Chapter 3

Approximate string indexing: comments to slideshow

Alexander Vahitov

Using simple mathematical arguments the matching probabilities in the suffix tree are bound and by a clever division of the search pattern sub-linear time is achieved.

The report is based on the article of G. Navarro and R. Baeza-Yates 'A Hybrid Indexing Method For Approximate String Matching'

3.1 Introduction

First Section 3.2 is about the task of the algorithm with some simple examples. Next Section 3.3 will tell you some basic ideas and algortihms used in the main resulting algorithm of the report, which is presented in Section 3.4. Then comes Section 3.5 where you will find some ideas to prove the average-case complexity of the algorithm (full complexity analysis can be found in the issue of G. Navarro and R. Baeza-Yates). The last part of my report is Section 3.6 with conclusions and future directions of scientific work in this field.

3.2 Our Task

We have a long text T and a short pattern (P). Our task is to find substrings from T which match our pattern approximately. Approximate matching means that we can accept some errors during the searching process (you can imagine that these are transportation errors). These errors are differences between the pattern and the founded text piece (*ocuurence*). We define 3 kinds of differences between strings: *insertion*, *replacement* and *deletion*.

If we have a pattern abc and a text $adbc$, then there are some obvious examples of differences between this strings :

$adc = a + \underline{d} + b + c$ - *insertion*

$abd = a + b + \underline{c \to d}$ - *replacement*

$ab = a + b + \underline{c \to \varnothing}$ - *deletion*

We call the minimum number of such changes needed to transform one string to another as *edit distance* (abbreviated *ed*) between the strings. For example, all the mentioned above strings have the edit distance equal to 1 (because we needed for transformation only 1 change).
If S can be transformed to S' with x changes, then S' can be transformed to S also with x changes (we remove deletions with insertions, and vice versa).
The resulting algorithm, which will be presented later, has to solve the approximate searching problem. There are some different approaches to this problem, and at first I will tell you some of them. The first variant is to build a suffix trie for the text and search in $O(n)$ time for the occurences. We can descend by the branches of the trie till the level where we can understand that this branch does not contain an occurence. If we use *suffix array* structure, we will free much memory (tis problem arises because suffix trees have very big memory requirements). This method of search is called Depth-First Search. We can generate a set of *viable prefixes* (possible prefixes for the pattern occurence) and search for them in our trie.
The second way is to build an on-line filtering algorithm. It can use some sort of index (for example, many algorithms are based on storing text *q-grams* - pieces of the text with length equal to some number q).But it is obvious that the number of errors in such algorithms is strongly bounded, nd it can be incompatible with many practical situations.
There is another outstanding algorithm by Myers based on the mixture of the two approaches. It uses q-grams. It divides a pattern in such pieces that their length is less than $q - k$ (where k is error-level, error number divided by pattern length). Then it generates for each text piece all the strings that can appear from the pattern when some errors are done. All this strings are searched then in the q-gram set, and the last step is merging founded piece occurences and searching for the occurence of the whole pattern.
Our algorithm at first will divide our pattern into some pieces. Then it will use approximate search to find the occurences of this pieces in the text, and then verify whether the occurence of some pattern piece can be continued to the occurence of the whole pattern.

3.3 Basic Ideas and Algoritms

3.3.1 Lemma: dividing the pattern

Dividing the pattern is useful for our algorithm. If we have two strings A and B,their edit distance $ed \leq k$ and we divide A intoj substrings, then at least one of the substrings appears in B with at most $\lfloor k/j \rfloor$ errors. This is obvious because we have to change A k times to transform it to j. Each change is applied to one of the substrings. The average number of changes per substring is $\frac{k}{j}$. So it is easy to see that there is at least one of A_i that has less than or equal to $\frac{k}{j}$ changes.

