Fast Text Searching for Regular Expressions or Automaton Searching on Tries

RICARDO A. BAEZA-YATES
University of Chile, Santiago, Chile
AND
GASTON H. GONNET
Informatik, Zurich, Switzerland

Made by Mike Lakunin
Saint-Petersburg State University

Introduction

- In the article it's considered the algorithm for efficient searching a set of strings described in terms of standard formal language (that is regular expression) in an extensive string (a text)

Where it supposed to be used in

Pattern Matching and Text searching are very important components of many problems such as:

- Text editing
- Data retrieval
- Symbol manipulation
- And many others…

Types of Searching

-Text:

- Preprocessed
- Not preprocessed

- Language specifications
  - Regular Expression (**RE**)
  - Other language specifications

There we're interested in preprocessed case when query is specified by **RE**

General Problem of Searching

The general searching problem consist of finding occurrences in a string and obtaining some of the following information:

1. The location of the (first) occurrence
2. The number of occurrences
3. All the locations of the occurrences

And in general they're not of the same complexity.

Refinement of Problem

We are to find only starting position for a match so it's enough to find the shortest string that matches query in a given position.

Besides we assume that empty string is not a member of our language. (trivial answer)

Work background
Traditional Approach

Traditional approach for search is to use Finite Automaton (**FA**) that recognizes the language defined by Regular Expression (**RE**) with the text as input.

Traditional Approach
Main Problems

- All the text must be processed so algorithm is linear of size of the text (for many applications it's unacceptable)

- If the automaton is deterministic both the construction time and number of states can be exponential in the size of RE (not crucial in practice)

Contribution of the paper

- There was many results for searching strings & set of strings however no new results for the time complexity of time complexity of search of **RE** have been published
- It's the first algorithm found which achieve **sublinear** expected time to search any **RE** in a text and **constant** or **logarithm** expected time for some restricted **RE**

Main Idea of Algorithm

The main idea behind these algorithms is the **simulation** of the **finite automaton of the query over a digital tree** (or Patricia tree) of the text (our index).

The time savings come from the fact that **each edge** of the tree is **traversed at most once**, and that **every edge represents pairs of symbols in many places of the text**.

How the report organized

1. Some terminology and the background
2. Algorithm in case of restricted class of **RE** (Case of **Prefixed RE**)
3. The main result: algorithm to search any **RE** with its expected time analysis
4. Heuristic to optimize generic pattern matching query

1. **Some terminology and the background**
2. Algorithm in case of restricted class of **RE** (Case of **Prefixed RE**)
3. The main result: algorithm to search any **RE** with its expected time analysis
4. Heuristic to optimize generic pattern matching query

Preliminary
Notation

- $\sum$ is an alphabet
- **ε** is an empty string
- **xy** is a concatenation of string x and string y

w=xyz:

- **x** is a prefix of **w**
- **z** is a suffix of **w**
- **y** is a substring of **w**

Preliminary
Notation (continuation)

- **+** is a union
- * is a Kleene closure

  i.e. **r*** is 0 or more occurrences of **r**
- ∑ is any symbol from ∑
- **r?**=e+r (0 or 1 occurrence of **r**)

Relationships between **FA**

Regular languages are accepted by deterministic (**DFA**) or nondeterministic

(**NFA**) finite automata.

- For **DFA** we use standard definition
- There is a simple algorithm that, given a regular expression *r*, constructs a **NFA** that accepts ***L*(*r*)** in ***O*(|r|)** time and space.
- There are also algorithms to convert a **NFA** to a NFA without e transitions(**$O(|r|^2)$** and to a **DFA** (**$O(2^a)$** (where **a=|r|**) states in the worst case)

Different Kinds of Text

- **Static** (changes rarely)
- **Dynamic** (changes often)

- **Structured**
  (consider a text as a set of words)
- **Unstructured**
  (consider a text as a single string)

Index for a Text
Definitions

- **Text** is string padded at its right end with an infinite number of null.
- **Suffix** or **Semi-infinite string (sistring)** is the sequence of characters starting at any position of the text and continuing to the right.
- **Tries** are recursive tree structures that use the digital decomposition of strings to represent a set of strings and to direct the searching.

About Tries (more precisely)

- If the alphabet is ordered, we have a lexicographically ordered tree. The root of the trie uses the first character, the children of the root use the second character, and so on. If the remaining subtrie contains only one string, that string's position is stored in an external node.

