Chapter 12

Automaton Searching on Tries.

Mikhail Lakunin

Above there were considered a lot of algorithms that allow us to search for a pattern in a string and a few powerful methods to give asymptotic estimations of such algorithms. There is an extension of such algorithms, not for one pattern but for a regular expression. The algorithms that are to be reviewed there run on the preprocessed text (we use Patricia tree) and are of logarithmic *expected* time of the size of the text for the stricted class of regular expressions and in a sublinear *expected* time for all regular expressions.

It's the first such algorithm to be found with this complexity.

The main purpose of the text is to give understanding of what's going on, and therefore some of the technicals could be omitted. For the precise explanation the reader is referred to the original paper.

12.1 Structure of the paper

1. What we actually want to do or our task
2. The algorithms in a "few words"
3. Introducing indexing structures such as suffix tries and Patricia trees and a notation used in the Regular Languages theory(the Theory of formal languages)
4. Consider the simple algorithm for the stricted class of a regular expression that runs in logarithmic time.
5. Consider more complicated general algorithm.
6. Give estimation of the general algorithm considering sketch of technicals.
7. How to apply "general estimation theorem" to the particular cases.
8. Consider some heuristic for improving size of the query.
9. Open problems and conclusion (generalization of what we has spoken about)

12.2 Our task or what we want to do

We want to find efficient algorithm of finding all occurrences of a query given in the form of a regular language searching on the static preprocessed text. By finding all the occurrences we mean that we are going to find all the positions of the text from where one substring from our query set could start. In general queries are not of the big length, so there and below we state that the size of the query is bounded by the constant. We use there automaton that searches through the indexing tree and we state that our algorithm is pretty efficient, i.e. works in sublinear expected time (in average case). Moreover there exist small stricted subset of such queries that the expected time will be logarithmic. Besides we consider a powerful tool for estimating expected time of given automaton, the theorem that allows it on basis of some knowledge about incidence matrix of the automaton. And after all of that we suggest an heuristic for enhancing or for clarifying the query that helps to work all the algorithm faster.

12.3 Algorithm in a “few words”

As in previous parts we've got a large input text that is static or in other words we can construct indexing structure on it and we don't bother about how to reconstruct it if the text will change. So we construct a trie (from word retrieval) that is indexing binary tree (we suppose our alphabet is binary; if it's not we just use some encoding to obtain binary text and hence binary alphabet) was discussed by Olga Sergeeva. It becomes obvious to us that we can find arbitrary substring in the text in height of indexing tree length. In the case of perfectly balanced tree it's $\log_2 n$, where n is the length of the string.

Now we're able to find a single substring in a large text in logarithmic of the size of the text expected time (in average case, considering uniform distribution) Let's complicate the task. If set of all the substrings we are to find in the text could be represented as a prefix from *minimal preword set* concatenated with any continuation. For such a set we can state that it's possible to search for that query in a time $o(\|query\|)$ independently of the size of the answer, i.e. in logarithmic expected time of size of the text.

Then we consider general regular expression query, construct Deterministic Automaton for it and traverse it on the index trie. It could be showed that Automaton don't visit all the vertices of the index trie (the quantity of which is $O(n)$),but only a part of them in average case, that's the main idea of sublinear expected time.

12.4 Basic index structures and notation

To approach crystal understanding we need to introduce or just to help to recall some basics facts we are going to manage with soon after.

As it has been said we consider only binary alphabet, any other alphabet could be easily converted to the binary one with some encoding

12.4.1 Indexing structures

Definition 12.1. Trie is a binary index tree for a text(tree each edge of that is marked with a character), that satisfy following conditions:

1. each path of it (from the root to a leaf) consist of the prefix of one and only one of the suffixes of the text
2. paths are as short as possible satisfying first condition
3. each leaf of it is marked with the position of the first character of the corresponded suffix

The main problem of the pure trie is that the upper bound of the number of inner nodes is $O(n^2)$ that approaches in unbalanced tries, that's why in practice enhanced structure is of often use. It's so-called Patricia tree.