Example:

$$ed('he\_likes', 'they\_like') = 3 = k;$$
$$A_1 = ' he\_', A_2 = ' likes' \Rightarrow j = 2;$$
$$ed('he\_', 'they\_') = 2; ed('likes', 'like') = 1 = \lfloor \frac{k}{j} \rfloor.$$

3.3.2 Computing edit distance

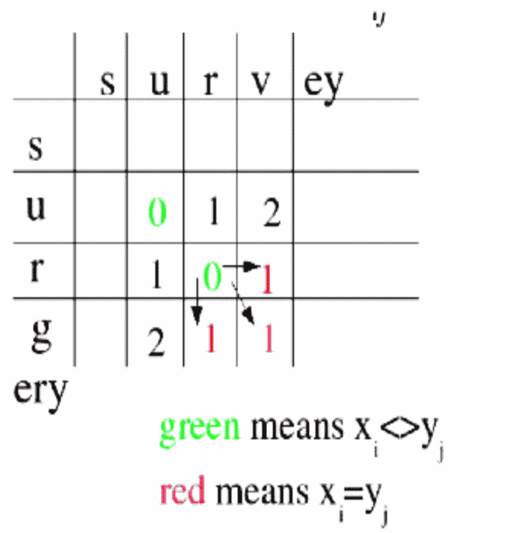
Computing edit distance is also useful for approximate string matching. There is a classical dynamic programming algorithm solving this problem. We have 2 strings: x and y with characters x_i and y_j. Let's consider C_{ij} as edit distance between $x_1..x_i$

and $y_1..y_j$. Let's fill the matrix of C_{ij} with such algorithm: $C_{i,0} = i$ because you need i insertions to the empty string to change it to $x_1..x_i$. And also $C_{0,j} = j$ by the same cause. $C_{ij} = C_{i-1,j-1}$ if $x_i = y_j$. Another case is when $x_i \neq y_j$. You need one deletion of the character y_j from the string $y_1...y_j$ to make it matching $x_1...x_i$ with $C_{i,j-1}$ errors. Also you may delete x_i to make C_{ij} equal to $C_{i-1,j+1}$. And you can replace x_i with y_j to make C_{ij} equal to $C_{i-1,j-1}$. So if $x_i \neq y_j$ then $ed(x_1..x_i, y_1..y_j) = 1 + min\{C_{i-1,j}; C_{i,j-1}; C_{i-1,j-1}\}$.

An example shows how does the algorithm work with $'survey'$ and $'surgery'$ strings. The cases when $x_i = y_j$ are marked with green, and other cases - with red. There are arrows from the minimal matrix element (left, upper or diagonal) which summed with 1 gives us C_{ij} to make it matching $x_1..x_i$ with $C_{i,j-1}$ errors.

$$x = x_1 x_2 \ldots x_m; y = y_1 y_2 \ldots y_n; x_p, y_q \in \Sigma$$
$$C_{ij} = ed(x_1 \ldots x_i, y_1 \ldots y_j);$$
$$(C) \text{ is a matrix filled with } C_{ij}$$
$$C_{0,j} = j; C_{i,0} = i;$$

$$C_{i,j} = \begin{cases} C_{i-1,j-1} & x_i = y_j \\ 1 + \min\{C_{i-1,j}, C_{i-1,j-1}, Ci, j-1\} & \text{else;} \end{cases}$$



Now let's consider y as the text and x as the pattern. Let's construct the algorithm to search the substrings from the text, approximately matching the pattern. The only different between this algorithm and previous one is that we initialize $C_{0,j}$ with 0 instead of j because the pattern matching process can start from every text position.

These are fullfilled matrices by the algorithm that simply computed the edit distance and the algorithm that searched the pattern in the text.

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	0	1	2	3	4	5	6
u	2	1	0	1	2	3	4	5
r	3	2	1	0	1	2	3	4
v	4	3	2	1	1	2	3	4
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	**2**

