Spell Checker

Project Report

Alyaman Huzaifa Massarani (900225917)
Mohamed Ibrahim Abdelmagid (900223215)
Saif Abd Elfattah (900225535)
Mohamed Khalil Brik (900225905)

CSCE221102 – Applied Data Structures
Dr. Amr Goneid
FALL 2023

Outline

I. Introduction
II. Problem Definition
III. Methodology
IV. Specification of Algorithms to be used
V. Data Specifications
VI. Experimental Results
VII. Analysis and Critique
VIII. Conclusions
References
Appendix

Abstract

The Trie Data Structure Project represents a critical endeavor in software engineering and algorithm design, addressing the pressing need for efficient data retrieval and management systems in the digital age. This project delves into developing and implementing the trie data structure in C++, showcasing its profound relevance to various market segments, especially within the technology sector. In a world driven by data-driven decision-making and user experience optimization, the project underscores the significance of rapid search capabilities and predictive text features in software applications.

The report navigates through the intricate layers of the project, from defining the problem of efficient word storage and retrieval to elucidating the chosen methodology for the trie's construction and utilization. Key algorithmic components, such as insertion and search algorithms, are meticulously outlined, revealing the trie's efficiency in handling vast datasets. The project adheres to stringent data specifications, utilizing a diverse and substantial word dataset for rigorous testing.

While the project has successfully met its objectives, it remains open to future enhancements, such as the adoption of smart pointers for memory safety and the extension of support to Unicode characters for broader applicability. In conclusion, the Trie Data Structure Project is a testament to the power of advanced data structures and the symbiotic relationship between innovative software solutions and the evolving needs of businesses in today's dynamic market landscape. This abstract offers a glimpse into the comprehensive report that delves deeper into the intricacies and achievements of this project.

Keywords

- Data Structures and Algorithms
- Trie Data Structure
- Prefix Tree
- Dynamic Data Structure
- Word Lookup
- Efficient Search Algorithm
- String Manipulation
- Lexicographical Sorting
- Memory Optimization
- Dynamic Memory Allocation
- Text Processing
- Character Indexing
- Alphabetical Ordering
- Search Optimization
- Autocomplete Feature
- Path Construction Using std::filesystem
- C++ Standard Template Library (STL)
- Smart Pointers (std::unique_ptr)

I. Introduction

The Trie Data Structure Project, while deeply rooted in software engineering and algorithm design, presents significant relevance and potential impact on various market segments. This project addresses a critical need for efficient data retrieval and management systems in today's digital landscape, where data-driven decision-making and user experience optimization are paramount. Businesses, especially those in the tech sector, continually seek ways to enhance their software offerings, making features like rapid search capabilities and predictive text increasingly vital. By implementing a trie data structure, the project offers a solution that can be integrated into various applications, from search engines and e-commerce platforms to text editing software and mobile apps. This enhances the performance and user experience of these applications and provides a competitive edge in the market. On the software side, the project stands as a testament to the importance of efficient algorithmic solutions in handling large volumes of data, showcasing how advanced data structures can be practically applied to solve real-world challenges in the software industry. Thus, this project not only contributes to the field of computer science from a technical standpoint but also aligns closely with market demands, highlighting the symbiotic relationship between innovative software solutions and business needs.

II. Problem Definition

The primary problem addressed by the trie data structure project is the efficient storage and retrieval of a large dataset of words, such as a dictionary or a word list. Traditional methods of storing such data in linear structures like arrays or linked lists result in huge search times, especially when dealing with large volumes of data. This inefficiency is particularly in applications requiring rapid access to data, such as autocomplete features in search engines or text editors, where quick and accurate suggestions are critical for enhancing user experience.

The trie, or prefix tree, solves this problem by organizing the dataset in a hierarchical, tree-like structure. This allows for faster search operations, especially when searching for words or verifying if a string is a prefix of any word in the dataset. Each node in the trie represents a character of the alphabet, and paths from the root to the leaf nodes represent words stored in the trie. This structure significantly reduces the time complexity of search operations, from linear time in the case of arrays or linked lists to time proportional to the length of the word being searched for, making it highly efficient for large datasets. Furthermore, the trie's ability to quickly find all words with a given prefix makes it ideal for implementing features like predictive text and spell-checking in applications.

III. Methodology

The spell-checker code is designed to efficiently identify and suggest corrections for misspelled words in a given sentence. The primary data structure used is a Trie, facilitating quick word storage, retrieval, and suggestion generation. The spell-checking algorithm processes user input, color-codes correct and incorrect words, and provides suggestions for misspelled words. Additionally, the program maintains statistics and offers options for user interaction.

Trie Class Functions

Trie Constructor:

Initializes a Trie object with an empty root node.

Node Constructor:

Initializes a Trie node with a boolean flag end and an array of child nodes.

insert:

Inserts a word into the Trie by creating nodes for each character.

search:

Searches for a word in the Trie by traversing the nodes.

suggest:

Generates suggestions for a misspelled word using character replacements, insertions, deletions, and swaps.

