Levenshtein Automata

Amy Wang
Stanford University
jwang19@stanford.edu

Li Deng
Stanford University
dengl11@stanford.edu

Baige Liu
Stanford University
liubaige@stanford.edu

Abstract

As our final project for CS166-Data Structure at Stanford, we studied the Levenshtein automata as a mechanism for spelling candidate generation within a certain maximum edit distance. We first implemented the construction of Levenshtein automata given a word, and construction for a large corpus as a trie as well as a Patricia trie. To verify its correctness and show its efficiency, we also implemented a naive approach for brute-forcing all candidates with a hash table for the corpus.

To facilitate easier understanding of the data structure, we built a Levenshtein walker, which interactively takes input from the user, and presents a step-by-step state transition on the web interface. We then further explored candidate generation for weighted edit distance and with extended operation including transposition.

1 Introduction

Given the target word, we often want to find its similar strings. The measurement of word similarity is often done by using Levenshtein distance[1] or n-gram distance[2]. In this paper, we will use Levenshtein distance as our measurement tool. Levenshtein distance is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. Given two input strings with lengths m and n seperately, dynamic programming can be used to calculate the Levenshtein distance which leads to quadratic time complexity. If the user gives some input string with spelling errors, we want to give reasonable suggestions whose Levenshtein distances to the target word are within a certain range. However, it is inefficient to compare it with each word in the dictionary and use dynamic programming to find out the set containing all the strings that are close to the target word. The solution is to use a Levenshtein automaton provided that our dictionary is stored in a definite finite state machine(or a trie). A Levenshtein automaton[3] for a string w and a number n is a finite state automaton that can recognize the set of all strings whose Levenshtein distance from the target string w is at most n.

The rest of this paper is organized as follows. In section 2, we provide with a basic review of Levenshtein automata and its main principles. In section 3, we include our experimental results of comparing Levenshtein automaton approach versus the naive algorithm, and time complexity analysis on the preprocessing time and runtime.

1.1 Levenshtein Edit Distance [4]

Mathematically, the Levenshtein distance between two strings a, b (of length $|a|$ and $|b|$ respectively) is given by

$$\mathrm{lev}_{a,b}(|a|, |b|)$$

where

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j) + 1 \\ \mathrm{lev}_{a,b}(i,j-1) + 1 \\ \mathrm{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Then given string s, max edit distance d and the dictionary D we want to find the set S, such that $S = \{a \in D | lev_{a,s} <= d\}$.

1.2 Corpus Pre-processing

We use the standard webster version 2 dictionary as our corpus, which reisdes in `/usr/share/dict/web2` for almost every computer, and has 235884 words after filtering out words with non-alphabetic characters. Firstly, to efficiently store all the words and enable fast retrieval, we need to convert the whole dictionary to a trie. We use a Python dictionary representation for it and each dictionary corresponds to one trie node with the keys being the children trie nodes.
Then, we further develop the trie into a finite state machine, which is not difficult since tries are often recognized as finite state machines. As long as we have both of them being finite state machines, we can traverse those two DFAs following the rule that we only follow the edges that both DFAs have in common and output the corresponding word when it causes the two DFAs to be in their final states.

1.3 Automaton Construction

Each state is a combination of the number of characters and the number of errors. Therefore, we have $(n + 1)(k + 1)$ states in total. We will use a tuple representation for the states in our implementation. In each state, we will add the transition from the current state to other states. And the transition rule is as follows. Every time we have a correct character, we will add an arrow pointing from the current state to the right state. Then as long as we don't exceed the limit in terms of the number of character and the number of errors, we will add transition from the current state(i, e) to the top-right state$(i + 1, e + 1)$ and top state$(i, e + 1)$for any character we can possibly have in the alphabet. We will also add an ϵ transition, representing the deletion of a character, from the current state(i, e) to the top-right state$(i + 1, e + 1)$.

