Harper FST Spellchecking

How leveraging data structures improved spellchecking speed by 90%

Grant Lemons

December 27, 2024

Introduction

About a year ago, a friend and classmate of mine, Elijah Potter, created a spellchecking software for developers called Harper, and it quickly became a part of our friend group's development environments. I enjoyed using the tool, and since it had become a core part of my writing and development environments, I became invested in making it better. In September of this year, I started contributing to the project on GitHub, and—after some smaller changes to familiarize myself with the project, chose to take on a rather large ticket to improve performance of the project's spellchecking component. The end result was an ~90% improvement in uncached spellchecking speed.

Today I'll go over some of the changes that led to this improvement, as well as the underlying structures that made it possible.

Original Implementation

Spellchecking in Harper is rather simple: words are compared to the dictionary, those not in the dictionary are fuzzy-found on the dictionary, possible matches ranked, and the top three returned to the user as suggested edits.

Pruning

The fuzzy-finding is based on Levenshtein edit distance, which is the number of character changes in an existing word in the dictionary needed to transform to the misspelled word. This is rather costly to run on every word in the dictionary, so we want to reduce the words we actually run it over. Pruning is reducing the number of available options through rougher, yet faster, methods in order to reduce the edit distances we need to calculate.

In the original implementation, this was accomplished by only considering words of a length allowed by the edit distance. So, for instance, if the maximum acceptable edit distance is 3, and the misspelled word was 6 characters long, only words of lengths 3–9 characters were

considered. The edits required to reach these lengths would be removing or adding three characters respectively. Any longer or shorter, and the edit distance would be guaranteed to be over our threshold.

To accomplish this pruning action, a `words` vector was built from Harper's curated dictionary, and sorted by length. An additional array (`word_len_starts`) stored the indexes at which the values in the `words` vector transitioned between word lengths. In order to prune the possible words, a slice over the `words` vector was taken using the transitional indexes for the appropriate word lengths indicated by `word_len_starts`.

Inefficiencies and Overhead

Once the words were pruned, the Levenshtein distance was calculated individually for each remaining word. A hash map was used to associate metadata with each word, accessed for every remaining word as a part of the result ranking process.

There are some pretty clear issues with this implementation. Because of the way words were stored, the Levenshtein distance is calculated individually, which is suboptimal. The metadata hash map also isn't ideal, since the overhead from the hash map is slower than just associating metadata by storing a pair of the word and metadata in the `words` vector.

FST-Based Implementation

The new implementation for Spellchecking is faster for several reasons, but is limited by the underlying data structures. Because of this, it isn't ideal for all cases of spellchecking in Harper. For instance, the dictionary needs all words to be inserted in lexicographical order, which means it can't be mutable. Additionally, there is a fairly significant overhead associated with constructing one. This isn't particularly significant at startup, as some delay is expected, but it would slow use down significantly in cases where the dictionary changes at runtime.

Appropriateness and Use-Case

Harper's spellchecking is done over a combination of four dictionaries: the Curated Dictionary, User Dictionary, File Dictionary, and Identifier Dictionary. Of these, only the Curated Dictionary is particularly suitable for the new implementation due to the aforementioned mutability restriction. Dictionaries that can be updated at runtime need to be reconstructed from scratch, and because of the overhead associated with reconstructing the dictionary, the performance gains are outweighed by the costs. Thankfully, searches in the Curated Dictionary make up almost all the runtime of spellchecking, so improvements are strongly reflected in the tool as a whole.

Those that require runtime mutability, thankfully are extremely short in comparison to the Curated Dictionary. As such, the old implementation—called the `FullDictionary` implementation—is more appropriate, and so is used here. Using multiple implementations

is made possible via the use of a `MergedDictionary`, which wraps several dictionaries and merges the outputs of each method across each of its "children". Because of this, we can merge mutable and immutable dictionaries of multiple types, making it possible to use the appropriate spellchecking implementation for each use-case (via dictionary type) across Harper.

DFA Dictionaries and Levenshtein Automata

The new implementation uses the FST crate to store the dictionary in an implementation of a Deterministic Finite Automata (DFA). The DFA stores words in a similar manner to a trie, but with combined suffixes as well as prefixes. The accepting states for this automata are the states immediately following the transition representing the final letter in each word, indicated below by double circles.