- **Character Replacement:**

  Replaces each character individually with a letter a - z, checking the Trie for valid words.

- **Character Insertion:**

  Inserts a character at each position in the word, checking the Trie for valid words.

- **Character Deletion:**

  Deletes each character individually, checking the Trie for valid words.

- **Adjacent Character Swapping:**

  Swaps adjacent characters, checking the Trie for valid words.

addWrongWordsToFile:

Appends wrong words and suggestions to a file ("wrong_words.txt").

updateStatistics:

Updates statistics for right and wrong words based on existing data and writes to a file ("statistics.txt").

printColored:

Prints a word with a specified color using ANSI escape codes.

Main function:

- Initializes a Trie with a dictionary file.
- Takes user input for a sentence, performs spell-checking, and updates statistics.

- Writes wrong words with suggestions to a file for memorization.

While the core spell-checking algorithm using a Trie is effective, the code could benefit from improvements in error handling, statistics tracking, and user interface clarity. Additionally, the color-coded output may be platform-dependent and could be made more portable.

IV. Specification of Algorithms to be used

Algorithm: Suggest function

vector<string> suggest(const string& wd);

Character Replacement:

```
for (int i = 0; i < word.length(); ++i) {
    for (char c = 'a'; c <= 'z'; c++) {
        string modifiedWord = word;
        modifiedWord[i] = c;
        if (searchWord(root, modifiedWord)) {
            suggestions.push_back(modifiedWord);
        }
    }
}
```

This section iterates through each character of the misspelled word (word) and replaces it with every letter from 'a' to 'z'. If the modified word exists in the dictionary (as determined by the searchWord function), it is added to the vector of suggestions.

Character Insertion:

```
for (int i = 0; i <= word.length(); ++i) {
    for (char c = 'a'; c <= 'z'; c++) {
        string modifiedWord = word;
```

```
        modifiedWord.insert(i, 1, c);
        if (searchWord(root, modifiedWord)) {
            suggestions.push_back(modifiedWord);
        }
    }
}
```

This section iterates through each position in the misspelled word and inserts every letter from 'a' to 'z' at that position. Again, if the modified word exists in the dictionary, it is added to the vector of suggestions.

Character Deletion:

```
for (int i = 0; i < word.length(); ++i) {
    string modifiedWord = word;
    modifiedWord.erase(i, 1);
    if (searchWord(root, modifiedWord)) {
        suggestions.push_back(modifiedWord);
    }
}
```

This part iterates through each character of the misspelled word and deletes the character at that position. If the modified word exists in the dictionary, it is added to the suggestions vector.

Adjacent Character Swapping:

```
for (int i = 0; i < word.length() - 1; ++i) {
    string modifiedWord = word;
    swap(modifiedWord[i], modifiedWord[i + 1]);
    if (searchWord(root, modifiedWord)) {
        suggestions.push_back(modifiedWord);
    }
}
```

This section iterates through the misspelled word and swaps adjacent characters. If the modified word exists in the dictionary, it is added to the vector of suggestions.
In summary, the suggest function generates a variety of possible corrections for a misspelled word by exploring different modifications. Each modification is checked against the dictionary (implemented as a Trie) to determine if it is a valid word. Valid words are then added to the vector of suggestions. These suggestions can be presented to the user for correction.

Algorithm: Trie.insert(word)
Input: A Trie object, word (string)
1. Set the current node to the root of the Trie.
2. For each character c in the word:
    a. Calculate the index of c in the node's children array.
    b. If the child node at the calculated index is null:
        i. Create a new Node.
        ii. Set the child node at the calculated index to the newly created Node.
    c. Move to the child node at the calculated index.
3. Mark the last node as the end of a word by setting the 'end' flag to true.

Algorithm: Trie.search(word)
Input: A Trie object, word (string)
Output: Boolean (true if the word is found, false otherwise)
1. Set the current node to the root of the Trie.
2. For each character c in the word:
    a. Calculate the index of c in the node's children array.
    b. If the child node at the calculated index is null, return false (word not found).
    c. Move to the child node at the calculated index.

3. Return true if the last node is marked as the end of a word, otherwise return false.

V. Data Specifications

In this trie data structure project, the input data primarily consists of a comprehensive dataset of words, specifically sourced from the "top-10000-words-in-english.txt" file obtained from "https://www.mit.edu/~ecprice/wordlist.10000". This file represents a significant and diverse collection of English words, making it an ideal candidate for demonstrating the trie's efficient data handling and retrieval capabilities. Each entry in the dataset is a string representing a single word, adhering to the lowercase English alphabet, thereby aligning with the trie's design to handle 26 letters. The choice of this dataset is strategic, as it encompasses a wide range of common English words, thus providing a realistic and challenging test bed for the trie. This dataset not only facilitates the testing of the trie's fundamental operations, such as insertion and search, but also enables the evaluation of the trie's performance in real-world scenarios, such as autocomplete functions or dictionary applications. A large, relevant word list ensures that the project outcomes are valid and applicable in practical settings where large volumes of text data must be processed efficiently.