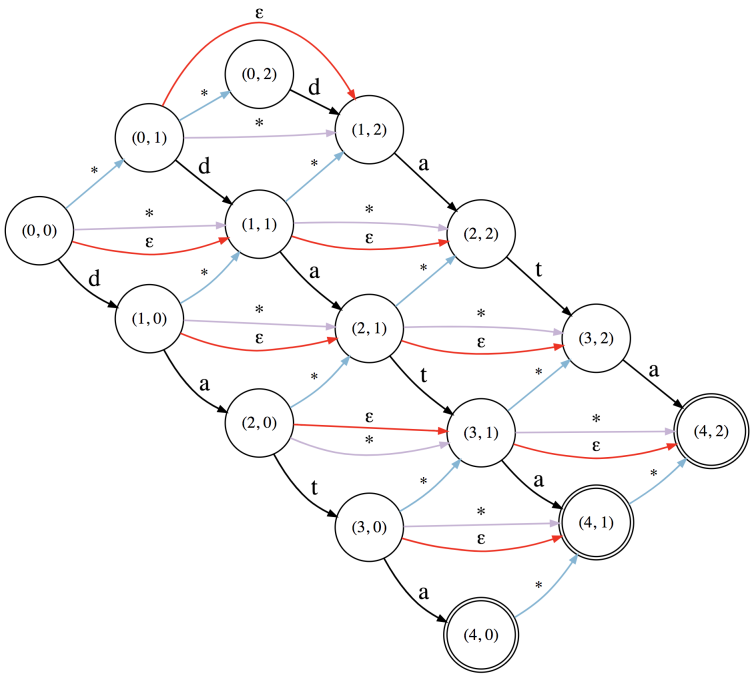


Figure 1: The NFA for a word "food" with max edit distance of 2.[5]

1.4 Matching

Here, we use a stack implementation. We will firstly push the initial two nodes from the two DFAs to the stack. While the stack is not empty, we keep popping from the stack and we push to the stack all the matched children nodes and record the path. In addition, whenever we find the two nodes that both in their ending states, we will output the pair. In this way, we can find all the words that are both in the corpus dictionary and acceptable by our Levenshtein automaton.

2 Implementation

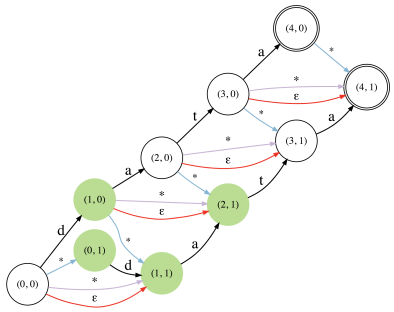
2.1 Naive Approach

In addition to Levenshtein automata, we have also implemented a naive model as our baseline to compare the Levenshtein automaton with. Although calculating pairwise edit distance using dynamic programming and looping through the whole corpus to pick the right words is the most straightforward solution to this problem, we consider it too naive to be useful baseline model and therefore decide to implement a stronger baseline model using the approach described as following.

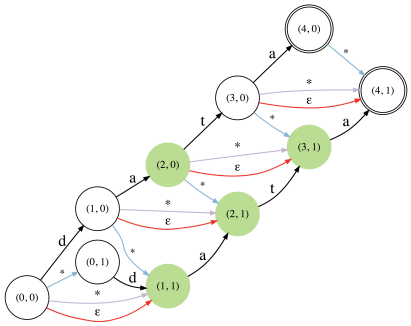
For a given word to find its similar words within edit distance k, we first compute all the candidates within a edit distance of 1. For $k = 1$, we compute all the pairs by splitting the word into two parts, after which insertions, deletions, and substitutions can be easily obtained from these pairs. For $k > 1$, we just iteratively compute all edit-1 candidates based on results within $k - 1$. Finally, we use the hash table for the corpus to check if a candidate word is valid.