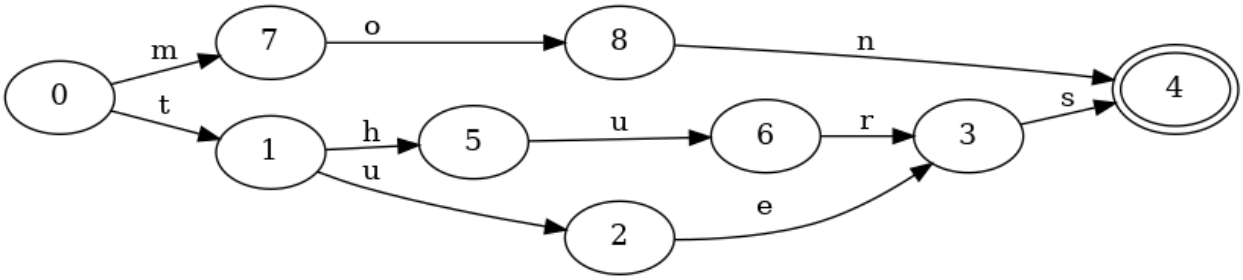


Figure 1: Example of words stored in a DFA, the stored strings are "mon", "tues", and "thurs". Credit to Andrew Gallant.

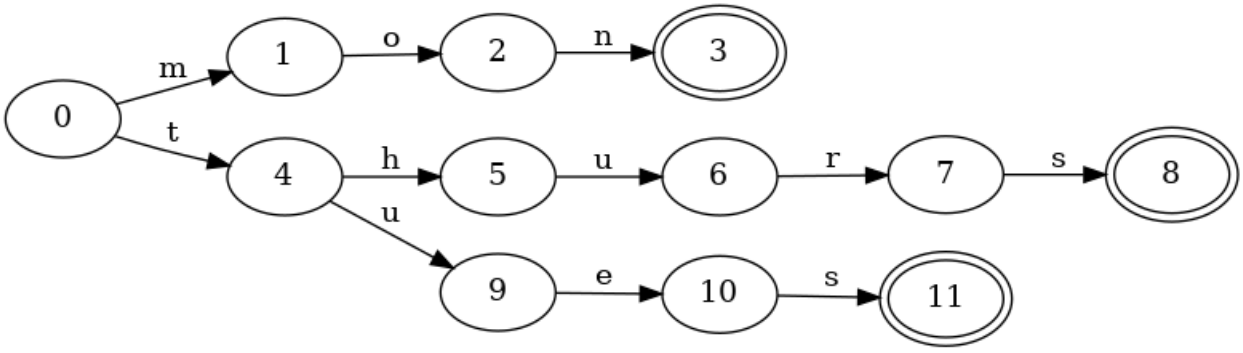


Figure 2: Example of an equivalent trie. Suffixes are not combined. Credit to Andrew Gallant.

Similarly, the Levenshtein automata can be built into a DFA accepting all words within a given edit distance of the misspelled word. These automata are complex, even with small acceptable edit distances, but it is rather easy to calculate the intersection of two DFA through the product construction. Accepting states in the intersection automata are those that are accepting in both the dictionary DFA and the generated DFA for the Levenshtein Automata. [Example of Product Construction] [Formal Product Construction]