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

3.3.3 NFA construction

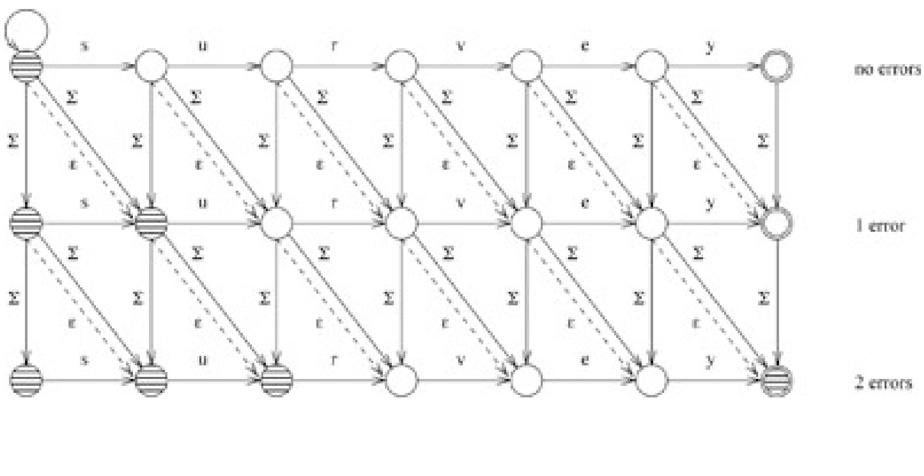
Here is presented *Nondeterministic Finite Automaton*. We will use it to search for approximate matches of pattern P in the text T. The initial state of the automaton corresponds to 0 errors in matching process and to the first character of the pattern. The automaton will go by pattern and text characters simultaneously. When it reaches the last pattern character, it means that we have found an occurence. It is presented lower in the picture as a table. In columns pattern characters are written, and each row is used to represent errors in matching. We search for pattern occurence accepting some errors, and transition between rows means accepting one error.
There are 4 kinds of transitions between states (we make changes with pattern and the text is remaining the same):

- if current text and pattern characters are the same;
- if current pattern character is replaced with the text one;
- if current text character is inserted to the pattern;
- if current pattern character is deleted.

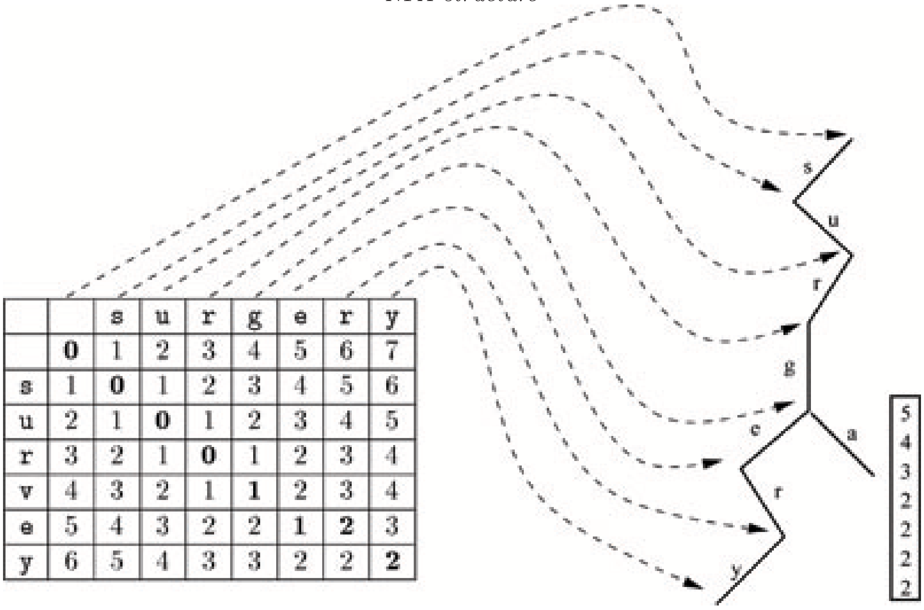
Look at the illustration of the automaton which searches the text for approximate matching the pattern *'survey'* with at most 2 errors. The rows correspond to the errors which are already made, and the columns correspond to the characters of the pattern already reached by the automaton.
The transitions are:

- horizontal, *if current text and pattern characters are the same*;
- solid diagonal, *if current pattern character is replaced with the text one*;
- vertical, *if current text character is inserted to the pattern*;
- dashed diagonal, *if current pattern character is deleted*.

The automaton finish states are the right ones.



NFA structure



		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

How NFA searches 'surga' in 'surgery'

3.3.4 Depth-First Search technique

And the last is the algorithm of the depth-first search. Here we define the $U_k(P)$ which is a set of the strings, matching to P with at most k errors. It is possible to search this string in the text, but the complexity of this search is quite large. We can use a suffix tree and search in it the strings from $U_k^t(P)$. These are the neighborhood elements which are not prefixes of other neighborhood elements.