2.2 Levenshtein automata

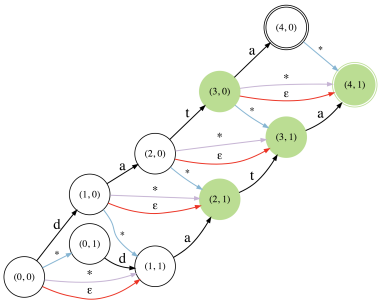
We use an online python module *automata(https://github.com/caleb531/automata)* to construct the DFA for a word. First a NFA is built, where each state is labeled (i, e) to keep track of what are the characters consumed so far, and how many edit operations have been used. And the transitions are based on insertions, substitutions and deletions as explained before. Now we use an example from our interactive explorer to further explain this.



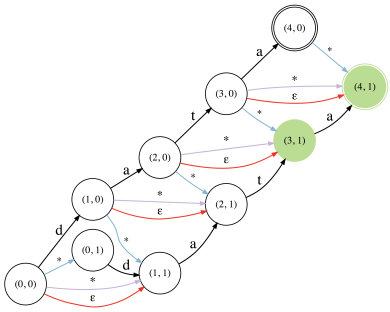
(1) Input: d



(2) Input: da



(3) Input: dat



(4) Input: date

Figure 2: NFA for the word "data"

To construct the Levenshtein Automation, we have to consider:

- **Start State:** The initial state is $(0, 0)$, meaning empty string with no edit operation.
- **Transitions:** There are four kinds of transitions (one regular transition plus three edit transition):
  - **Regular Input:** From state (i, e) to state $(i + 1, e)$. When we encounter an expected character, we just transiition to $(i + 1, e)$ without paying for any operations; like the $(0, 0)$ to $(1, 1)$ transition in the figure (1) above;
  - **Input Characters:** All alphabet letters from a to z are the input characters;
  - **Insertions:** An $*$ transition from state (i, e) to state $(i, e+1)$. For any character input, we can take it as an insertion at current place with 1 unit of cost; like the transition from $(1, 1)$ to $(2, 1)$ when the input is da in the figure (2) above;
  - **Deletions:** An ϵ transition (i, e) to $(i + 1, e + 1)$. Without consuming any input, we can jump to $i+1$ by deleting the $(i+1)^{th}$ character with 1 unit of cost. Like the $(3, 0)$ to $(4, 1)$ transition in the figure (3) above, which explains why *dat* can be accepted.
  - **Subsitutions:** An $*$ transition from state (i, e) to state $(i + 1, e + 1)$. By consuming any character, we can jump to $i + 1$ by subsituting the original $(i + 1)^{th}$ character as the new input character. Like the transition from $(3, 0)$ in figure (3) to $(4, 1)$ in figure (4) above, which causes *date* to be accepted by the automata, meaning that *date* is matched with *data* within 1 edit distance.
- **Accepting States:** All states with $(i = n, e)$ are all accepting states with $0 \leq e \leq k$, where n is the length of the orignal word.

As a side story, it turns out that the original implementation of the *automata* module has a bug for converting a NFA to a DFA when the initial state has ϵ transitions, which is exposed in our case. So we opened a Github issue and a Pull Request.

2.3 Weighted Edit Distance

So far we have been talking about edit operations, i.e. insertions, deletions and substitutions with the same cost. How about when different operations have different costs?

Like the amortized Banker's method, we allocate the word with an initial k credits, and only transition to the next state if we have enough remaining credits.

For the naive approach, we can no longer iteratively call the $edits1$ function until k becomes 0; so we instead use a stack to do a DFS with $(word, credits)$ for each state. For construction of the Levenshtein NFA, the states are no longer a regular grid of nk