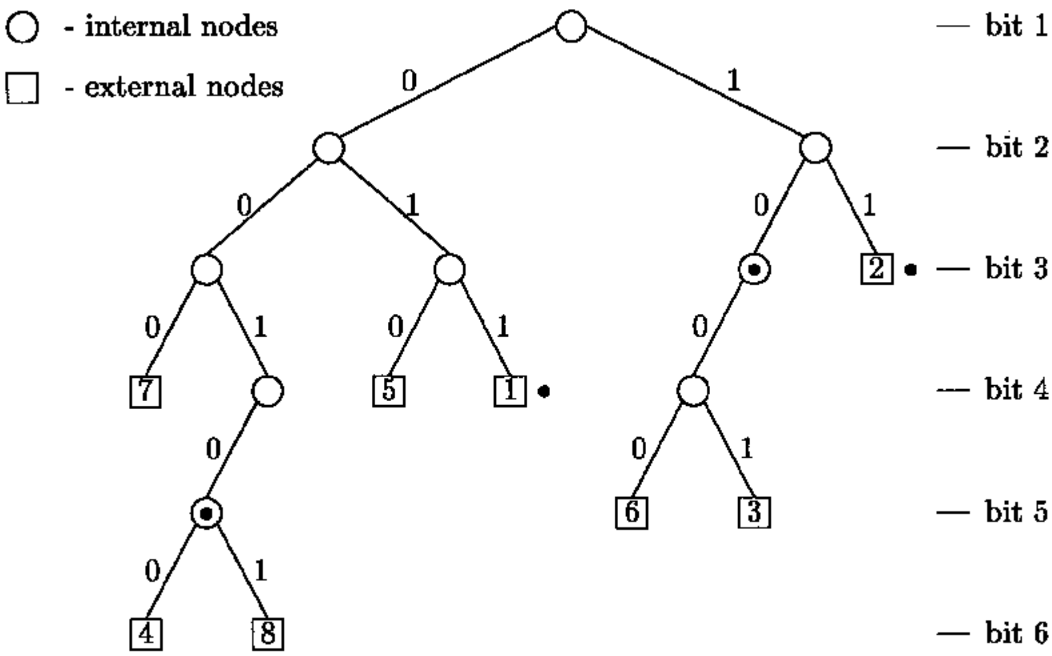

Figure 12.1: Binary trie (external node label indicates position in the text) for the 01100100010111

Definition 12.2. Patricia tree constructed on the basis of the trie, but with one enhancement. We exclude such nodes that have only one descendant and one ancestor, in other words we exclude nodes that are not leafs and that are not nodes of the "bifurcation". To not miss the correspondence with the initial text we remember the quantity of steps we need to omit to get to the next "bifurcation" node.

Note. Asymptotically in average case:
Height of the trie is: $2log_2(n) + O(log_2(n))$ [Reg81]
Height of the Patricia tree: $log_2(n) + O(log_2(n))$ [Reg81]

12.4.2 Basic definitions of the Theory of Formal Languages

General definition and notation.

1. $\mathbf{\Sigma}$ is a set of all characters (Alphabet), in some cases one character from the set
2. ε is an empty string
3. **xy** is concatenation of strings x and y
4. $\mathbf{w} = \mathbf{xyz}$ if w is concatenation of strings x, y, z then:
   (a) **x** is a prefix of w
   (b) **z** is a suffix of w
   (c) **y** is a substring of w

Let's define operations we are to use specifying regular expression (RE).

1. $\mathbf{r}, \mathbf{q}$ are sets of strings
2. $\mathbf{r} + \mathbf{q}$ is a unite of these sets ($r + q = r \cup q$)
3. **r**? is one or zero occurrences of r ($r? = \varepsilon + r$ in our notation)
4. $\mathbf{r^k}$ is k occurrences of r
5. $\mathbf{r^{\leq k}}$ is from zero to k occurrences of r ($r^{\leq k} = \sum_{i=0}^{k} r^i$)
6. $\mathbf{r}^*$ is Kleene closure,i.e. any number of occurrences of r ($r^* = \sum_{i=0}^{+\infty} r^i$)
7. $\mathbf{r}^+$ is one or more occurrences of r ($r^+ = \sum_{i=1}^{+\infty} r^i$)

12.5 Algorithm for a restricted class of regular expression

There we consider algorithm of searching for the stricted class of queries. It was briefly reviewed above, there we define it more exactly.
Let's introduce new expression class so-called *prefixed regular expressions* that is subclass of regular expressions class that was defined above.