Examples

- **Original text:**

  ”The traditional approach for searching a regular expression…”

- **Sistrings**

1. “The traditional approach for searching …”
2. “he traditional approach for searching a…”
3. “e traditional approach for searching a …”
4. “onal approach for searching a regular …”

Examples (continuation)

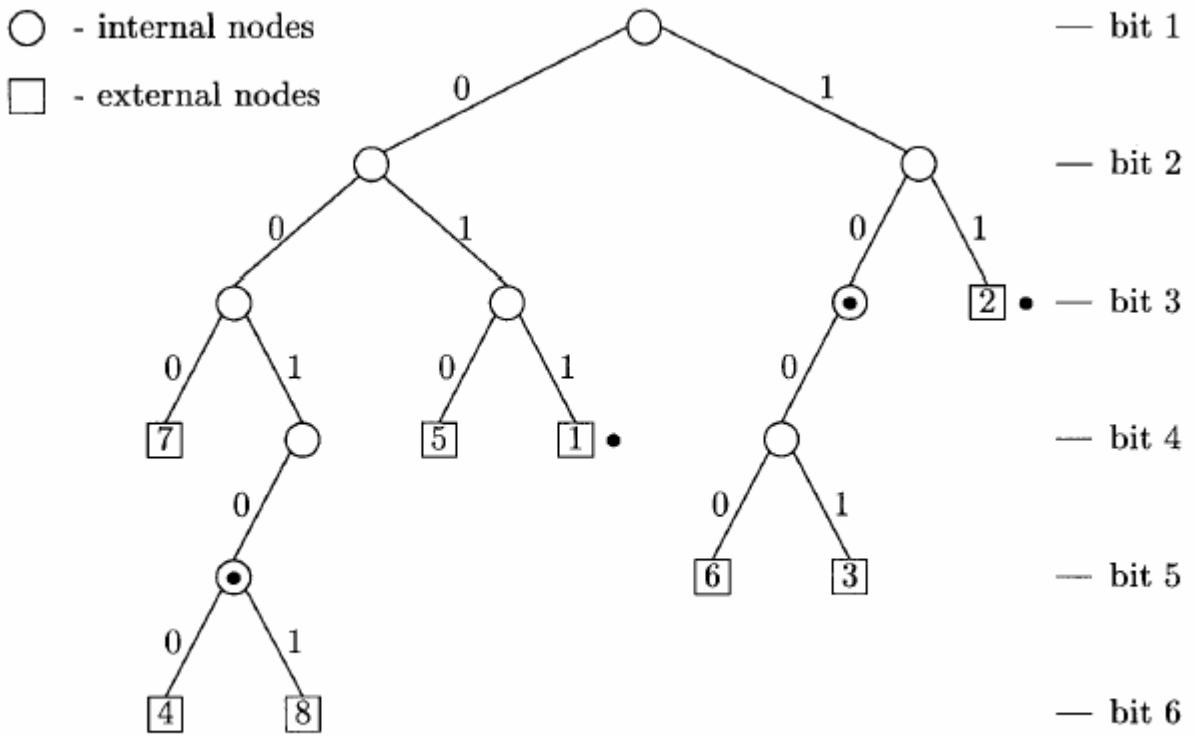


FIG. 1. Binary trie (external node label indicates position in the text) for the first eight sistrings in "01100100010111 . . .".

Practical Aspect
Patricia tree

- Ordinary Tries – Internal nodes **$O(n^2)$**
- **Patricia Tree-**A *Patricia tree* [Morrison 1968] is a trie with the additional constraint that single-descendant nodes are eliminated. A counter is kept in each node to indicate which is the next bit to inspect.

- Patricia Tree internal nodes quantity is **n – 1**

n –there and below is number of sistrings

Some numbers around tries

- Expected height of a random trie:

$$H(n) = 2\log_2(n) + o(\log_2(n))$$

- Expected height of a Patricia tree:

$$H(n) = \log_2(n) + o(\log_2(n))$$

Suffix Tries

- **Suffix Tree** is a trie using all sistrings
- **Compact Suffix Tree** the same as Patricia trie (but with a little modification in storage method)
- **PAT arrays** (just stores the leaves of the trie in lexicographical order according to the text contents)

Relationships between Tries and Suffix Trees

- Although a **random trie** (independent strings) is different from a **random suffix tree** (***dependent*** strings) in general, both models are **equivalent** when the symbols are **uniformly** distributed. Moreover, the **complexity should be the same for both cases**, because **Hst(*n*)<= Ht(*n*).** Simulation results suggest that the asymptotic ratio of both heights converges to 1

1. Some terminology and the background
2. **Algorithm in case of restricted class of RE (Case of Prefixed RE)**
3. The main result: algorithm to search any **RE** with its expected time analysis
4. Heuristic to optimize generic pattern matching query

Searching a Set of Strings
With common prefix

- Efficient searching of all sistrings having a given prefix can be done using a Patricia tree.

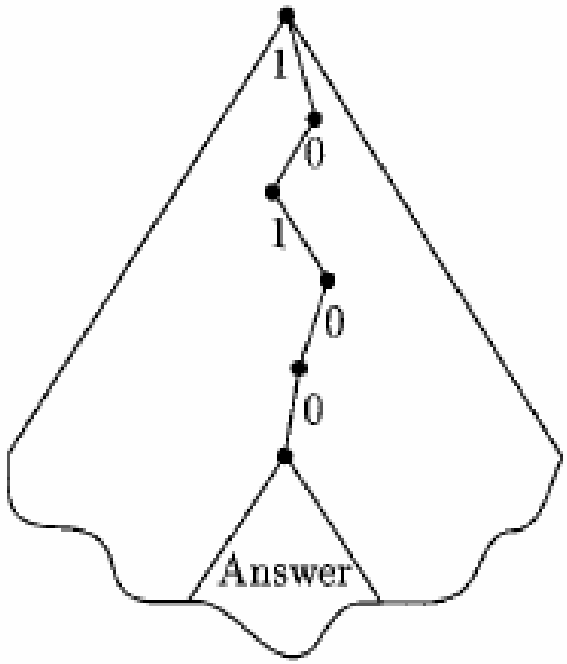


FIG. 2. (a) Prefix searching for "10100". (b)

Finite Set of Strings

We extend the prefix searching algorithm to the case of a finite set of strings.

