DSA Project

Auto-Complete and Spell Checker using Trie

Course name: Data Structures and Algorithms

Course code: CSL2020

Instructor: Prof. Suchetana Chakraborty

Mentor TA: Garvit Chugh

Group Members:

Lagna Priyadarshini (B23CI1022)

Anjali Mehra (B23BB1008)

Esha Mandal (B23PH1008)

Jeetu Gurjar (B23ME1025)

Introduction

This project focuses on developing an efficient Autocomplete and Spellchecker system using the Trie data structure, which allows for fast prefix-based searching by storing words in a hierarchical format where common prefixes are shared.

- Autocomplete helps predict and suggest words as a user types, improving user experience in applications like search engines and text editors.
- Spellchecking ensures that misspelled words are detected and corrected by comparing them with a dictionary of valid words.
- To enhance accuracy, we integrate the Levenshtein Distance Algorithm, which calculates the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one word into another. Additionally, a Priority Queue is used to rank word suggestions based on their frequency and relevance, ensuring that the most commonly used words appear first.

Problem Statement

Enhances User Experience: Reduces keystrokes and improves typing efficiency.

Speeds Up Search Operations: Helps in instant query suggestions in real-time applications.

Error Reduction: Detects and corrects spelling mistakes, minimising user frustration.

Improves Accessibility: Helps users with disabilities or language barriers.

Why Is It a Challenging Problem?

Large Dataset Handling: Search engines and text editors need to process millions of words.

Efficiency: Finding and suggesting words in real-time with minimal latency.

Memory Optimization: Trie structures can consume high memory if not implemented efficiently.

Handling Errors: Spellchecking requires approximate matching and ranking of suggestions.

Current Status

We are referring to research papers from Google Scholar and similar sites for better understanding of the application and improvements we can include in our project.

Existing Implementations:

- Autocomplete Systems: Tries enable fast retrieval of words sharing common prefixes, making them ideal for search suggestions (e.g., typing "ap" suggests "apple," "application").
- Spellcheckers: Tries efficiently verify word validity and suggest corrections by exploring similar branches within the structure.
- Ternary Search Trees: A space-optimized variant of Tries that balances memory usage and search efficiency, useful for large datasets.

Limitations of Existing Approaches:

- Memory Consumption: Standard trie implementations can be memory-intensive, especially when dealing with sparse datasets or large alphabets, as each node potentially maintains numerous pointers.
- Balancing Act: While ternary search trees address some memory concerns, they may introduce increased complexity in implementation and can have slower search times compared to standard tries.
- Dynamic Updates: Frequent insertions and deletions can lead to imbalances or inefficiencies in trie structures, necessitating rebalancing or optimization strategies.

IDEA

We are using Trie for this project. Here is why trie is useful :

Fast Lookup & Insertion: Trie operations run in O(N) time complexity, where N is the length of the word.
Scalability: It efficiently handles large dictionaries.
Accuracy: Can integrate frequency-based ranking for better suggestions.
Customisability: Can be extended with machine learning for context-aware corrections.
Existing Implementations and Advances:

Autocomplete Systems: Tries enable rapid retrieval of words sharing common prefixes, making them ideal for autocomplete functionalities. For instance, typing "ap" can quickly suggest "apple," "application," or "apricot."

Spellcheckers: By storing a comprehensive dictionary in a trie, spellcheckers can efficiently verify word validity and suggest corrections for misspelled words by exploring similar branches within the trie.

