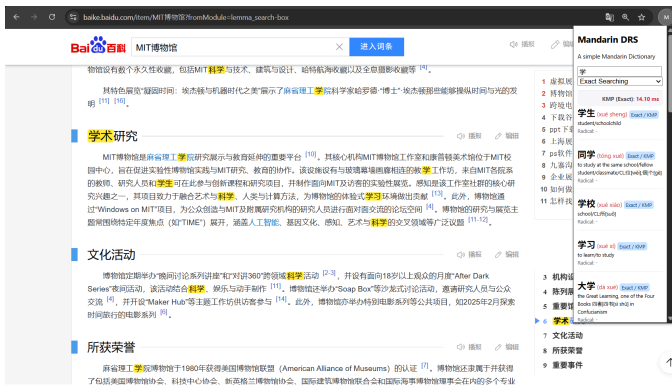

Fig 15. Pinyin Search Test Case 4 - Result 1 *(Source: Author's document)*

Fig 16. Pinyin Search Test Case 4 - Modified Query *(Source: Author's document)*

The extension result shows the search does its job perfectly, showing only the Hanzi 学 and phrases containing it, not only starting with it. Result 1 also shows how it filters multiple phrases with 学 just like the input. Result 2 shows how Pinyin input still works and shows same result.

5. Website: https://baike.baidu.com/item/MIT%E5%8D%9A%E7%89%A9%E9%A6%86?fromModule=lemma_search-box

Input: study
Result:

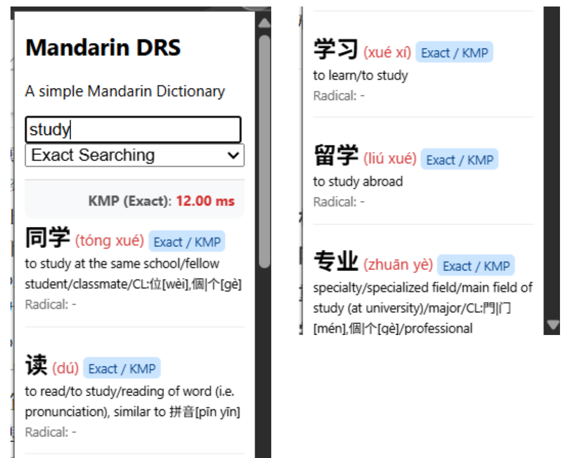

Fig 17. Pinyin Search Test Case 5 - Extension Result *(Source: Author's document)*

Fig 18. Pinyin Search Test Case 5 - Result 1 *(Source: Author's document)*

The extension result shows the search still does its job perfectly for definition input, showing only the results that has the word "study" in their definition. It also highlights only related Hanzi, like 研究 and 专业.

- Radical Searching

6. Website: https://baike.baidu.com/
Input: 白
Result:

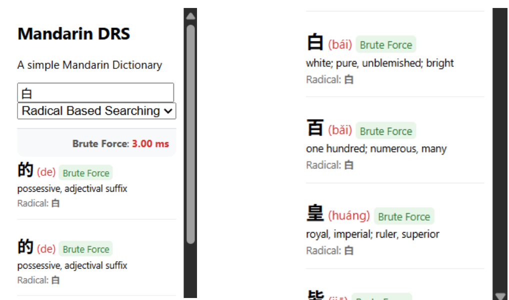

Fig 16. Pinyin Search Test Case 5 - Extension Result *(Source: Author's document)*

Fig 13. Pinyin Search Test Case 3 - Result 1 *(Source: Author's document)*

The extension result shows radical based searching works well by showing words with 白 as their radical. It also succeeds in highlighting the words wanted, including 百 and 的 in the webpage. Additionally, note it takes only 3.00 ms, this shows the perfect match of using brute force to search in the dictionary as it still takes only very little time.

V. Conclusion

The development of this project, Mandarin Dictionary Retrieval System as a Chromium browser-based extension demonstrates the successful integration of fundamental string matching algorithms into a modern web ecosystem.

Furthermore, the program proves that algorithmic routing—assigning specific algorithms to specific data characteristics—yields optimal computational performance. The Knuth-Morris-Pratt (KMP) algorithm significantly excels in the Exact Search mode, specifically when parsing extensive English definitions, as its border function effectively eliminates redundant backtracking. Conversely, the deliberate selection of the Brute Force algorithm for radical-based searching is practically justified; since Hanzi radicals constantly consist of only a single Hanzi, reducing the worst case scenario execution time. Finally, the implementation of the Weighted Levenshtein distance utilizing Dynamic Programming successfully resolves the primary challenge outlined in this paper: the limitation of standard Latin keyboards in typing Mandarin tone marks. By substituting absolute integer penalties with fractional costs accordingly, the algorithm provides a highly robust fuzzy search mechanism, which tolerates tone mark omissions while returning accurate dictionary matches with high confidence scores, thereby proving the system's reliability and effectiveness as an accessible assistive tool for language learners.

VI. Appendix

The github repository created for this project can be accessed on https://github.com/Tensai-033/Mandarin-Dictionary-Retrieval-System.

Acknowledgment

The author is extremely grateful to God for all the guidance during the process of learning leading to the competion of this paper. The author would as well thank his lecturer of Algorithm Strategy IF2211 in class K-1, Dr. Rinaldi Munir, for sharing their knowledge and for their patience throughout the whole lecture. Furthermore, the author treasures the constant support from his loved ones, family and friends, which undoubtedly complements his efforts.

References

[1] GeeksforGeeks, "Applications of String Matching Algorithms", [Online]. Available: https://www.geeksforgeeks.org/dsa/applications-of-string-matching-algorithms/ [Accessed 19-Jun-26]

[2] Munir, Rinaldi, "Pencocokan String (String/Pattern Matching)", [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/23-Pencocokan-string-(2026).pdf [Accessed 19-Jun-26]

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009

[4] GeeksforGeeks, "Naive algorithm for Pattern Searching", [Online]. Available: https://www.geeksforgeeks.org/dsa/naive-algorithm-for-pattern-searching/ [Accessed 19-Jun-26]

[5] GeeksforGeeks, "KMP Algorithm", [Online]. Available: https://www.geeksforgeeks.org/dsa/kmp-algorithm-for-pattern-searching/ [Accessed 19-Jun-26]

[6] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323-350, 1977.

[7] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707-710, 1966.

[8] https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0

[9] Google Chrome Developers, "What are extensions? - Chrome Developers," *developer.chrome.com*, 2023. [Online]. Available: https://developer.chrome.com/docs/extensions/mv3/overview/. [Accessed 19-Jun-26]

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, 19 Juni 2026

Ray Owen Martin - 13524033