- The simplest solution is to use prefix searching for every member of the set.
- However, we can improve the search time by recognizing common prefixes in our query.

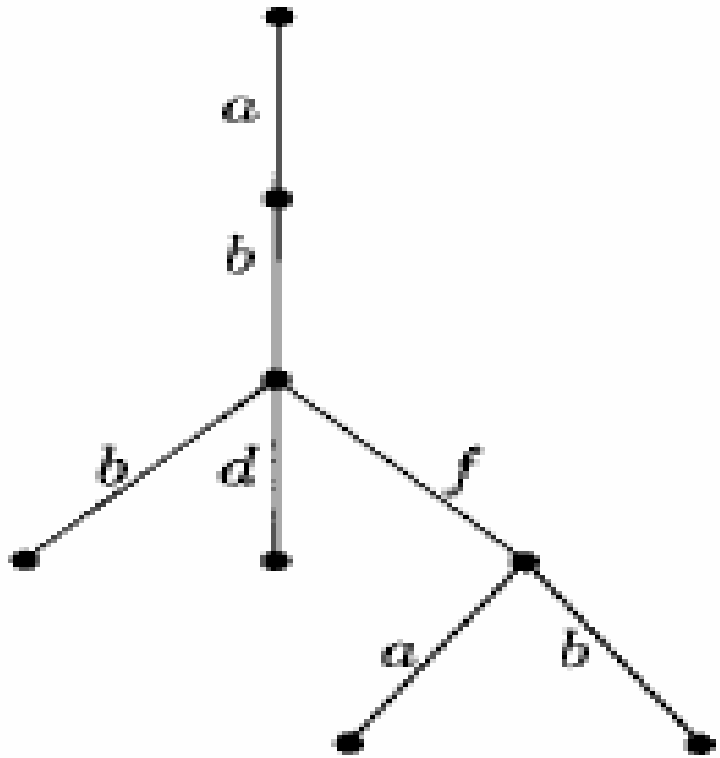
Complete Prefix Tree (**CPT**)

Complete prefix trie (**CPT**) is the trie of the set of strings, such that:

- there are **no truncated paths**, that is, every word corresponds to a complete path in the trie; and

- if a word ***w*** is a **prefix** of another word ***x***, then *only* ***w*** is stored in the trie. In other words, the search for ***w*** is sufficient to also find the occurrences of ***x***.

Complete Prefix Tree (picture)



Complete prefix trie of the words abb, abd, $abfa$, $abfb$, $abfaa$, and $abfab$.

First version of algorithm

- Construct **CPT** of the query using binary alphabet
- Traverse simultaneously, if possible, the complete prefix trie and the Patricia tree of sistrings in the text (the "index"):

  - That is, **follow** at the same time a **0** or **1 edge**, 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**.

Note:

- Since we may skip bits while traversing the index, a final comparison with the actual text is needed to **verify the result**.

Analyzing results of first version of algorithm

- Let **|*S*|** be the sum of the lengths of the set of strings in the query. The construction of the complete prefix trie requires time ***O*(|*S*|).** Similarly, the number of nodes of the trie is ***O*(|*S*|)**, thus, the searching time is of the same order.

- An extension of this idea leads to the definition of prefixed regular expressions.

Prefixed Regular Expression (**PRE**) (definition)

Def (recursive definition):

- **Ø** is a **PRE** and denotes the empty set.
- **ε** (empty string) is a **PRE** and denotes the set {**ε**}.
- For each symbol ***a*** in ∑, ***a*** is a **PRE** and denotes the set **{*a*}**.

Prefixed Regular Expression (**PRE**) (definition) (continuation)

If ***p*** and ***q*** are **PRE**s denoting the regular sets ***P*** and ***Q***, respectively, ***r*** is a regular expression denoting the regular set ***R*** such that **ε** is in ***R***, and ***x*** *is* in S,then the following expressions are also **PRE**s:

- ***p*** + ***q*** (union) denotes the set ***P*** **U** ***Q***.
- ***xp*** (concatenation with a symbol on the left) denotes the set ***xP***.
- ***pr*** (concatenation with an e-regular expression on the right) denotes the set ***PR***, and
- ***p**** (star) denotes the set ***P****.

