1 LevenshteinAutomaton

heylichen@qq.com

The source code is at https://github.com/heylichen/LevenshteinAutomaton.

Original content is from https://julesjacobs.com/2015/06/17/disqus-levenshtein-simple-and-fast.html. As a learning note I add some of my understandings.

1.1 Levenshtein distance

Given two character strings s1 and s2, the **edit distance** between them is the minimum number of edit operations required to transform s1 into s2.

Most commonly, the edit operations allowed for this purpose are:

- insert a character into a string;
- delete a character from a string and
- replace a character of a string by another character; for these operations, edit distance is sometimes known as **Levenshtein distance**.

For example, the edit distance between cat and dog is 3.

standard dynamic programming algorithm

It is well-known how to compute the (weighted) edit distance between two strings in time O(|s1| * |s2|), where |si| denotes the length of a string si. The idea is to use the dynamic programming algorithm in Figure 1.1, the (i, j) entry of the matrix will hold (after the algorithm is executed) the edit distance between the strings consisting of the first i characters of s1 and the first j characters of s2. 核心逻辑在第 8-10 行，从 3 种情况里选最小的。

```
EDITDISTANCE($s_1, s_2$)
 1  $int\ m[i,j] = 0$
 2  for $i \leftarrow 1$ to $|s_1|$
 3  do $m[i,0] = i$
 4  for $j \leftarrow 1$ to $|s_2|$
 5  do $m[0,j] = j$
 6  for $i \leftarrow 1$ to $|s_1|$
 7  do for $j \leftarrow 1$ to $|s_2|$
 8      do $m[i,j] = \min\{m[i-1,j-1] + \text{if } (s_1[i] = s_2[j]) \text{ then 0 else 1fi},$
 9                       $m[i-1,j] + 1,$
10                       $m[i,j-1] + 1\}$
11  return $m[|s_1|,|s_2|]$
```

Figure 1.1: Dynamic programming algorithm for computing the edit distance between strings s1 and s2.

Figure 1.2 shows an example Levenshtein distance computation of Figure 1.1. The typical cell [i, j] has four entries formatted as a 2 * 2 cell. The lower right entry in each cell is the min of the other three, corresponding to the main dynamic programming step in Figure 1.1. The cells with numbers in italics depict the path by which we determine the Levenshtein distance.

			f		a		s		t	
		0	**1**	**1**	**2**	**2**	**3**	**3**	**4**	**4**
c		**1**	*1*	*2*	**2**	3	**3**	4	**4**	5
		1	*2*	*1*	**2**	**2**	**3**	**3**	**4**	**4**
a		**2**	**2**	**2**	*1*	*3*	*3*	*4*	4	5
		2	3	**2**	*3*	*1*	*2*	*2*	**3**	**3**
t		**3**	**3**	**3**	3	**2**	*2*	*3*	*2*	*4*
		3	4	**3**	4	**2**	*3*	*2*	*3*	*2*
s		**4**	**4**	**4**	4	**3**	**2**	3	*3*	*3*
		4	5	**4**	5	**3**	4	**2**	*3*	*3*

Figure 1.2: Example Levenshtein distance computation. The 2 * 2 cell in the [i, j] entry of the table shows the three numbers whose minimum yields the fourth. The cells in italics determine the edit distance in this example.

Iterative with two matrix rows

```
function LevenshteinDistance(char s[0..m-1], char t[0..n-1]):
    // create two work vectors of integer distances
    declare int v0[n + 1]
    declare int v1[n + 1]

    // initialize v0 (the previous row of distances)
    // this row is A[0][i]: edit distance from an empty s to t;
    // that distance is the number of characters to append to  s to make t.
    for i from 0 to n:
        v0[i] = i

    for i from 0 to m - 1:
        // calculate v1 (current row distances) from the previous row v0

        // first element of v1 is A[i + 1][0]
        //   edit distance is delete (i + 1) chars from s to match empty t
        v1[0] = i + 1

        // use formula to fill in the rest of the row
        for j from 0 to n - 1:
            // calculating costs for A[i + 1][j + 1]
            deletionCost := v0[j + 1] + 1
            insertionCost := v1[j] + 1
            if s[i] = t[j]:
                substitutionCost := v0[j]
            else:
                substitutionCost := v0[j] + 1

            v1[j + 1] := minimum(deletionCost, insertionCost, substitutionCost)

        // copy v1 (current row) to v0 (previous row) for next iteration
        // since data in v1 is always invalidated, a swap without copy could be more efficient
        swap v0 with v1
    // after the last swap, the results of v1 are now in v0
    return v0[n]
```

