Spell Checker Project Report

Darko Konstantinovikj

December 4, 2024

1 Introduction

This report describes the implementation of a spell checker project. The goal of the project was to build a spell checker that can read text, check each word against a dictionary, allow users to add custom words, suggest corrections for misspelled words, and optimize performance for large inputs. The project also includes both unit and property-based testing to ensure correctness.

2 Task Overview

The task was to implement a spell checker in Haskell with the following features:

- **Console Interface**: Allow users to interact with the spell checker through the command line or another user interface.
- **Text Checking**: The spell checker should be able to read text and check each word against a dictionary to identify any misspelled words.
- **Custom Words**: Users should be able to add custom words to the dictionary, allowing for a personalized experience.
- **Suggestions for Corrections**: For misspelled words, the program should suggest potential corrections.
- **Performance**: The solution should be optimized for performance to handle large texts efficiently.
- **Error Reporting**: The program should handle errors related to file I/O and invalid inputs gracefully.
- **Testing**: The solution should include both unit and property-based tests to verify correctness and performance.

3 Architecture of the Solution

The architecture of the solution can be broken down into the following components:

3.1 Trie Data Structure

The spell checker uses a **Trie** data structure to efficiently store and search for words in the dictionary. The Trie allows for fast prefix-based searches and ensures that we can efficiently check if a word exists in the dictionary.

3.2 Dictionary Size

The dictionary used in the project contains over **300,000 words**. This large dictionary is stored within the Trie data structure, allowing for efficient searching even with such a vast number of words. Using a Trie ensures that checking for the existence of a word remains performant, as the time complexity for lookups is linear in terms of the length of the word, not the size of the dictionary.

3.3 Spell Checking

The core logic of the spell checker involves reading the input text, normalizing the words (i.e., converting them to lowercase), and checking them against the words stored in the Trie. If a word is not found, it is flagged as misspelled.

3.4 Custom Word Addition

Users can add custom words to the dictionary. These words are inserted into the Trie, ensuring they are available for future spell checks.

3.5 Suggestions for Corrections

If a word is misspelled, the system generates suggestions by finding words in the dictionary with a small Levenshtein distance from the misspelled word. This provides the user with a list of similar words that could be used as corrections.

3.6 Performance Optimization

The program is optimized for speed by using efficient data structures and algorithms. For example, the Trie allows for fast searching, and the Levenshtein distance algorithm is optimized to minimize unnecessary computations.

3.7 Error Handling

The program handles errors related to file reading/writing and invalid input gracefully. If a file is missing or corrupted, or if the user provides invalid input, the program outputs an error message and exits cleanly.

3.8 Testing

The solution includes both **unit tests** and **property-based tests** using HUnit and Hedgehog, respectively. Unit tests ensure that individual functions perform as expected, while property-based tests check the correctness of the spell checking logic under a variety of random inputs.

4 Architecture Decisions

4.1 Why Use a Trie?

The Trie is an ideal data structure for this problem because it allows for efficient searching, insertion, and prefix-based operations. Since a Trie organizes words in a tree structure, it makes checking if a word exists in the dictionary very fast, with a time complexity of $O(k)$, where k is the length of the word.

4.2 Why Use Levenshtein Distance for Suggestions?

Levenshtein distance is a widely-used metric for calculating how similar two strings are. It counts the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another. This is perfect for generating spelling suggestions, as it provides a way to identify the closest matches in terms of character similarity.

4.3 Why Test Libraries?

- **Hedgehog**: Hedgehog was chosen for property-based testing because it allows for generating a wide range of test cases and checking properties across large input spaces. This is important for ensuring that the spell checker handles various edge cases effectively.
- **HUnit**: HUnit was chosen for unit testing because it integrates seamlessly with the `Tasty` framework and provides a simple way to verify the correctness of specific functions.

5 Performance Investigation

Performance was an important consideration for this project, as the spell checker needs to process potentially large texts and dictionaries. Several steps were taken to optimize performance:

5.1 Trie Structure

Using a Trie allows for fast word lookups, which is critical when checking large texts against the dictionary. Since the time complexity of searching for a word in a Trie is proportional to the length of the word, it remains efficient even with large dictionaries containing over 300,000 words.

5.2 Levenshtein Distance Optimization

The implementation of Levenshtein distance was optimized to reduce unnecessary computations. By caching intermediate results and using dynamic programming, the algorithm avoids recomputing values for overlapping subproblems, thus speeding up the correction suggestion process.

5.3 Input Processing

Words are processed sequentially, one at a time, for spelling check. Additionally, the program normalizes words to lowercase before processing them, which helps reduce the number of comparisons needed for accurate matching.

5.4 Handling Large Inputs

The spell checker is designed to handle large inputs in a memory-efficient manner by reading and processing text files line by line. This approach ensures that memory consumption remains manageable, even when working with large dictionaries or documents.

6 Libraries Chosen

- **Tasty**: The `Tasty` framework was chosen because of its flexibility and compatibility with both unit and property-based testing. It allows for easy integration of different test types and provides a clean output format.
- **Hedgehog**: Hedgehog was used for property-based testing because it is well-suited for Haskell and allows for easy generation of random test cases. This is especially useful for testing properties of functions that deal with random data, such as the spell checking logic.
- **Data.Map**: `Data.Map` from the `containers` package was used for the Trie implementation, as it provides efficient key-based access and updates. While a pure Trie implementation is possible using custom data structures, `Data.Map` offers a simple and efficient solution.
- **Levenshtein Distance**: The Levenshtein algorithm was implemented using dynamic programming, which minimizes computational overhead and makes it suitable for generating suggestions for misspelled words in a large dictionary.