Prefixed Regular Expression (**PRE**)
(Example)

PRE:

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

Not PRE:

$$a\Sigma^* b$$

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

Main Property of PRE

There exists a **unique finite** set of words in the language, called ***prewords***, such that:

- for any other string in the language, one of these words is a proper prefix of that string
- the ***prefix property****:* no word in this set is a proper prefix of another word in the set.

Applying **CPT** Technique to a **PRE** query

- To search a **PRE** query, we use the algorithm to search for a set of strings,using the prewords of the query. Because the number of nodes of the **complete prefix trie** of the **prewords** is **O(|*query*|)**, the search time is also ***O*(|*query*|)**.

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

How the report organized

1. Some terminology and the background
2. Algorithm in case of restricted class of **RE** (Case of Prefixed **RE**)
3. **The main result: algorithm to search any RE with its expected time analysis**
4. Heuristic to optimize generic pattern matching query

General Automaton Search

In this section, we present the algorithm that can search for **arbitrary regular expressions** in time **sublinear** in ***n*** on the **average**. For this, we simulate a **DFA** in a **binary trie** built from all the sistrings of a text.

General Automaton Search (Step by Step)

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

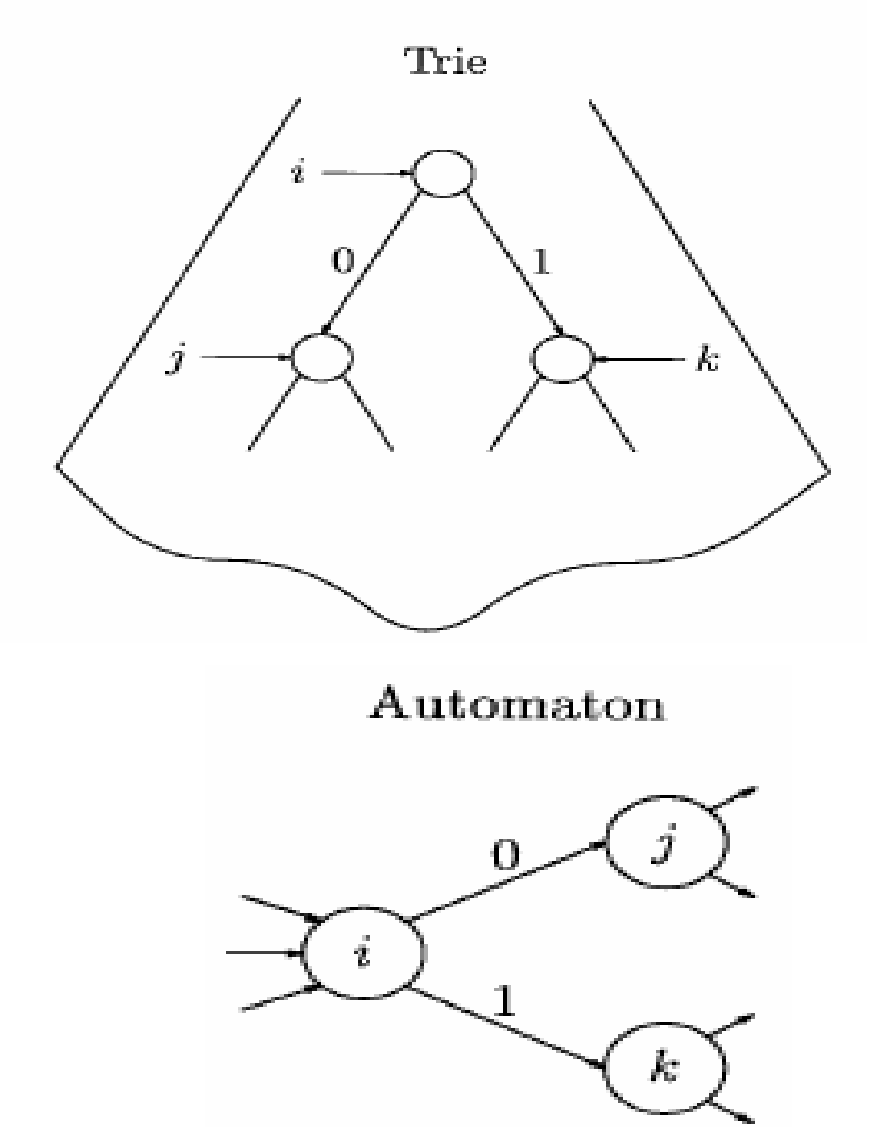
   (Each state will then have at most two outgoing transitions, one labelled 0 and one labelled 1)

General Automaton Search (Step by Step) (continuation)