In the second for loop, after each iteration, v0[j] stores the edit distance between the strings consisting of the first i+1 characters of s and the first j characters of t. Each array v0 correspond to a "state" of a prefix of string s, the state store the edit distances between current prefix of string s and prefixes of string t.

1.2 What's a Levenshtein automaton good for anyway?

When you're doing full text search your users may misspell the search terms. If there are no search results for the misspelled search term you may want to automatically correct the spelling and give search results for the corrected search term, like Google does with its **"Did you mean X?"** . If somebody searches for "bannana" you want to give them results for "banana" . The standard way of measuring similarity between two strings is the Levenshtein distance, The Levenshtein distance between "bannana" and "banana" is 1 because there's 1 'n' inserted (or deleted, depending on which way around you view it). When a search query gives no results we could instead search for all queries with a Levenshtein distance of 1 to the query. So if somebody types "bannana" we would generate all words with Levenshtein distance 1 to it, and search for those. If that still doesn't yield any results, we would generate all words with distance 2, and so forth.

The problem is that the number of different search terms is huge. We can insert, delete and substitute any letter, so that's "aanana" , "canana" , "danana" , etcetera, just for the first letter. This is especially impossible if you are using the full unicode alphabet. Another method, which is the method that Lucene used previously (before lucene 4.0), is to **simply scan the whole full text** index for any word that is within a given Levenshtein distance to the query. This is very costly: the index could have millions of words in it so we don't want to do a full search over the whole index.

A better approach is to exploit the structure of the index. It's usually a trie, or a string B-tree, or even a sorted list of strings. A Levenshtein automaton can be used to efficiently eliminate whole parts of the index from the search. Suppose the index is represented as a trie:

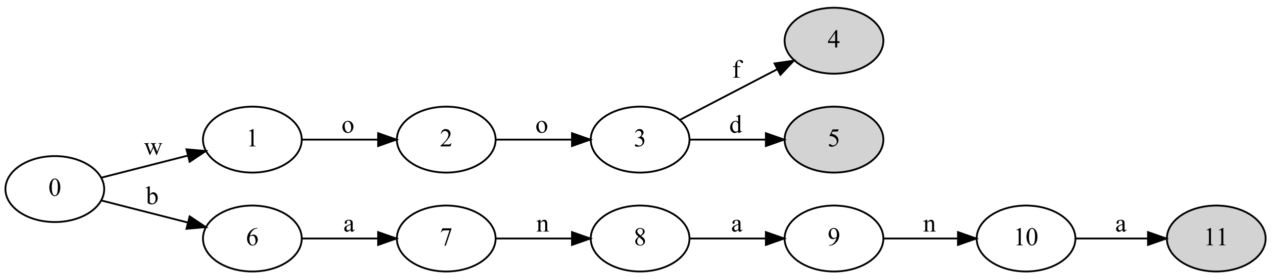

Figure 1.3: This trie contains the words "woof" , "wood" and "banana" .

When we search this trie for words within Levenshtein distance 1 of "bannana" the whole left branch is irrelevant. The search algorithm could find that out like this. We start at the root node, and go left via "w" . We continue searching because it's possible that we will find "wannana" when we go further. We continue down to "o" , so now we are at node 2 with a prefix of "wo" . We can stop searching because no word starting with "wo" can be within Levenshtein distance of 1 of "bannana" . So we backtrack and continue down the right branch starting with "b" . This time we go all the way down to node 11 and find "banana" , which is a match. What have we gained? Instead of searching the entire trie we could **prune some subtries from the search**. In this example that's not a big win, but in a real index we can have millions of words. The subtree under "wo" could be huge, and we don't need to search any of it.

