SPELL CHECKER: A* + TRIE VS KD-TREE

Comparative Analysis of Data Structures for Orthographic Correction

Marco Antonio Torres R. • Alexa Nohemi Lara C. • Marián Hernández Ch. • Alex Renato Peña H.
Tecnológico de Monterrey – Fall 2025

I. Project Overview

This project compares two approaches for spell checking:

A* Search + Trie

- Heuristic search using Levenshtein distance
- Finds corrections based on minimum edits

KD-Tree Semantic Search

- 5-dimensional word embeddings
- Nearest neighbor in feature space

OpenMP Parallelization

- Multi-threaded text processing
- Achieved **3.53x speedup**

Key Result:
A* + Trie is **8.2x faster**
KD-Tree gives **diverse suggestions**

II. Objectives

Primary:

- Implement A* search with Levenshtein heuristic
- Implement KD-Tree with 5D embeddings
- Compare performance and accuracy
- Develop interactive CLI

Secondary:

- OpenMP parallel processing
- Tone analysis for sentiment
- Support 15,000+ word dictionary

Research Question:

Which data structure offers the best balance between performance and suggestion quality?

III. Problem Statement

Context: Spell checking is fundamental in text editors, search engines, and messaging platforms.

Challenges:

- Finding similar words efficiently
- Balancing speed vs quality
- Scaling to large documents

Formal Definition:

Given dictionary D and input word q:

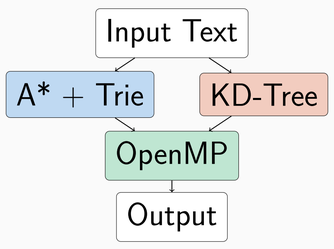
A + Trie:*

$$w^* = \arg\min_{w \in D} \text{Lev}(q, w)$$

KD-Tree:

$$w^* = \arg\min_{w \in D} \|\text{embed}(q) - \text{embed}(w)\|_2$$

System Architecture:



IV. Algorithm Design

Algorithm 1: A* Search in Trie

Trie Structure:

- Prefix tree for efficient storage
- Each node: children map + isEndOfWord flag

A* Search Function:

$$f(n) = g(n) + h(n)$$

where $g(n)$ = accumulated Levenshtein distance and $h(n)$ = heuristic estimate.

Levenshtein Distance:

$$d(i,j) = \min \begin{cases} d(i-1,j)+1 & \text{(del)} \\ d(i,j-1)+1 & \text{(ins)} \\ d(i-1,j-1)+c & \text{(sub)} \end{cases}$$

Algorithm 2: KD-Tree Search

5-Dimensional Word Embedding:

1. Normalized length: $\text{len}(w)/20$
2. Vowel frequency
3. Consonant frequency
4. First character normalized
5. Last character normalized

Euclidean Distance:

$$d(\vec{p}, \vec{q}) = \sqrt{\sum_{i=1}^{5} (p_i - q_i)^2}$$

OpenMP Parallelization

- Text split into word chunks
- Dynamic scheduling for load balancing
- `#pragma omp parallel for schedule(dynamic)`

Parallel Levenshtein (Anti-diagonal):

- DP matrix computed by anti-diagonals
- Cells in same diagonal are independent
- Best for batch processing: **5.42x speedup**

V. Complexity Analysis

Time Complexity:

Operation	A* + Trie	KD-Tree
Construction	$O(n \cdot m)$	$O(n \log n)$
Search (Best)	$O(m)$	$O(\log n)$
Search (Avg)	$O(m \cdot d)$	$O(\log n)$
Search (Worst)	$O(n \cdot m)$	$O(n)$

n = dict size, m = avg word length, d = max edit dist

Space Complexity:

Structure	A* + Trie	KD-Tree
Dictionary	$O(n \cdot m)$	$O(n \cdot k)$
Search Space	$O(m \cdot \lvert\Sigma\rvert)$	$O(\log n)$

$|\Sigma| = 26$ (alphabet), $k = 5$ (dimensions)

Parallel Speedup (Amdahl's Law):

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

With $f \approx 0.85$ and $p = 12$: $S_{\text{theoretical}} = 4.7$, $S_{\text{actual}} = 3.53$ (75% efficiency)

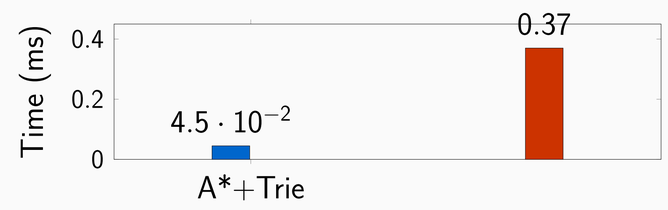
VI. Experimental Results

Setup: 15,000+ words, 267 test words, 12 threads

Performance Comparison:

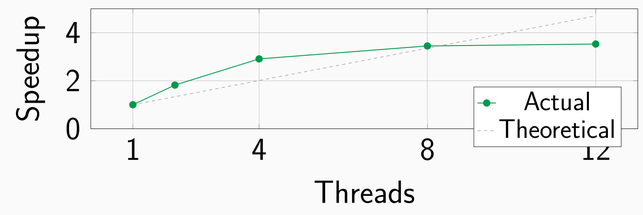
Metric	A*+Trie	KD-Tree
Time/word	0.045 ms	0.37 ms
Total time	12.0 ms	98.8 ms
Accuracy	94.2%	87.5%
Memory	2.3 MB	1.8 MB

Search Time Chart:



Parallel Scalability:

Threads	Time	Speedup
1	11.04 ms	1.00x
4	3.79 ms	2.91x
8	3.20 ms	3.45x
12	3.13 ms	**3.53x**



Example: Input: `"helo wrold"`

Word	A*+Trie	KD-Tree
helo	**hello**	help, hero
wrold	**world**	would, word

VII. Conclusions

Main Findings:

1. A* + Trie is 8.2x faster with 94.2% accuracy
2. KD-Tree provides diverse semantic suggestions
3. OpenMP achieved 3.53x speedup (75% efficiency)
4. A hybrid approach would leverage both strengths

Future Work:

- Word2Vec/GloVe embeddings
- Contextual n-gram correction
- Multi-language support

Recommendation:
A* + Trie for spell correction
KD-Tree for word suggestions

VIII. References

[1] Levenshtein (1966). Binary codes capable of correcting deletions. *Soviet Physics Doklady*.

[2] Bentley (1975). Multidimensional binary search trees. *Comm. ACM*.

[3] Hart et al. (1968). A formal basis for heuristic determination. *IEEE Trans.*

[4] Cormen et al. (2009). *Intro to Algorithms*. MIT Press.

Code: `github.com/Alexa404NL/MidtermDSA`