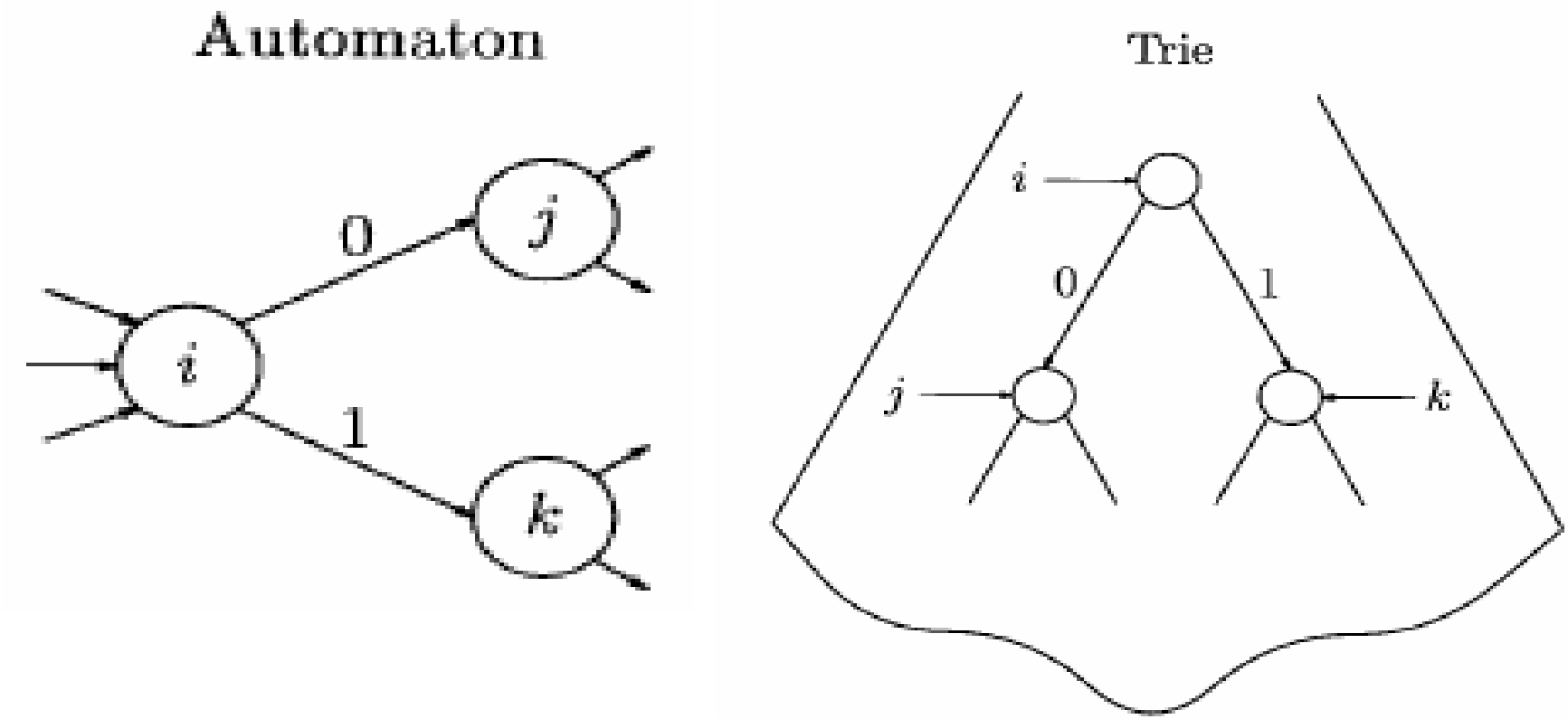
4. Simulate binary **DFA** on the binary trie of all sistrings. So associate:

   -root of the tree with initial state

   -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**



General Automaton Search (Step by Step) (picture)



General Automaton Search (Step by Step) (where to stop)

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.

Analyzing Algorithm

- To adapt the algorithm to run on a **Patricia tree**, it is necessary to read the corresponding text to **determine the validity of each "skipped" transition**(the sistring starting at any position in the current subtree may be used)

  The complexity of the algorithms is considered in the following theorem…

Main Theorem

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

$$O((\log_2 n)^t n^r)$$

$r = \log_2 \lambda < 1$

$\lambda = \max_i(|\lambda_i|)$

$t = \max_i(m_i - 1, \text{such that } |\lambda_i| = \lambda)$

Where the λ_is are the eigenvalues of H with multiplicities m_i

Remarks about the proof

- In the proof we use Mellin Transform

Analyzing Theorem

- As a corollary of the theorem above, it is possible to characterize the complexity of different types of regular expressions according to the DFA structure.
- 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 $O(polylog(n))$.

How the report organized

1. Some terminology and the background
2. Algorithm in case of restricted class of **RE** (Case of Prefixed **RE**)
3. The main result: algorithm to search any **RE** with its expected time analysis
4. **Heuristic to optimize generic pattern matching query**

Substring Analysis for Query Planning

- 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.