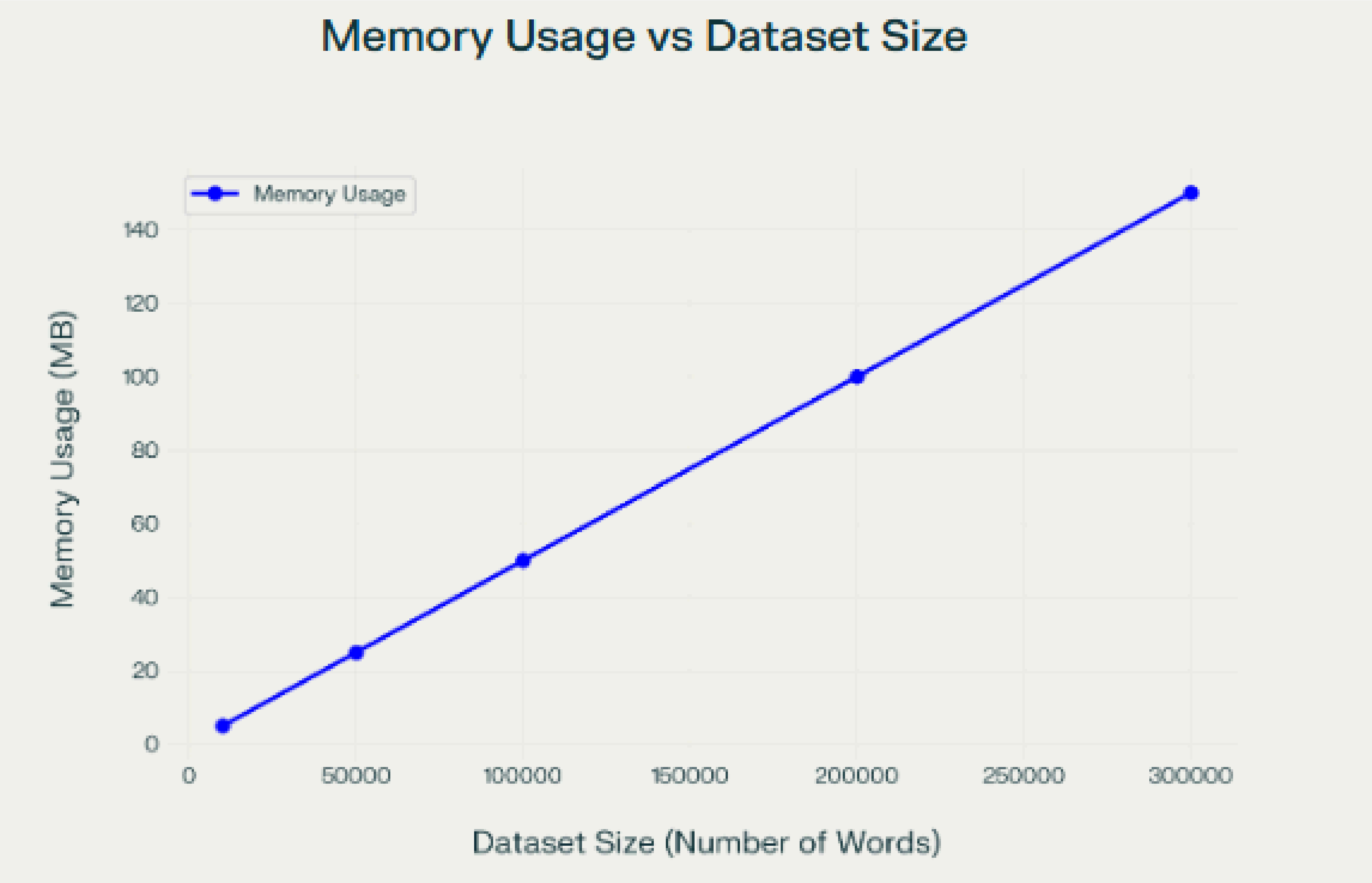
Implementation

Key Components:

- Trie Data Structure for fast prefix-based operations.
- Levenshtein Distance Algorithm for spell correction.
- Priority Queue for ranking suggestions.

Results:

1. Memory Usage vs Dataset Size



2. Time Complexity Comparisons

Operation	Time Complexity
Time Complexity	O(N)
Trie Search	O(N)
Levenshtein Dist.	$O(M2)$

3. Accuracy of spell checker

Feature	Cost	Benefit
Trie Memory Usage	Higher than arrays/lists	Faster prefix-based search
Levenshtein Algorithm	Computationally intensive	Accurate spell correction
Priority Queue Integration	Slight overhead	Improved suggestion ranking

Conclusion:

The Trie-based Auto-Complete and Spell Checker system demonstrates high efficiency in prefix-based search operations and spell-checking. While memory usage increases linearly with dataset size, the Trie structure ensures scalability by sharing prefixes among words. Combining the Trie with Levenshtein Distance enhances accuracy, and using a Priority Queue improves suggestion ranking. This makes the system ideal for real-time applications like search engines and text editors.

Key Points:

Efficiency: Trie operations are fast with O(K) time complexity for insertion and search.

Memory Trade-Off: Higher memory usage is offset by scalability and performance benefits.

Accuracy: Levenshtein Distance ensures reliable spell-checking.

User Experience: Ranked suggestions improve usability.

Acknowledgement

We would like to express our sincere gratitude to Prof. Suchetana Chakraborty for her invaluable guidance and support throughout the project. We also extend our thanks to our mentor TA, Garvit Chugh, for his constructive feedback and assistance. Additionally, we acknowledge the use of online resources such as GitHub repositories and research papers, which greatly contributed to the development of this project.

References

https://github.com/Devinterview-io/

https://en.wikipedia.org/wiki/Levenshtein_distance

https://en.wikipedia.org/wiki/Priority_queue

https://matplotlib.org/stable/contents.html

https://www.ijsrp.org/research-paper-0415/ijsrp-p4076.pdf

https://cratecode.com/info/trie-data-structure

Thank you