Definition 12.3. Prefixed regular expression (PRE) is:

1. $\emptyset$ is a PRE (the empty set)
2. ε is a PRE($\{\varepsilon\}$ the set of empty string)
3. for each $a \in \Sigma$, a is a PRE ($\{a\}$)
4. if $p, q \in$ PRE and $r \in$ RE (such that $\varepsilon \in r$) and $x \epsilon \Sigma$ then:
   (a) $p + q$ is a PRE (union)
   (b) xp is a PRE (concatenation with a character on the left)
   (c) pr is a PRE (concatenation with ε-regular expression on the right)

(d) $p*$ is a PRE

Example 12.1. This is PRE

$$ab(bc^* + d^+ + f(a+b))$$

Example 12.2. This is not PRE

$$a\Sigma^* b$$

Example 12.3. This is not PRE too

$$(a+b)(c+d)(e+f)$$

Statement. There exist such **unique** and **finite** subset(we call it *preword* set) of the (given) PRE set that satisfy following conditions:

1. for every word in the PRE set there exist unique prefix from *preword* set
2. for every word from the *preword* set there's no other prefix in the set

To search a PRE query, we construct a tree (so-called Complete Prefix Trie) in similar way as we construct trie, but we use not suffixes of the text as we do in case of trie but all the strings in *preword* set of the query. And then traverse through that tree and through our index trie simultaneously for an answer.

Definition 12.4. Complete Prefix Trie (CPT) is the trie of the set of the strings such that:

1. there are no truncated paths(as in Patricia tree), that is, every word corresponds to a complete path in the trie
2. if a word x is a prefix of another word w, then only x is stored in the trie.

The second rule has appeared since the search for x is sufficient to also find the occurrences of w. We are going to find only starting positions, aren't we?
We've constructed the Complete Prefix Trie so we are now able to traverse through that.
We traverse simultaneously, if possible, the complete prefix trie(CPT) and the trie (Patricia tree) of all suffixes of the text (the index). That is, follow at the same time a 0 or 1 edge(we consider binary alphabet), if they exist in both the trie and the CPT. All the subtrees in the index associated with terminal nodes in the complete prefix trie are the answer. As in prefix searching, because we may skip bits while traversing the index, a final comparison with the actual text is needed to verify the result.
Let's formulate the algorithm.

Algorithm. Searching for a query:

1. Construct the Complete Prefix Trie from the query
2. Traverse simultaneously the complete prefix trie and Patricia tree to obtain an answer.

Note. Since we may skip bits while traversing the index(if we used Patricia tree), a final comparison with the actual text is needed to verify the result.

Because the number of nodes of the complete prefix trie of the *preword* set is $O(|query|)$, the search time is also $O(|query|)$.

Conclusion. It is possible to search a PRE-query in time $O(|query|)$ independently of the size of the answer.

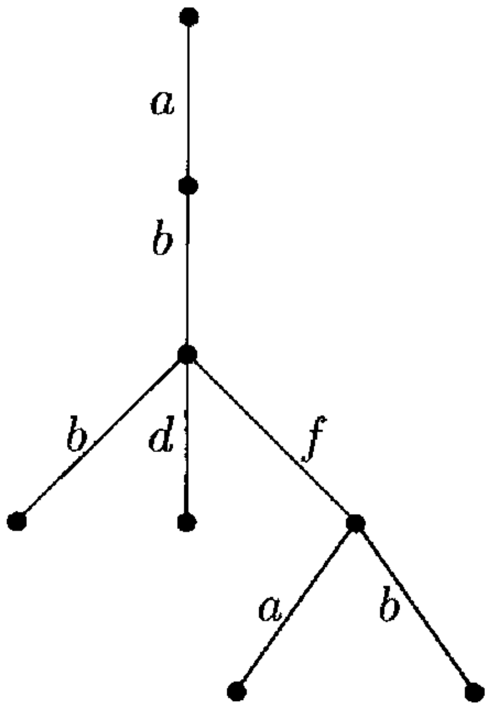

Figure 12.2: Complete Prefix Trie for a query $ab(bc^* + d^+ + f(a+b))$

12.6 General algorithm

The extension of the previously discussed idea leads to general algorithm of searching query given as regular expression.