Substring Graph

- **Def** ***Substring graph*** of a regular expression ***r*** is an **acyclic directed graph** such that each node is labelled by a string.And it's defined recursively by the following rules

Substring Graph (recursive definition)

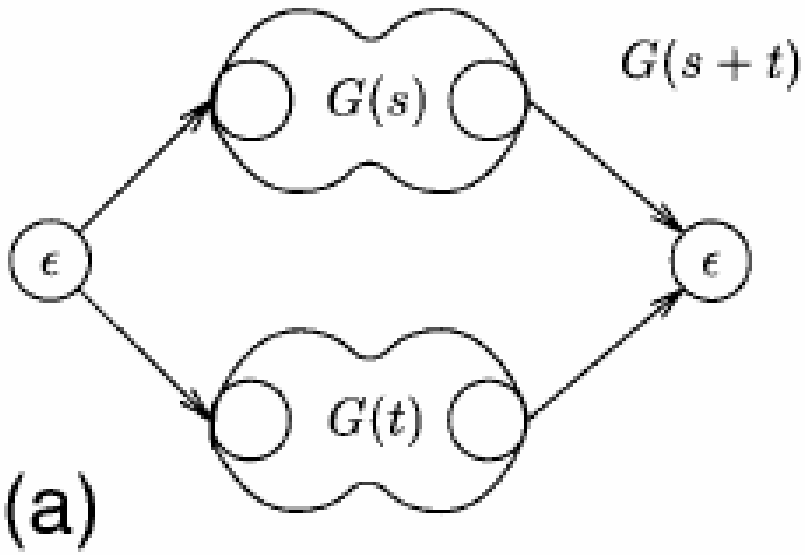
1. *G*(ε) is a single node labelled ε.

2. *G*(*x*) for any *x in* ∑ is a single node labelled with *x*.

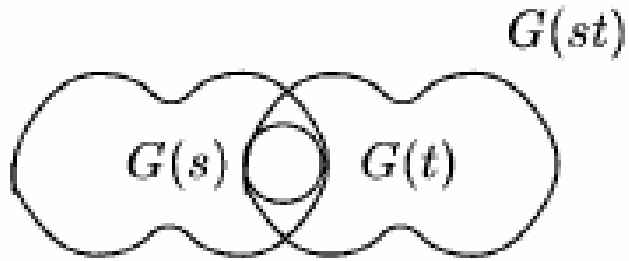
Substring Graph (recursive definition)

3. *G*(*s*+*t*) is the graph built from *G*(*s*) and *G*(*t*) with an ε-labelled node with edges to the source nodes and an ε-labelled node with edges from the sink nodes(a).

4. *G*(*st*) is the graph built from **joining** the **sink** node of *G*(*s*) with the **source** node of *G*(*t*) (b), and relabelling the node with the **concatenation** of the **sink** label and the **source** label.



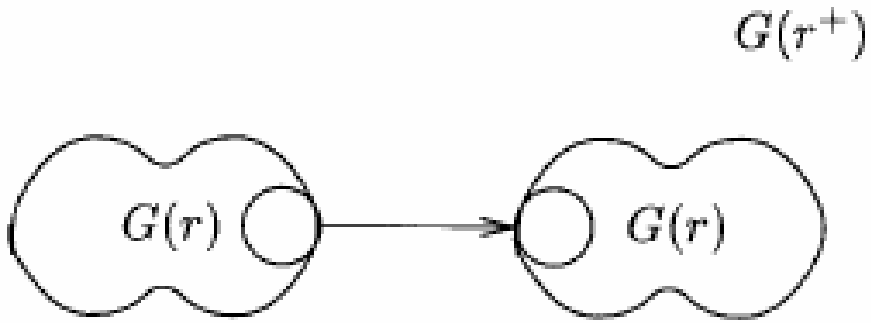
(a)



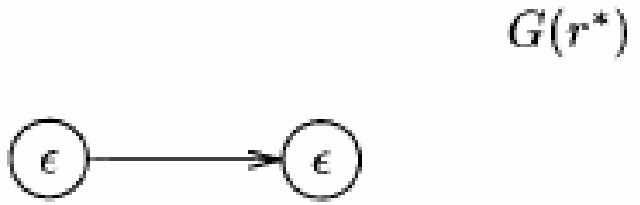
(b)

Substring Graph (recursive definition)

5. ***G*(*r*1)** are **two copies** of ***G*(*r*)** with an edge from the **sink** node of one **to** the **source** node of the other, as shown in (c).
6. *G*(*r**) is two ε-labelled nodes connected by an edge (d).

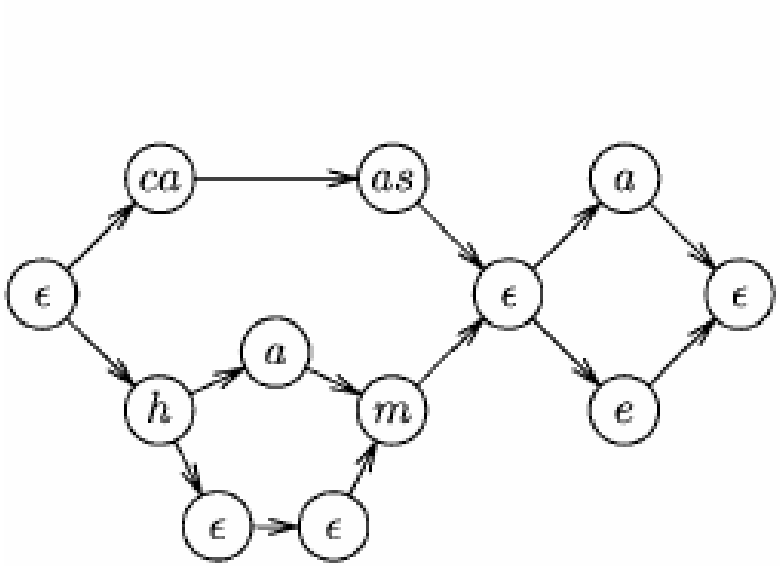


(c)