This is where Levenshtein automata come in. Given a word and a maximum edit distance we can contruct a Levenshtein automaton. We feed this automaton letters one by one, and then the automaton can tell us whether we need to continue searching or not. For this example we create a Levenshtein automaton for "banana" and N=1. We start at the root node and feed it "w" . Can that still match? The automaton says yes. We continue down and feed it "o" . Can that still match? The automaton says no, so we backtrack to the root. We also backtrack the automaton state! We feed that automaton "b" and ask whether it can still match? It says yes. We continue down the right path feeding it letters until we've arrived at node 11. Then we ask "does this match?" . The automaton says yes, and we've found our search result. Here's the interface for a Levenshtein automaton in Java:

```java
// T is type of State, for example List<Integer>
public interface LevenshteinAutomata<T> {
    T getStartState();
    T step(T state, char ch);

    boolean isMatch(T state);

    boolean canMatch(T state);

    Iterable<Character> transitions(T state);
}
```

We have a constructor that takes the query string and a max edit distance n. We have a method getStartState that returns the start state. We have a method step(state, c) that returns the next state given a state and a character. We have isMatch(state) that tells us whether a state is matching, and we have canMatch(state) which tells us whether a state can become matching if we input more characters. canMatch is what we will use to prune the search tree. Here's an example:

```
public class RowLevenshteinAutomataTest {

  @Test
  public void testCanMatch() {
    LevenshteinAutomata<List<Integer>> la = new RowLevenshteinAutomata("banana", 1);
    List<Integer> state = la.getStartState();

    state = la.step(state, 'w');
    // True, "w" can match "bannana" with distance 1
    System.out.println(la.canMatch(state));
    state = la.step(state, 'o');
    //False, "wo" can't match "bannana" with distance 1
    System.out.println(la.canMatch(state));
  }
}
```

State can be encapsulated in automata class. But State acutally has different lifecycle as other info in Levenshtein-Automata, such as string and maxEdits. So State is declared as a mehtod parameter in LevenshteinAutomata interface. This will be clear in the building DFA section.

1.3 The O(length of string) version

```
public class RowLevenshteinAutomata implements LevenshteinAutomata<List<Integer>> {
  private final List<Integer> startState;
  private final String string;
  private final int maxEdits;

  @Override
  public List<Integer> getStartState() {
    return startState;
  }

  public RowLevenshteinAutomata(String string, int maxEdits) {
    this.string = string;
    List<Integer> localState = new ArrayList<>(string.length() + 1);
    localState.add(0);
    for (int i = 0; i < string.length(); i++) {
      localState.add(i + 1);
    }
    this.startState = localState;
    this.maxEdits = maxEdits;
  }

  @Override
  public List<Integer> step(List<Integer> state, char ch) {
    List<Integer> newState = new ArrayList<>(string.length() + 1);
    newState.add(state.get(0) + 1);
    for (int i = 0; i < string.length(); i++) {
      int cost = ch == string.charAt(i) ? 0 : 1;
      // dist = min(state.get(i) + cost, state.get(i + 1) + 1, newState.get(i) + 1)
      int dist = Math.min(
          state.get(i) + cost,
          state.get(i + 1) + 1
      );
      dist = Math.min(dist, newState.get(i) + 1);
      newState.add(dist);
    }
    return newState;
  }

  @Override
  public boolean isMatch(List<Integer> state) {
    return state.get(state.size() - 1) <= maxEdits;
  }

  @Override
  public boolean canMatch(List<Integer> state) {
    for (Integer integer : state) {
      if (integer <= maxEdits) {
        return true;
      }
    }
    return false;
  }

  @Override //Hide implementation for now
  public List<Character> transitions(List<Integer> state);
}
```

This is a row based implementation. Excecution trace is as follows. Each time calling step with a char from input string makes a state transition. In each row, say the state is state i, the state array[j] stores the edit distance between the string of first i characters of input string (shown in "input string" column) and first j characters of the automaton's string. The first 0 characters of any string is empty.