3.4 Main algorithm

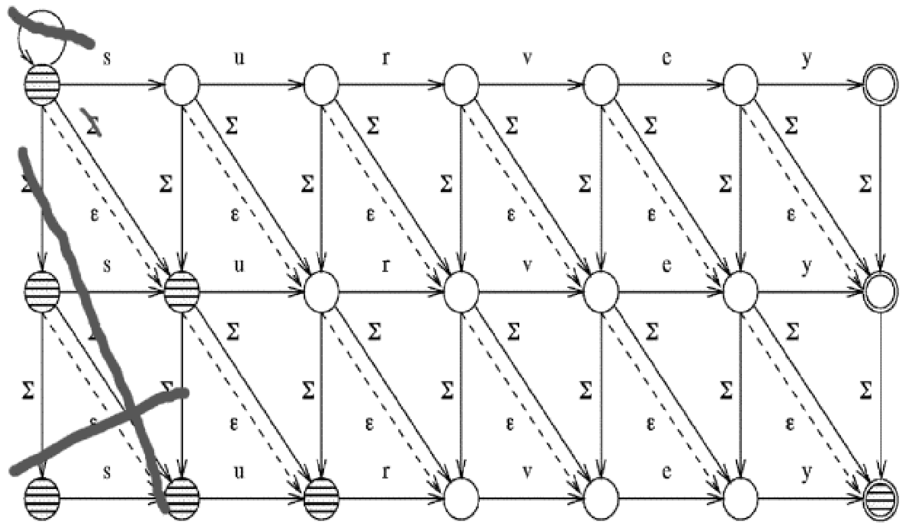
Our algorithm, searching in the suffix tree, has to start from the root, consider some string x incrementally,determine when $ed(x, P) \leq k$ and determine when adding of any character makes the ed greater than k.

In the picture you can see how the algorithm works with the input of the suffixes from the suffix tree. It fills the matrix like in the example above and analyzes it's elements.The algorithm increments x and updates the last column in O(m) time. When the element in the last row (last column element) is $\leq k$, the match is detected. Otherwise, if all the values in the column are $\geq k$, the match cannot be detected.

This is the illustration of the algorithm working in the suffix tree with 2 suffixes, matching the pattern $'surgery'$. It is shown the column of a matrix for the suffix $'surga'$.

The cost of the suffix tree search is exponential in m and k, so it's better to perform j searches of patterns of length $[\frac{m}{j}]$ and $\frac{k}{j}$ errors (remember the lemma about dividing!). That's why we divide patterns. So, we divide our pattern into j pieces and search them using the above algorithm. Then, for each match found ending at text position i we check the text area $[i - m - k..i + m + k]$. But the larger j, the more text positions need to be verified, and the optimal j will be found soon.

Now we have to adapt our NFA for searching in the suffix tree. At first, we'll search for matching from the beginning ot the suffix, so we don't need *initial self-loop*. Second, we don't need *initial insertions* to the pattern - because if te suffix matches with such insertions, we will find the suffix matching without these insertions. So, we remove the down-left triangle of the automaton, under diagonal. And at last, we can start match with $k+1$ first characters of the pattern (because we removed initial insertions, only initial deletions remain, initial replacement is the same as initial deletion).



This is the illustration to the changes of our NFA from the previous example.

3.4.1 Suffix Arrays

Here I can say something about using suffix arrays instead of suffix trees. Suffix arrays have less space requirements, but the time complexity of the search in the case of suffix array should be multiplied by $\log n$. Suffix array replaces nodes with intervals and traversing to the node is going to the interval. If there is a node and it's children, then the node interval contains children intervals. More information on suffix arrays was in Olga Sergeeva's report.

3.5 Complexity Analysis

Let's analyse the algorithm to determine it's complexity and the best variant of partitioning the pattern. Now we'll find the average number of nodes at level l. If we are working with random text, then the number of suffixes in level l is σ^l, and for small l the number of suffixes longer than l is nearly n. In the probability model of n balls thrown into σ^l urns we find that the average number of filled urnes is $\theta(\min\{\sigma^l, n\})$. If $l \leq m'$, at least $l-k$ text characters must match the pattern, and if $l > m'$, at least $m'-k$ pattern characters must match the text. There is no difference, which exactly is the length of the pattern prefix. So, we sum all the probabilities for different pattern prefix lengths:

$$\sum_{m'=l-k}^{l} \frac{1}{\sigma^{l-k}} C_l^{l-k} C_{m'}^{l-k} + \sum_{m'=l+1}^{l+k} \frac{1}{\sigma^{m'-k}} C_{m'-k}^{l} C_{m'}^{m'-k}$$