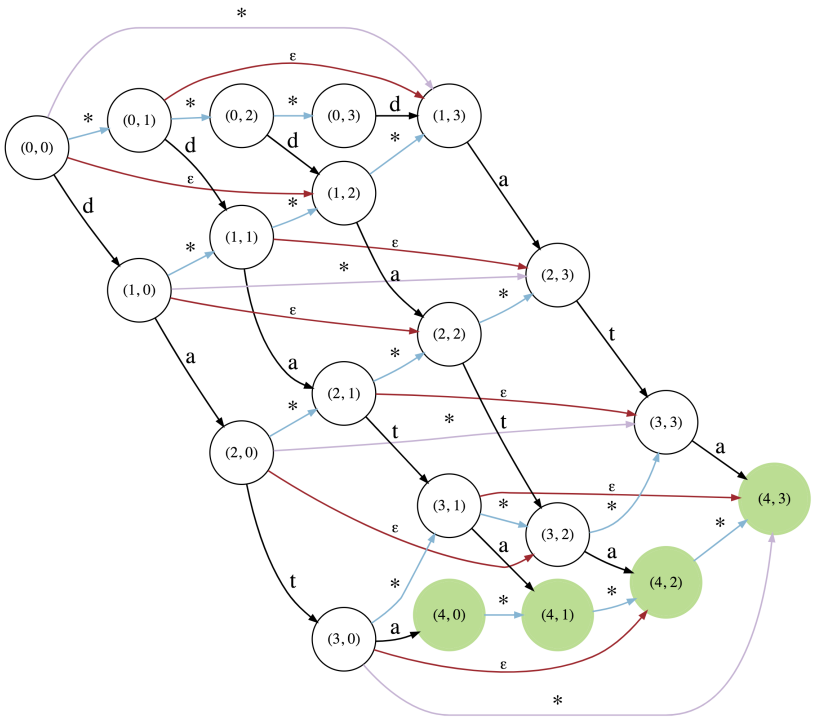


Figure 3: Weighted NFA for the word "data".

We did implement the weighted Levenshtein Automata, and present the brief runtime comparison with naive approach in the next section.

2.4 Extended Edit Operation: Transposition

The standard edition operations just include insertions, deletions, and substitutions. However it might be often the case that two neighboring characters are transposed, as users type with a keyboard.

In this case, the original Levenshtein Automation does not work, since the transposition ab to ba will be equivalent to one deletion of a with one insertion of a, which results in a cost of 2, instead of just 1.

Initially we tried adding special transitions between states to reflect the tranposition operation, by combining a deletion with on insertion into one extra transition with cost 1. However, only insertion of the previously deleted character is equivalent to transposition, which requires us to remember the previous character in the original word. And such information is lost in the standard Levenshtein Automation.

This inspires us about the second approach: expand the states in the Automation from a pair of (i, e) to (pre, i, e) to keep track of the previous character in the word. So that insertions of the previous character can also have cost 0.

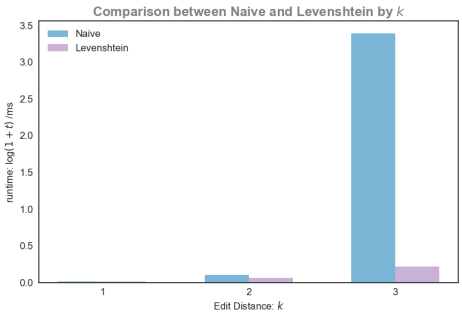
Due to time constraint, we did not implement the extended edit operation, but we think it should not be too difficult with our expanded-state idea.

3 Results & Analysis

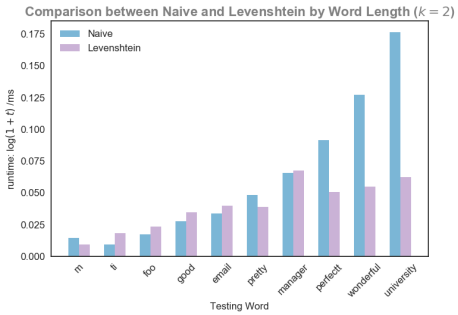
3.1 Efficiency comparison

As can be seen in Figure 4, the Levenshtein automata outperform the naive approach almost all the time, especially when word lengths and k are relatively large.