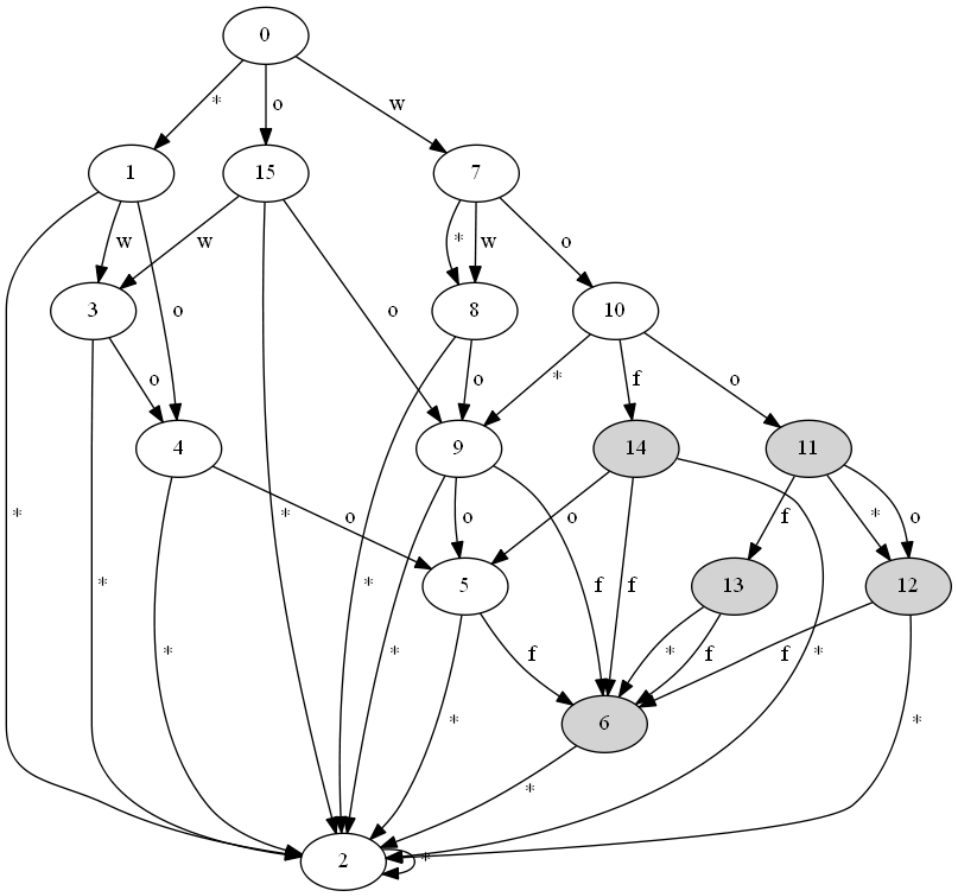


Figure 3: Example of a DFA built from a Levenshtein Automata for the word “woof”, accepting words with a maximum edit distance of 1. Shaded nodes represent accepting states. Credit to Jules Jacobs.

In order to associate each word with metadata, a small variation within the FST crate was used that associates weights with transitions (called a Deterministic Finite Transducer), leading it to function as a map, associating words to integer values. This is valuable for our dictionary implementation, as we associate metadata with each word, which we can store in a vector, indexed through our values in the transducer.

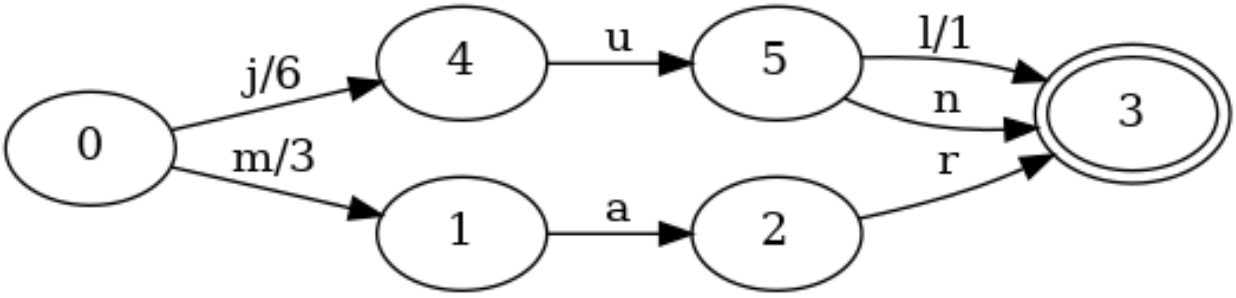


Figure 4: Example of Deterministic Finite Transducer. Values are “jul”: 7, “jun”: 6, “mar”: 3. Credit to Andrew Gallant.

DFA Further Reading

- Andrew Gallant — Index 1,600,000,000 Keys with Automata and Rust
- Jules Jacobs — Levenshtein automata can be simple and fast
- Paul Masurel — Of Levenshtein automata implementations
- Klaus Schulz and Stoyan Mihov — Fast String Correction with Levenshtein-Automata
- Madhusudan Parthasarathy — The product construction: Closure under intersection and union

Implementation Performance and Bottlenecks

While the actual operation of taking the intersection of the two automata is fast, there is a large upfront cost to construct the Levenshtein DFA from a given word. As our application runs spellchecking for each word of a document, it is desirable to hoist as much of that operation out of the spellchecking operation, which we achieved through the `levenshtein_automata` crate’s `LevenshteinAutomataBuilder`. This builder can quickly construct automata for a given distance, but is slow to construct. As such, we store a thread-local collection of builders for associated distances. Only the first spellchecking operation with a given maximum edit distance will construct a builder and add it to the collection.

Likewise, the Dictionary DFA is very slow to construct, taking about 100 milliseconds, but we can abstract it out to be thread-local and build it at startup, which is fast enough to be unnoticeable at startup. We looked into pre-computing the automata, but the additional size made it prohibitive for web-based uses of Harper, and the dependency issues created were great enough that we settled for startup construction.

In total, the changes made by switching to this implementation decreased measured spellchecking time by ~90%, and increased startup time by only ~100 milliseconds. An incredible tradeoff, which only improves the longer the document or the more time spent editing in the same session.

Conclusion

Doing the work to implement this new dictionary not only improved the project's spellchecking, it accomplished another goal, it helped me to understand the project on a deep level, which has helped me to make further contributions.

One of my goals for the last few years has been to feel comfortable entering an open-source project, understanding it, and working to improve it. While I had a lot of help getting there on this project (bothering Elijah before and after lectures), getting my hands dirty here has made me realize that goal is coming to fruition. I would urge anyone who has a tool they rely on all the time to take some time trying to make it a little bit better. It may not change the world, but it can improve your workflow, lead you to learn, and maybe just make someone's day.