inputString = xoof this.string = woof this.maxEdits = 1

input string	input char	state	0	1 w	2 wo	3 woo	4 woof
		state 0	0	1	2	3	4
x	x	state 1	1	1	2	3	4
xo	o	state 2	2	2	1	2	3
xoo	o	state 3	3	3	2	2	2
xoof	f	state 4	4	4	3	2	1

Figure 1.4: init RowLevenshteinAutomata with string="woof", maxEdits=1, step all chars in "xoof"

1.3.1 Converting to a DFA

Lucene builds a DFA out of the Levenshtein automaton. We don't need to do that since we can search the index directly with the step() based automaton, but if we wanted to, can we convert to a DFA? A DFA has a finite number of states, but our automaton has an infinite number states. Each time we step the numbers in the row will increase. The thing is that many of those states are equivalent. For example if n = 2 then these states are all equivalent:

```
[3, 2, 1, 2, 3, 4, 5]
[500, 2, 1, 2, 500, 500, 500]
[3, 2, 1, 2, 3, 3, 3]
```

Once a number goes above n it doesn't matter whether it's 3, 5 or 500. It can never cause a match. So instead of increasing those numbers indefinitely we might as well keep them on 3. So we can represent any number above n by n+1. Step method is modified as follows. Before returned, every state value larger than maxEdits is normalized to maxEdits + 1.

```
  public List<Integer> step(List<Integer> state, char ch) {
    List<Integer> newState = new ArrayList<>(string.length() + 1);
    newState.add(state.get(0) + 1);

    for (int i = 0; i < string.length(); i++) {
      int cost = ch == string.charAt(i) ? 0 : 1;
      // dist = min(state.get(i) + cost, state.get(i + 1) + 1, newState.get(i) + 1)
      int dist = Math.min(
          state.get(i) + cost,
          state.get(i + 1) + 1
      );
      dist = Math.min(dist, newState.get(i) + 1);
      newState.add(dist);
    }
    //to make state value finite
    List<Integer> normalizedNewState  = new ArrayList<>(newState.size());
    // MAX_EDITS_PLUS_1 = maxEdits+1
    for (Integer integer : newState) {
      integer = integer > maxEdits ? MAX_EDITS_PLUS_1 : integer;
      normalizedNewState.add(integer);
    }
    return normalizedNewState;
  }
```

Now the number of states is finite! (Because the distance number in state array is finite, and the state array is of finite length. So the number of distinct state arrays is finite.) We can recursively explore the automaton states while keeping track of states that we've already seen and not visiting them again.

There's one more thing we need however: the automaton needs to tell us **which letters to try from a given state**. We only need to try the letters that actually appear in the relevant positions in the query string. If the query string is "banana" we don't need to try "x" , "y" , and "z" separately, since they all have the same result. Also, if an entry in the state is already 3, we don't need to try the corresponding letter in the string, since it can never cause a match anyway. Here's the code to compute the letters to try:

```
public class RowLevenshteinAutomata implements LevenshteinAutomata<List<Integer>> {
    @Override
    public List<Character> transitions(List<Integer> state) {
      List<Character> result = new ArrayList<>(string.length());
      for (int i = 0; i < string.length(); i++) {
        if (state.get(i) <= maxEdits) {
          result.add(string.charAt(i));
        }
      }
      return result;
    }
}
```

Understanding tansitions method

say the state correspond to curent input string inputStringK, is a string of length K. In the for loop, for each i, if state.get(i)<=maxEdits, then the edit distance between inputStringK and first i characters of the automaton's string (which is this.string.substring(0,i)). Then if the next character to try equals the next character of the automaton's string (which is string.charAt(i)), we can get a match in the end. Otherwise this is no chance to get a match, so no need to try other characters.

for a state Sk, let the characters returned by transitions be transChars, and all other chars in the alphabet be *. We prove that from state Sk, for each character in *, the automaton will transition to the same state. So we can represent all characters not in transChars as one character category as *.