12.6.1 What we want to do there

There we present the algorithm that can search for arbitrary regular expression in time sublinear in n on the average. For this, we simulate a DFA (Deterministic Finite Automaton) in a binary trie built from all the suffixes of a text.
Since the situation is pretty similar with the previous algorithm. Let's start from presenting algorithm itself.

Algorithm. General Automaton Search

1. Convert the regular expression(query) into minimized DFA(independent of the size of the text)
2. Eliminate outgoing transitions from final states
3. Convert character DFA into binary DFA (Each state will then have at most two outgoing transitions, one labeled with 0 and one labeled with 1), i.e. applying some encoding
4. Simulate binary DFA on the binary trie of all suffixes we've constructed(See pictures of DFA and binary index trie).
   That is:
   (a) root of the tree with initial state.
   (b) for any internal node associated with state i, associate its left descendant with state j, if $i > j$ for a bit 0, and associate its right descendant with state k, if $i > k$ for a bit 1.
5. For every node of the index associated with a final state, accept the whole subtree and halt.
6. On reaching an external node, run the remainder of the automaton on the single string determined by this external node.

12.7 Efficiency Estimation for the General Algorithm

The precise average case analysis of the above algorithm is not simple. Some explanations were given in "few words" in the start of the paper, There I'll try to clarify the situation a bit, for precise explanation reader is referred to the original paper.

12.7.1 The structure of the proof

The situation is as follows. We have Patricia indexing trie with $O(n)$ nodes and the DFA(Deterministic finite automaton) and we want to calculate how many nodes of the Patricia indexing trie will be visited by automaton. We want to prove that this quantity is less then $O(n)$ (asymptotically).
There I want to give a sketch of the proof on step by step basis.

1. We introduce N_n^i, i.e. the quantity of nodes of our index Patricia tree that was visited by Automaton when the quantity of suffixes in the tree is n and we've engaged our automaton on the i-th node.

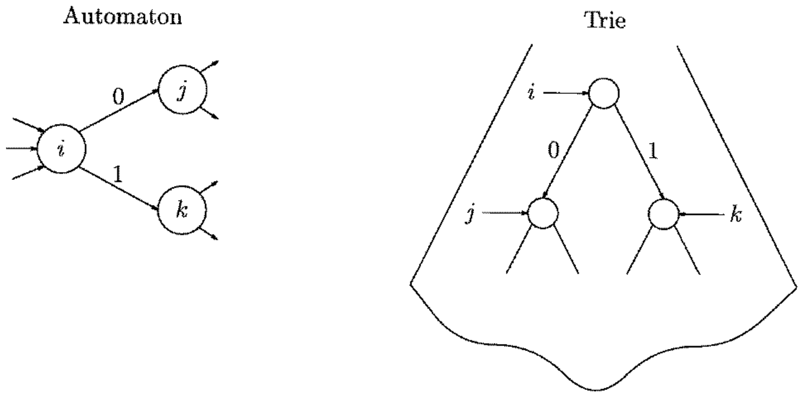

Figure 12.3: DFA with corresponded index trie

4

2. We note that our automaton from each node can go only in 2 directions in the index tree and taking into account all the possible combinations of the ways suffixes could go in i-th node we can say , therefore we can say that:

$$N_n^i = 1 + \frac{1}{2^n}\left(\sum_{l=0}^{n}\binom{n}{l}\left(N_l^j + N_{n-l}^k\right)\right)$$

   where j, k are states DFA goes on $0, 1$ correspondingly (see picture with DFA and trie)
3. Now it looks like equation could be solvable with generating function. We apply generate function,
4. and convert it to matrix equation writing it for all the starting nodes in a unit.
5. Then we "solve" equation and find the generating function. The generating function equals to the matrix series, where the matrix is incidence matrix of DFA.
6. Then we convert it to a Jordan form and decompose matrix equation looking at the single values in a matrix.
7. Then we apply Mellin transform method to get some knowledge about this single values series and about matrix series
8. Using that we got some estimation of eigenvalues and information about N, i.e. about number of nodes was visited and about number of comparison DFA made (in average case)
9. Finally we obtain following theorem(taking in account number of steps we need for verifying in case of Patricia tree):

   Theorem 12.1. *The expected number of comparisons performed by a minimal DFA for a query q represented by its incidence matrix H while searching the trie of n random strings is sublinear, and given by:*

$$O\left((log_2 n)^t\, n^r\right)$$

   ,where $r = log_2\lambda$,$\lambda = max_i(|\lambda_i|)$,$t = max_i(m_i - 1$,such that $|\lambda| = \lambda_i)$, and the λ_i are the eigenvalues of H with multiplicities m_i.