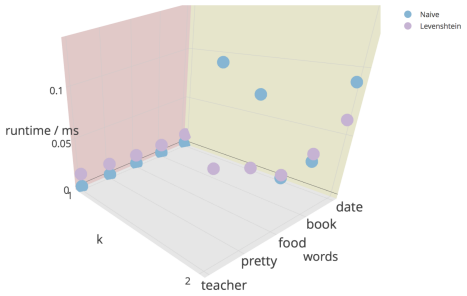
- **As max edit distance k grows**, the runtime of naive approach grows exponentially, being much more inefficient than the automation;
- **As words become longer**: in the naive approach, there are more splits and more reconstructions, so the runtime grows very fast. However the number of state in the Levenshtein NFA just grows by no more than a constant factor; and for long word, since it is walking on a trie, it soon falls off and stop the search, resulting in even lower runtime for longer words.
- **Weighted Edit Distance:** Since for a large k, the runtime of the naive approach is incomparable to the Levenshtein automata, so we limit k to be no more than 2 to just see the effect of adding different weights. The 3D plot below is the runtime with $(cost_insert = 2, cost_delete = 3, cost_substitute = 2)$. And the Levenshtein Automata is also generally faster than the naive approach.



(1) Runtime by k



(2) Runtime by word length



(3) Runtime for the weighted automata

Figure 4: The runtimes of the naive approach and using Levenshtein automata

3.2 Space Comparison for Corpus Preprocessing:

Instead of loading the large corpus from the disk and then construct the preprocessed data structure every time, we instead save the preprocessed data structure as a python `pickle` file on disk to save time, and compare their serialized size for approximate comparison.

As the table below shows:

- **An `automata` object**: saved as an automata is the most space-inefficient
- **A Hash Table**: saved as a python `set` is the most space-efficient;
- **Trie**: saved as a multi-level python dictionary, where each key can be only one character. The total number of nodes is 758659;
- **Patricia Trie**: compress the trie by absorting silly nodes into one, saved as a multi-level dictionary, where each key can be multi-characters, but each dictionary must have more than one key. The total number of nodes is 314372, and the serialized size is also smaller than the size of serialized trie.

Size	Approach
353M	automaton object
4.4M	Hash table
7.3M	Trie
5.5M	Patricia Trie height

3.3 Expected behavior on large k

Now we switch from actual implementation to more theoretic analysis. The NFA preprocessing is quite fast – kn, or just n for fixed k – but backtracking can potentially take exponential time. The DFA conversion can be determined completely by k, as the only information necessary to decide what a transition is is the relative ordering of the state and the positions in the set making up the state that match an input symbol. When k is fixed, then, there are a constant number of possible states and transitions per character. Thus, for the application of spelling correction, where $k = 1$, the algorithm using Levenshtein automata is much faster than the traditional dynamic programming algorithm for finding Levenshtein distance: let n represent the length of the word to be spell-corrected, and D the sum of the lengths of the words in the dictionary; using a Levenshtein automata is $\langle O(n), O(D)\rangle$ compared to $\langle O(1), O(nD)\rangle$ using dynamic programming and $\langle O(1), O(n^k)\rangle$ for the naive approach we implemented.

However, this is not necessarily true when k is not fixed. The general preprocessing time for arbitrary k is $O(nk \log k 2^{k^2+\min(2k+1,W)})$ (which is itself upper bounded by $O(nk \log k 4^k 2^{k^2})$). To see this, note that:

- For any position, only $\min(2k+1, n)$ characters in the input must be considered to determine the next state, as anything more would either run past the end of the input string or result in too many errors. There are therefore $2^{\min(2k+1,n)} = O(2^{\min(2k+1,n)}) = O(4^k)$ different possibilities for the characteristic vectors that act as the symbols in the automaton. (During runtime, it's assumed that it's possible to compute these vectors for any input in O(1).)
- The paper defines a DFA construction a little more adapted to the purpose of Levenshtein automata, which allows the maximum number of NFA states in each DFA state to be k^2. Therefore, the number of ways to fill these sets can be counted by $\binom{k^2}{1} + \binom{k^2}{2} + ... + \binom{k^2}{2k+1}$. Because the sum of the ith row of Pascal's triangle is equal to 2^i, this sum is upper-bounded by $O(2^{k^2})$ and lower-bounded by $\Omega(4^k)$.
- Each of the $O(n)$ locations can have that at most many different DFA states built from it, making $O(n2^{k^2})$ states total. Multiplying the number of states with the number of symbols, this is $O(n2^{k^2+\min(2k+1,n)})$ transitions to calculate.
- Each transition is calculated by considering each of the positions $i^{\#e}$ in the state, finding the first index $j > i$ in the symbol where the characteristic vector has a 1 at index j,