Assume automaton's string is "woof", maxEdits = 1. In state Sk, assume the Matrix row is M[k]. M[K+1][0] = M[k][0]+1 is fixed for any transition character, so is M[k].

```
newState returned by step method is M[k+1]
for j >=1 and j < 4,  M[k+1][j] = min(
                                      M[k][j-1]+cost,
                                      M[k+1][j-1]+1,
                                      M[k][j] +1
                                      )
cost = stepChar == "woof".charAt(j-1) ? 0 : 1;
```

suppose the next input char is stepChar. if stepChar is in transChars, then state.get(j-1) <= maxEdits and stepChar == string.charAt(j-1). if stepChar is not in transChars, it's in *. Then state.get(j-1) > maxEdits or stepChar != string.charAt(j-1) for j>=1 and j<4.

if Sk.get(j-1) > maxEdits, then M[k+1][j] = maxEdits + 1, no matter cost is 0 or 1, M[k][j-1]+cost is maxEdits + 1. whether Sk.get(j-1) > maxEdits is true depends only on current state Sk and is fixed for any input character. In this case M[k][j-1]+cost is fixed for any character in *.

if stepChar != string.charAt(j-1) then cost is 1, no matter what the specific character is. In this case M[k][j-1]+cost is still the same for any character in *, j>=1 and j<4.

In both cases, M[k][j-1]+cost is the same for any character in *, j>=1 and j<4.

Notice M[k][j] +1 depend only on current state, is the same for any transition character.

M[k+1][j-1]+1 depend on previous calculated M[k+1] array. If for every transition character in *, and for each j (j>=1 and j<4), M[k][j-1]+cost and M[k][j] +1 is the same, then M[k+1][j-1]+1 is also the same.

So M[k+1][j] is the same for every transition character in *, and for each j (j>=1 and j<4).

So for every transition character (stepChar) in *, step(state , stepChar) return the same new state M[k+1]. They are in the same category, we only need one symbol to represent them all: *.

Now that we have this we can recursively enumerate all the states and store them in a DFA transition table:

```java
public abstract class AbstractLevenshteinAutomataDFA<T> {

  private final LevenshteinAutomata<T> automaton;
  // for generating state id
  private int idCounter = 0;
  // for remember state id that have been encountered
  private Map<String, Integer> stateMap;
  // result DFA states
  private List<Transition> transitions;
  private Set<Integer> matchedStateIds;

  // for match method, computed from transitions
  private Map<Integer, Map<Character, Integer>> transitionMap;
  private boolean matchEmptyInput;

  protected static final char SEP = ',';
  protected static final Character OTHER = '*';

  public AbstractLevenshteinAutomataDFA(LevenshteinAutomata<T> automaton) {
    this.automaton = automaton;
    init();
  }

  private void init() {
    matchedStateIds = new HashSet<>();
    stateMap = new HashMap<>();
    transitions = new ArrayList<>();
    explore(automaton.getStartState());
    transitions.sort(Comparator.comparing(Transition::getFromStateId)
        .thenComparing(Transition::getToStateId).thenComparing(Transition::getCh));
    initForMatch();
  }

  private Integer explore(T state) {
    String stateKey = genKey(state);
    Integer existedStateId = stateMap.get(stateKey);
    if (existedStateId != null) { //encountered before, no need to explore.
      return existedStateId;
    }
    int currentStateId = idCounter; //assign state id and remember it
    idCounter++;
    stateMap.put(stateKey, currentStateId);

    if (automaton.isMatch(state)) { // check if it is in final state set
      matchedStateIds.add(currentStateId);
    }

    //chars contains characters
    Collection<Character> chars = automaton.transitions(state);
    // OTHER=* stands for any other characters
    chars.add(OTHER);

    for (Character ch : chars) {
      T newState = automaton.step(state, ch);
      Integer toStateId = explore(newState);
      transitions.add(new Transition(currentStateId, ch, toStateId));
    }
    return currentStateId;
  }

  /** gen key for a state, to check if a state has been encountered in stateMap*/

 protected abstract String genKey(T state);

 private static class Transition {
    private final Integer fromStateId;
    private final Character ch;
    private final Integer toStateId;

    public Transition(Integer fromStateId, Character ch, Integer toStateId) {
      this.fromStateId = fromStateId;
      this.ch = ch;
      this.toStateId = toStateId;
    }
    // getter methods
  }
}
```

Notice the explore method called with start state during construction. To make the DFA, we need to explore all possible states from start state.