12.8 How to apply "general estimation theorem" to the particular cases

To analyze average time of searching for particular query or particular type of the query ,we need to consider DFA of the query and its incidence matrix and eigenvalues of the matrix. And then we can apply "the main theorem" to obtain estimation.

Example 12.4. For example, DFAs having only cycles of length 1, have a largest eigenvalue equal to 1, but with multiplicity proportional to the number of cycles, obtaining a complexity of $O(polylog(n))$.

Example 12.5. For the regular expression $(0(011)0)^*1$, the eigenvalues are:

$$2^{1/3}, -\frac{1}{2}(2^{1/3} - 3^{1/2}2^{1/3}i), -\frac{1}{2}(2^{1/3} - 3^{1/2}2^{1/3}i), 0$$

, and the first 3 have the same modulus. The solution in this case is $N_n^1 = O(n^{1/3})$

12.9 Heuristic for optimizing the query

12.9.1 What we mean by optimizing

In this section, we present a general heuristic, which we call **substring analysis**, to plan what algorithms and order of execution should be used for a generic pattern matching problem, which we apply to regular expressions.
The aim of this section is to find from every query a set of necessary conditions that have to be satisfied.

12.9.2 Substring graph

Definition 12.5. Substring graph of a regular expression is an acyclic directed graph such that each node is labeled by a string. And its defined recursively by the following rules:

1. $G(\varepsilon)$ is a single node labeled ε.
2. $G(x)$ for any $x \in \Sigma$ is a single node labeled with x.
3. $G(s+t)$ is the graph built from $G(s)$ and $G(t)$ with an ε-labeled node with edges to the source nodes and an ε-labeled node with edges from the sink nodes.
4. $G(st)$ is the graph built from joining the sink node of $G(s)$ with the source node of $G(t)$, and relabeling the node with the concatenation of the sink label and the source label.
5. $G(r^+)$ are two copies of $G(r)$ with an edge from the sink node of one to the source node of the other.
6. $G(r^*)$ is two ε-labeled nodes connected by an edge .

12.9.3 How to Use

reducing the size of the query

After building $G(q)$, we search for all node labels in $G(q)$ in our index of suffixes, determining whether or not that string exists in the text ($O(|q|)$ time). For all nonexistent labels, we remove:

1. the corresponding node,

2. adjacent edges,
3. and any adjacent nodes (recursively) from which all incoming edges or all outgoing edges have been deleted.

This reduces the size of the query.

estimating final answer size

From the number of occurrences for each label we can obtain an upper bound on the size of the final answer to the query:

1. for adjacent nodes (serial, or "and" nodes) we multiply both numbers
2. for parallel nodes ("or" nodes) we add the number of occurrences

Note. ε-nodes are treated in special way.

Managing ε-nodes:

1. consecutive serial ε-nodes are replaced by a single ε-node.
2. chains that are parallel to a single ε-node, are deleted
3. the number of occurrences in the remaining ε-nodes is defined as 1

After the simplifications,ε-nodes are always adjacent to non-ε-nodes, since ε was assumed not to be a member of the query

12.10 Open problems and conclusion

We have shown that using a trie or Patricia tree, we can search for many types of string searching queries in logarithmic average time, independently of the size of the answer. We also show that automaton searching in a trie is sublinear in the size of the text on average for any regular expression, this being the first algorithm found to achieve this complexity. Similar ideas have been used since for approximate string searching by simulating dynamic programming over a digital tree [Gonnet et al. 1992; Ukkonen 1993], also achieving sublinear time on average. In particular, Gonnet et al. [1992] have used this algorithm for protein matching.

In general, however, the worst case of automata searching is linear. For some regular expressions and a given algorithm it is possible to construct a text such that the algorithm must be forced to inspect a linear number of characters. The pathological cases consist of periodic patterns or unusual pieces of text that, in practice, are rarely found.

Finding an algorithm with logarithmic search time for any RE query is still an open problem [Galil 1985]. Another open problem is to derive a lower bound for searching REs in preprocessed text.