Text Analysis with Enhanced Annotated Suffix Trees

Algorithms and Implementation

Mikhail Dubov[1]

National Research University Higher School of Economics
Computer Science faculty, Moscow, Russia

AIST'2015

Table of Contents

Annotated suffix trees

- **Letter-based** method for text analysis
- **Annotated suffix trees:** full-text index
- **Basic computation:** relevance score of a keyphrase to the text collection indexed by AST
- **Range of applications:**
  - Text classification (e.g. spam filtering)
  - Feature extraction
  - Keyphrase analysis (stay tuned)

Suffix trees

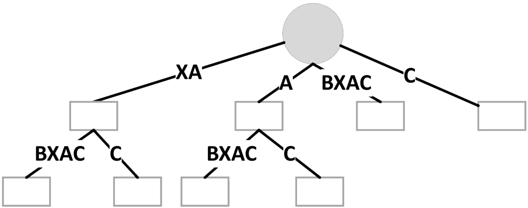


Figure: Suffix tree for string “XABXAC“

- **Suffix tree** for a string S ($|S| = n$) is a rooted directed tree encoding all the suffixes of that string [3]
- The concatenation of edge labels on every path from the root node to one of the leaves makes up one of the suffixes of that string, i.e. $S[i \dots n]$.
- It is also required that each internal node has two or more children, and each edge is labeled with a non-empty substring of S.

Suffix trees

- Various $O(n)$ construction algorithms exist (Ukkonen, Weiner)
- Establishes a linear-time solution for the **exact pattern matching** problem
- Suffix tree is a **full-text index**

Annotated suffix trees

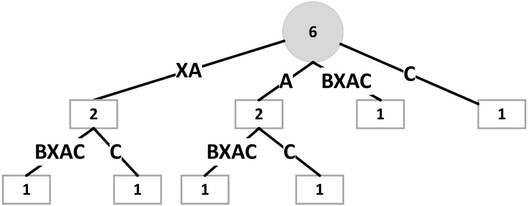


Figure: Annotated suffix tree for string “XABXAC“

- Extension: node labels
- Node label $f(v)$ indicates the number of entries of the substring on the path from root to v in the text collection

AST relevance score

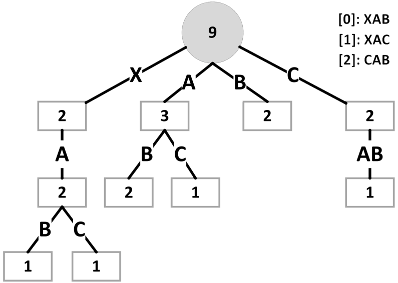


Figure: Naive AST representation (as a trie) for a collection of 3 strings

- *Conditional probability of a node* given its parent:

$$\hat{p}(v) = \frac{f(v)}{f(parent(v))}$$

AST relevance score

Relevance score computation for keyword S in text collection T (described in terms of a **trie**, not a **tree**):

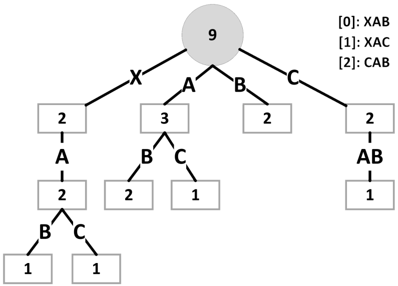
- For each suffix $S[i \dots n]$ of S, try to match it against the suffix tree $AST(T)$, starting at the root.
- If, for suffix s, we matched exactly k symbols in the tree, then

$$score_{suff}(s) = \frac{\sum_{i=1}^{k} \hat{p}(v_i)}{k},$$

  where v_i is the i-th node on the matching path starting at the root (if $k = 0$, then $score_{suff}(s) = 0$).
- The final score for keyword S is obtained as

$$SCORE(S) = \frac{\sum_{i=1}^{|S|} score_{suff}(S[i:])}{|S|}.$$

AST relevance score: Example 1

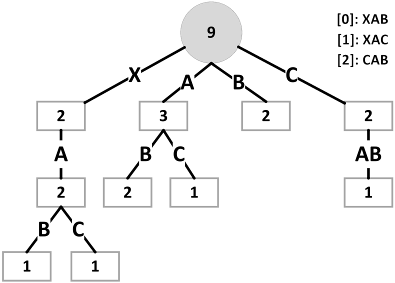


$$T = [\text{"}XAB\text{"}, \text{"}XAC\text{"}, \text{"}CAB\text{"}]$$

$$SCORE(\text{"}ABC\text{"}) = \frac{score(\text{"}ABC\text{"}) + score(\text{"}BC\text{"}) + score(\text{"}C\text{"})}{3} =$$

$$= \frac{(0.33 + 0.67)/2 + (0.22)/1 + (0.22)/1}{3} = 0.31$$

AST relevance score: Example 2

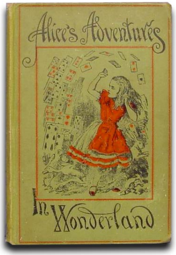


$$T = [\text{“}XAB\text{“}, \text{“}XAC\text{“}, \text{“}CAB\text{“}]$$

$$SCORE(\text{“}XYZ\text{“}) = \frac{score(\text{“}XYZ\text{“}) + score(\text{“}YZ\text{“}) + score(\text{“}Z\text{“})}{3} = $$

$$= \frac{(0.22)/1 + 0 + 0}{3} = 0.07$$

AST relevance score: Example 3



$$SCORE(\text{“}Alice\text{“}) = 0.32$$
$$SCORE(\text{“}Bob\text{“}) = 0.04$$

(Usually, SCORE > 0.2 is a strong evidence of relevance)