Here's the resulting DFA for "woof" and n = 1:

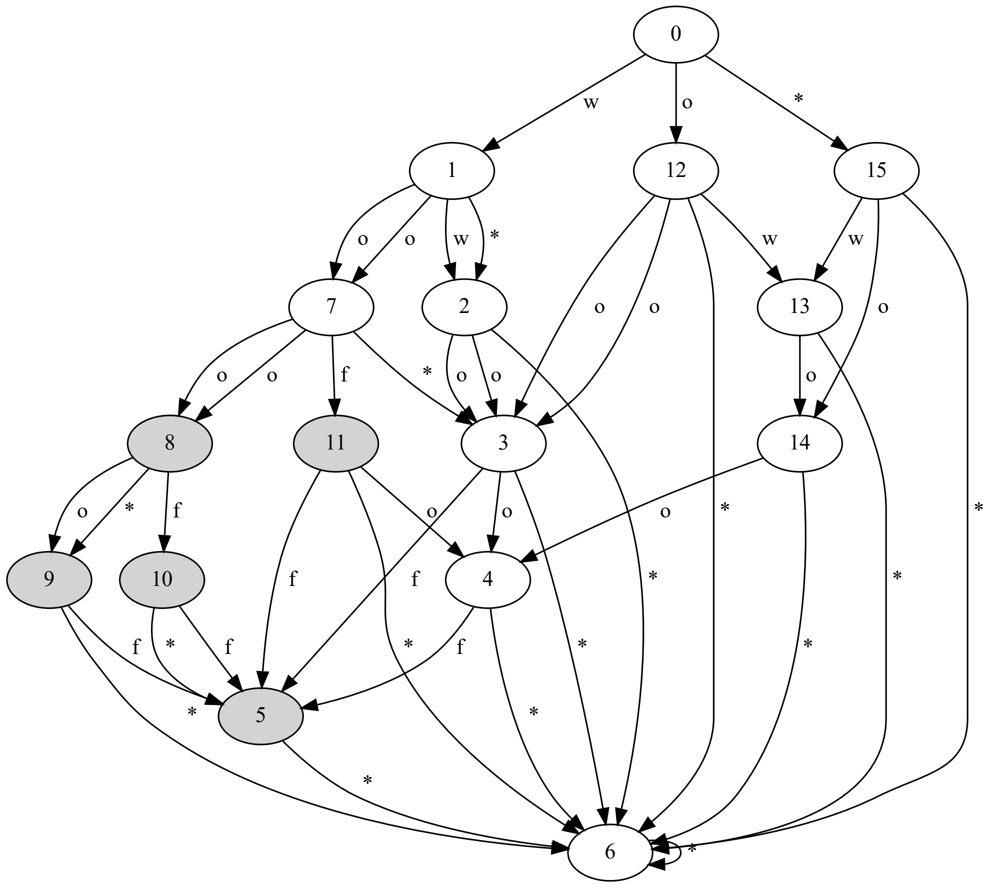

Figure 1.5: DFA for woof and maxEdits = 1

The shaded nodes are the accepting states. In this case the DFA has the minimum possible number of states, but just like for the method in the paper this is not guaranteed to happen.

So how much time does it take to create this DFA? If there are k nodes in the DFA then we need to step O(k) times. Each step takes O(length of the query string) time, because we compute a matrix row of that size. So how many nodes are in the DFA? Let' s go back to what the Levenshtein matrix means. M[i,j] := levenshtein distance between A[0..i] and B[0..j]. The Levenshtein distance between a string of length i and a string of length j is at least |i - j|. So if we are in row number i, then only the entries i-n ... i+n are <= n. This is 2n + 1 entries. Let' s see what this means for Lucene' s use case, which only supports n = 1,2. For n = 1 each row has at maximum 3 entries which are <= 1 and for n = 2 each row has at maximum 5 entries which are <= 2. This means that for fixed n, there are only a constant number of entries in each row that we care about, so for fixed n the number of states in the DFA is linear in the size of the query string.