In the first sum the largest term is first one: $\frac{1}{\sigma^{l-k}}C_l^k$, and we can bound the whole sum with $(l-k)\frac{1}{\sigma^{l-k}}C_l^k$. By Stirling's approximation we have

$$C_l^k = \left(\frac{e^l\sqrt{2\pi l}}{k^k(l-k)^{l-k}\sqrt{2\pi k}\sqrt{2\pi(l-k)}}\right)^2 \left(1+O(\frac{1}{l})\right)$$

And the whole first sum is $(l-k)\gamma(\beta)^l O(\frac{1}{l})$, where

$$\gamma(\beta) = \frac{1}{\sigma^{1-x}x^{2x}(1-x)^{2(1-x)}}$$

Here you see that $(l-k)\gamma(\beta)^l O(\frac{1}{l}) = O(\gamma(\beta)^l)$. The first sum exponentially decreases when $\gamma(\beta) < 1$, it means that:

$$\sigma > \left(\frac{1}{\beta^{2\beta}(1-\beta)^{2(1-\beta)}}\right)^{\frac{1}{1-\beta}} = \frac{1}{\beta^{\frac{2\beta}{1-\beta}}(1-\beta)^2} > \frac{e^2}{(1-\beta)^2} \Leftrightarrow \beta < 1 - \frac{e}{\sqrt{\sigma}},$$

because $e^{-1} < \beta^{\frac{\beta}{1-\beta}}$ if $\beta \in [0,1]$. The second summation can be also bounded with $O(\gamma(\beta)^l)$, and the probability of processing a given node at depth l is $O(\gamma(\beta)^l)$. In practice, e should be replaced by $c = 1.09$ (founded experimentally) because we have found only upper bound for the probability, but not the exact upper bound.
Using the formulas bounding the probability of matching, let's consider that in levels

$$l \leq L(k) = \frac{k}{1-\frac{c}{\sqrt{\sigma}}} = O(k)$$

all the nodes are visited, and in levels $l > L(k)$ nodes are visited with probability $O(\gamma(\frac{k}{l})^l)$. Remember that the average number of visited nodes at the level l (for small l) is $\theta(\min\{n, \sigma^l\})$.
Now we will speak about searching of a single pattern in the text using our automaton with depth-first search technique. We can define three cases of analysis of this search process:

- $L(k) \geq \log_\sigma n$, $n \leq \sigma^{L(k)}$ - 'small n', online search is preferable and no index is needed (since the total work is n); It shows that the indexing technique does not work for very small texts.
- $m + k < \log_\sigma n, n > \sigma^{m+k}$ - 'large n', the total search cost is

  $$\sigma^{L(k)} + \frac{\sigma^k(1+\beta)^{2(m+k)}}{\beta^{2k}},$$

  independent of n;
- $L(k) < \log_\sigma n \leq m + k$, 'intermediate n', the search is sublinear of n in time if error level $\beta < 1 - \frac{e}{\sqrt{\sigma}}$.

Now let's add to our analysis the pattern partitioning mechanism. Remember that j here is a number of pieces in which pattern is divided. After dividing, the mechanism analised above is used. With pattern partitioning,

- First case conditions are:

  $$\frac{k}{1-\frac{c}{\sqrt{\sigma}}} \geq j\log_\sigma n, n \leq \sigma^{L(\frac{k}{j})},$$

  complexity is $O(n)$.

- Here $m + k < j\log_\sigma n, n > \sigma^{\frac{m+k}{j}}$, if $\beta = \frac{k}{l} < 1 - \frac{e}{\sqrt{\sigma}}$ the complexity is $O(n^{1-\log_\sigma \frac{1}{\gamma(1+\beta)}})$. This is sublinear of n, we use $j = \frac{m+k}{\log_\sigma n}$. This j is simply the smallest of possible j's in this case (with j less than $\frac{m+k}{\log_\sigma n}$ we get into the first case).
- Third case has it's own j for the minimum of complexity, but it can get over the bounds of this case, so in most cases we can simply use such j as in second case, and we will get sublinear of n time complexity.

3.6 Conclusions

- The splitting technique balances between traversing too many nodes of the suffix tree and verifying too many text positions
- The resulting index has sublinear retrieval time $O(n^\lambda), 0 < \lambda < 1$ if the error level is moderate.
- In future there can appear more exact algorithms to determine the correct number of pieces in which the pattern is divided and there are (and may appear in future) some better algorithms for verifying after matching a piece of pattern.