calculating the image set of the elementary transition of $i^{\#e}$, then taking the union of all of the sets, ignoring positions with a certain property. We can make this step a little faster by storing the characteristic vector as a balanced tree over the indices where the value is 1 and making that step $O(\log k)$. If we store each state as a balanced tree over the positions with a special comparison function that allows positions that are ignored by the paper's DFA construction to be marked as equal, we can union all of the sets in $O(k \log k)$ by joining the trees.

This huge preprocessing time means that in applications where searching for strings of a large upper bound on Levenshtein distance – about $\log WD$ – it may actually be more efficient to use dynamic programming. However, we likely never see this case. Spelling correction usually does not use $n \geq 4$, because doing so would wildly generate, and even $O(4^n 2^{n^2})$ of such small distances would require an extremely small dictionary for preprocessing a Levenshtein automaton to overtake dynamic programming on all the dictionary entries.

For an application where support for large n is needed for some reason, it's possible to preprocess only for the $O(n)$ states with one position, and then union all of the image sets for each position in a state during runtime for $\langle O(Wn^2 \log n 4^n), O(Dn \log n) \rangle$. In fact, if no preprocessing is done and all states and transitions are calculated at runtime, with the use of the binary search trees on the characteristic vectors and states, we can achieve $\langle O(1), O(Dn^2 \log n) \rangle$.

4 Conclusion

In this paper, we presented a basic review of the Levenshtein automata, compared the Levenshtein automaton approach with the naive one and analyzed the time complexity. We show by experimental results that Levenshtein automata is an effective model to use for the spell correction task.

The combination of a Levenshtein Automata with a Trie presents exceptional performance:

- In preprocessing phase, the compressed trie has comparable size with a hash table;
- In candidate generation phase, instead of like the naive approach which brute-force all possible operations, Levenshtein Automata just walks together with the trie, and take the shared transitions, which is much more efficient;
- In matching phase, the Automata just checks if the final state is accepted or not, without using a hash table to look up.

4.1 Future Work

Because the automata are fairly uniform across positions in a string, it would not be difficult to extend these structures to an application where one must find fuzzy matches of a string as it is typed without recomputing each step, adding one more to the length as it consumes each character. Furthermore, with the $\langle O(1), O(Dn^2 \log n) \rangle$ modified DFA method mentioned above, it may be possible to use Levenshtein automata to calculate maximum edit distance. The naive approach using maximum $n = W + D$ is worse than dynamic programming, but there may be an approach that averages to a better runtime. As mentioned previously, it would also be interesting to look more into building an automaton that handles transpositions and other types of errors that involve errors changing the order of characters.

4.2 Code Repository

The readers are welcome to see our code at the public Github repository: https://github.com/dengl11/CS166-Project.

References

[1] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[2] Richard C. Angell, George E. Freund, and Peter Willett. Automatic spelling correction using a trigram similarity measure. *Inf. Process. Manage.*, 19(4):255–261, 1983.

[3] Klaus U. Schulz and Stoyan Mihov. Fast string correction with levenshtein automata. *IJDAR*, 5(1):67–85, 2002.

[4] Li Deng. Levenshtein automata. `https://github.com/dengl11/CS166-Project`, 2018.

[5] Nick Johnson. Damn cool algorithms: Levenshtein automata. `http://blog.notdot.net/2010/07/Damn-Cool-Algorithms-Levenshtein-Automata`, 2010.