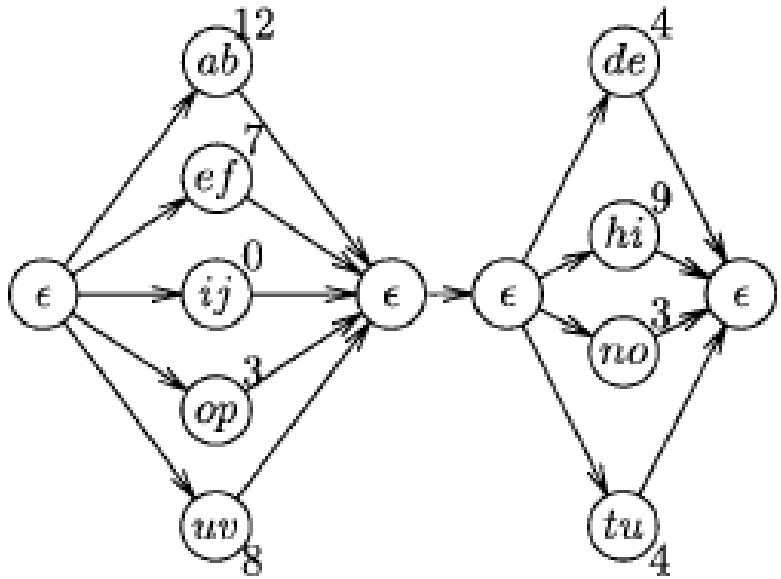


(d)

Substring Graph (examples)



(a)



(b)

(a) $(ca^{+}s + h(a + e^{*})m)(a + e)$

(b) $(ab + ef + ij + op + uv)\Sigma^{*}(de + hi + no + tu)$

Substring Graph (How to Use)

- After building $G(q)$, we search for all node labels in $G(q)$ in our index of sistrings, determining whether or not that string exists in the text (O(|q|) time). For all nonexistent labels, we remove:
- —the corresponding node,
- —adjacent edges, and
- —any adjacent nodes (recursively) from which all incoming edges or all outgoing edges have been deleted.

  This reduces the size of the query.

Substring Graph (How to Use) (continuation)

- **from the number of occurrences for each label we can obtain an upper bound on the size of the final answer to the query**:
  - For adjacent nodes (serial, or *and*, nodes) we multiply both numbers
  - for parallel nodes (*or* nodes) we add the number of occurrences

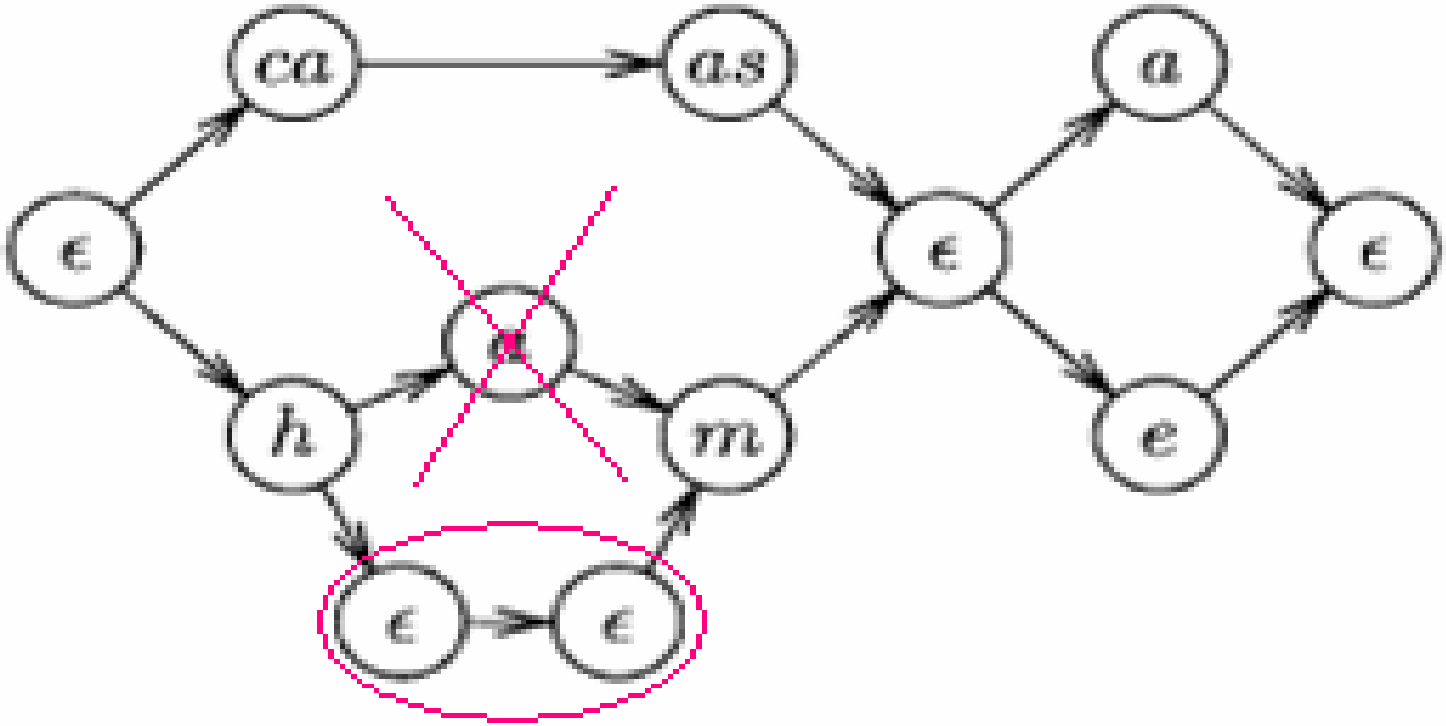
  Note: ε-nodes are simplified and treated in a special way