1.4 The O(max edit distance) version

So if we are building an automaton with n = 2 for a query string of size 1000, then each state is a matrix row of size 1001, but it will contain at most 5 entries which aren' t equal to 3! Why are we still computing the other 996 entries which are all equal to 3?! The standard solution to such a problem is to use a sparse vector. Instead of storing all entries, we only store the ones which are not equal to 3. Instead of this:

```
state = [3,3,3,3,3,3,3,3,3,3,3,3,2,1,2,3,3,3,3,3,3]
```

we store

```
state = [
    {
      "stateIndex": 11,
      "distance": 2
    },
    {
      "stateIndex": 12,
      "distance": 1
    },
    {
      "stateIndex": 13,
      "distance": 2
    }
  ]
```

Only include distance <= n in states. Notice if query string is length M, then state is array of length M+1. state[i] stores the edit distance between input string correspond to current state and first i characters of query string.

Remember that there are at most 2n + 1 entries not equal to n + 1, so these arrays will have a size that is independent of size of the string. Computing the step(state,c) becomes a bit more difficult with this representation, but not much more difficult:

```
public class SparseLevenshteinAutomata implements LevenshteinAutomata<List<IndexValue>> {
  private final String string;
  private final int maxEdits;

  public SparseLevenshteinAutomata(String string, int maxEdits) {
    this.string = string;
    this.maxEdits = maxEdits;
  }

  @Override
  public List<IndexValue> getStartState() {
    int initCount = maxEdits + 1;
    List<IndexValue> indexValues = new ArrayList<>(initCount);
    for (int i = 0; i < initCount; i++) {
      indexValues.add(new IndexValue(i, i));
    }
    return indexValues;
  }

  @Override
  public List<IndexValue> step(List<IndexValue> state, char ch) {
    IndexValue oldFirst = !state.isEmpty() ? state.get(0) : null;
    List<IndexValue> newStates = new ArrayList<>(maxEdits * 2 + 1);
    if (oldFirst != null && oldFirst.getStateIndex() == 0 && oldFirst.getDistanceValue() < maxEdits) {
      newStates.add(new IndexValue(0, oldFirst.getDistanceValue() + 1));
    }

    for (int i = 0; i < state.size(); i++) {
      IndexValue current = state.get(i);
      // currentStateIndex is the prefix len of this.string
      // currentStateIndex - 1 is the char index of this.string that has stepped in current state
      int currentStateIndex = current.getStateIndex();
      if (currentStateIndex == string.length()) {
        //this.string exhausted, stop
        break;
      }

      int cost = ch == string.charAt(currentStateIndex) ? 0 : 1;
      int value = current.getDistanceValue() + cost;

      IndexValue newPrevious = newStates.isEmpty() ? null : newStates.get(newStates.size() - 1);
      if (newPrevious != null && newPrevious.getStateIndex() == currentStateIndex) {
        value = Math.min(value, newPrevious.getDistanceValue() + 1);
      }
      IndexValue oldNext = i + 1 < state.size() ? state.get(i + 1) : null;
      if (oldNext != null && oldNext.getStateIndex() == currentStateIndex + 1) {
        value = Math.min(value, oldNext.getDistanceValue() + 1);
      }

      if (value <= maxEdits) {
        newStates.add(new IndexValue(currentStateIndex + 1, value));
      }
    }
    return newStates;
  }

  @Override
  public boolean isMatch(List<IndexValue> state) {
    return !state.isEmpty() && state.get(state.size() - 1).getStateIndex() == string.length();
  }

  @Override
  public boolean canMatch(List<IndexValue> state) {
    return !state.isEmpty();
  }

  @Override
  public Set<Character> transitions(List<IndexValue> state) {
    Set<Character> result = new LinkedHashSet<>();
    for (IndexValue indexValue : state) {
      int prefixLen = indexValue.getStateIndex();
      if (prefixLen < string.length()) {
        result.add(string.charAt(prefixLen));
      }
    }
    return result;
  }

}
```

Now each step takes O(max edit distance) time instead of O(length of string). So for Lucene which has max edit distance of 1 or 2, this is constant time. We can now build the DFA in linear time.