VI. Experimental Results

```
Enter a word or a sentence (type 'exit' to end): you a
re lved by us
you are lved by us
Suggestions for lved: lived loved led

Enter a word or a sentence (type 'exit' to end): I hel
 poor peopl
i hel poor peopl
Suggestions for hel: del gel mel rel tel her hey heel
heel hell held hell help el hl he
Suggestions for peopl: people
```

The screenshot illustrates a spell-checking program's functionality, with correctly spelled words highlighted in green and misspelled words indicated in red. An impressive feature of the program is its ability to provide highly accurate suggestions for misspelled words, enhancing the user's writing experience by offering immediate corrections and improving text quality.

statistics.txt

```
1  Right Words: 6
2  Wrong Words: 3
3  Percentage of Right words: 0.66667
4
```

Run Statistics:

The program takes a significant step further by generating a text file that serves as a progress history for students aiming to improve their English writing skills. This file records the count of correctly and incorrectly spelled words and calculates and displays the accuracy of the user's writing. This feature

encourages students to practice writing quickly and provides them with a tangible record of their progress, making it an invaluable tool for enhancing their language skills.

wrong_words.txt

```
1  lved Suggestions: lived loved led
2  hel Suggestions: del gel mel rel tel her hey heel heel hell
   held hell help el hl he
3  peopl Suggestions: people
```

Weekly Learning Experience:

A separate file is also generated containing only the misspelled words and their suggestions. This personalized list allows students to focus on and memorize words they consistently spell incorrectly, enhancing their weekly learning and spelling retention.

VII. Analysis and Critique

Algorithm Analysis - Trie.insert(word):

The insert algorithm is used to insert a word into the Trie data structure. Let's analyze its key characteristics:

1. Time Complexity:

- The time complexity of inserting a word into the Trie is O(L), where L is the length of the word.
- The algorithm iterates through each character of the word, and for each character, it performs constant-time operations.

2. Space Complexity:

- The space complexity is O(L), where L is the length of the word.

- The space required is proportional to the length of the word as new nodes are created for each character.

3. Efficiency:

- The algorithm efficiently handles word insertion by traversing the Trie, creating nodes as needed.
- It ensures that common prefixes among words are shared, reducing memory consumption.

4. Practical Considerations:

- The algorithm is well-suited for dictionary-based applications where words need to be efficiently stored and retrieved.

Algorithm Analysis - Trie.search(word):

The search algorithm is employed to determine whether a given word is present in the Trie. Let's analyze its key characteristics:

1. Time Complexity:

- The time complexity of searching for a word in the Trie is O(L), where L is the length of the word.
- The algorithm traverses the Trie, examining each character of the word.

2. Space Complexity:

- The space complexity is O(1) as the algorithm uses a constant amount of additional space regardless of the word length.

3. Efficiency:

- The algorithm efficiently checks for word existence by navigating the Trie based on the word's characters.
- It is particularly efficient for applications requiring quick word lookup.

4. Practical Considerations:

- The algorithm is well-suited for spell-checking and autocomplete applications where rapid word retrieval is crucial.

Suggestions Generation: Generating suggestions for a misspelled word involves iterating through the word and trying various modifications, resulting in a time complexity of O(L^2), where L is the length of the word.

The overall time complexity of the spell-checking process is O(N * M * L^2), where N is the number of words in the sentence, M is the average length of words, and L is the maximum length of a word.

Space Complexity

Trie Storage: The Trie structure consumes space proportional to the total number of characters in all stored words. The space complexity is O(T), where T is the total number of characters in the dictionary words.

File I/O: The space complexity related to file input and output operations is minimal and can be considered constant.

The overall space complexity is dominated by the Trie structure, resulting in O(T).

VIII. Conclusions

In conclusion, the Trie Data Structure Project has demonstrated its significance in addressing the critical need for efficient data retrieval and management systems in today's digital landscape. By implementing a trie data structure, this project has showcased its potential impact on various market segments, especially in the tech sector, where rapid search capabilities and predictive text are essential for enhancing user experiences. From a software engineering perspective, the project has emphasized the importance of efficient algorithmic solutions in handling large volumes of data, highlighting the practical applicability of advanced data structures in real-world challenges.

References

Sedgewick, Robert, and Kevin Wayne. Algorithms. 4th ed., Addison-Wesley, 2011.

Weiss, Mark Allen. Data Structures and Algorithm Analysis in C++. 4th ed., Pearson, 2013.

Cormen, Thomas H., et al. Introduction to Algorithms. 3rd ed., The MIT Press, 2009.

Knuth, Donald E. The Art of Computer Programming, Volume 3: Sorting and Searching. 2nd ed., Addison-Wesley, 1998.

Meyers, Scott. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media, 2014.

Josuttis, Nicolai M. The C++ Standard Library: A Tutorial and Reference. 2nd ed., Addison-Wesley, 2012.

Price, Edward. "Wordlist 10000 - MIT." MIT, https://www.mit.edu/~ecprice/wordlist.10000.

Appendix

P.N: Code is provided on GitHub.