Substring Graph (How to Use) (managing ε-nodes)

- **consecutive serial ε -nodes** are replaced by a **single ε -node** (for example, the lowest nodes in (a)),<picture on the next slide>
- chains that are parallel to a single e-node, are deleted (e.g., the leftmost node labelled with ***a*** in (a))
- the number of occurrences in the remaining

  ε -nodes is defined as 1

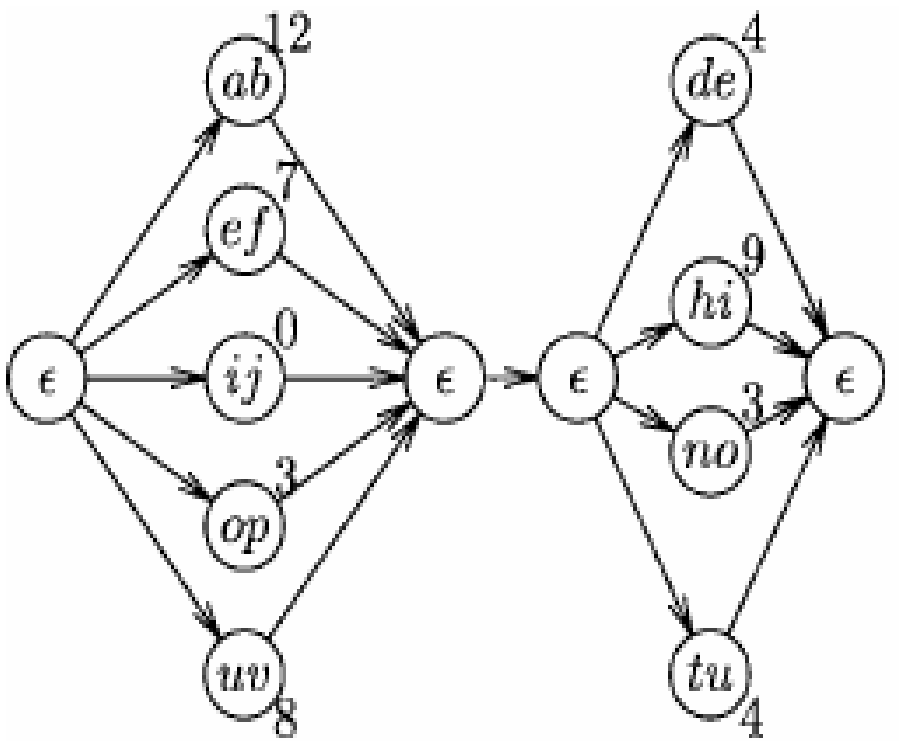
  (after the simplifications,ε -nodes are always adjacent to non-ε-nodes, since ε was assumed not to be a member of the query).



(a)

Substring Graph (How to Use) (example)

- In our example, the number of occurrences for Figure 6(b) can be bounded by600, this bound is useful in deciding future searching strategies for the rest of the query.



Result/Final Remark

- Using Patricia Tree we can search for many types of string independently of size of the answer in logarithmic average time
- Automaton searching in a trie is sublinear in the size of the text on average for any regular expression
- The worst case is linear. The pathological cases consist of periodic patterns or unusual pieces of text that, in practice, are rarely found.
- most real queries coming from users of Oxford English Dictionary as **RE** were resolved in about $O(\sqrt{n})$ node inspections

Result/Final Remark (open problems)

- To find algorithm with logarithmic search time for any **RE**
- to obtain the lower bound for searching **RE** in preprocessed text
- Also there is interesting problem is if there exist efficient algorithm for bounded “followed-by” problem:

$$s_1 \Sigma^{\leq k} s_2$$

The End