AST relevance score: alternatives & summary

- **Alternative solution:** count the number of occurrences of a keyword in the text colleciton
  - **Word-based** approach
  - **Requires at least normalization**, NLP involved
  - Can also use the **Levenstein distance** for more sensitivity
  - Relevance score definition & interpretation is not obvious
- **AST Relevance score:**
  - **Letter-based, "fuzzy"** approach
  - **Language-independent**, no NLP involved
  - **Interpretation**: *average conditional probability* of an occurrence of a single symbol of the input key phrase in the text collection

Table of Contents

Enhanced annotated suffix trees

- In the original papers, AST was represented as a **trie** $\Longrightarrow$ $O(n^2)$ time & space complexity.
- Even when implemented properly with suffix trees, the AST construction time & space usage still has a large **hidden constant** behind $O(n)$.
- We propose an **enhanced** implementation that uses **suffix arrays**.

From suffix tries to suffix trees

Ensure the linearity of our data structure:

- Suffix trie: one node per letter, $O(n^2)$ time & space
- Suffix tree: compacted edges, no chains, $O(n)$ time & space

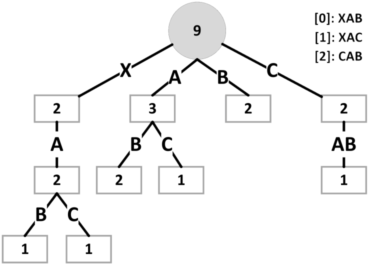


Figure: Suffix trie

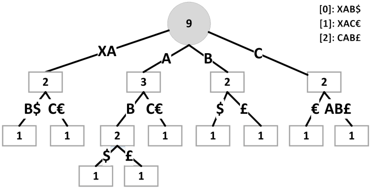


Figure: Suffix tree

From suffix tries to suffix trees

To construct *annotated suffix trees* in $O(n)$, simple preprocessing is needed:

Algorithm **LinearASTConstruction(C)**
Input. String collection $C = \{S_1, \ldots, S_m\}$
Output. Generalized annotated suffix tree for C.

1. Construct $C' = \{S_1\$_1, \ldots, S_m\$_m\}$, where $\$_i$ are unique characters that do not appear in $S_1 \ldots S_m$.
2. Construct a generalized suffix tree T for collection C' using a linear-time algorithm (e.g. the *Ukkonen algorithm*).
3. **for** l **in** *leaves*(T)
4.   **do** set $f(l) \leftarrow 1$
5. Run a postfix depth-first tree traversal on the suffix tree T. For each inner node v, set $f(v) \leftarrow \sum_{u \in children(v)} f(u)$.

From suffix tries to suffix trees

One minor change in the suffix relevance score:

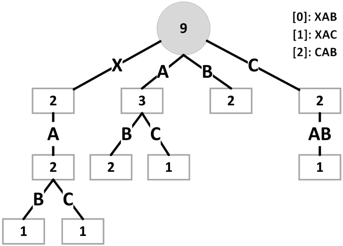


Figure: Suffix trie

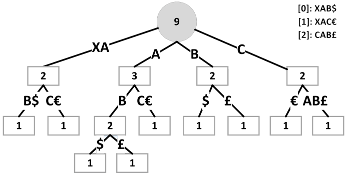


Figure: Suffix tree

$$score_{suff}(s) = \frac{\sum_{i=1}^{k} \hat{p}(v_i)}{k}$$

If l is the number of symbols in the match, then

$$score_{suff}(s) = \frac{\sum_{i=1}^{k} \hat{p}(v_i) + l - k}{l}$$

From suffix trees to suffix arrays

- **Suffix array** for a string S ($|S| = n$) is an array of n integer numbers, enumerating the n suffixes of S in lexicographic order.

Table: Suffix array for string “XABXAC“
(the suffixes are not actually stored)

i	*suffix array*	*S[suff[i]:]*
0	2	ABXAC
1	5	AC
2	3	BXAC
3	6	C
4	1	XABXAC
5	4	XAC

- Suffix arrays are more space efficient than suffix trees.

Enhanced suffix arrays

- Abouelhoda, Kurtz, & Ohlebusch [1] have shown that it is possible to systematically replace every algorithm that uses suffix trees with another one based on suffix arrays.
- Need to enhance the suffix array with two auxiliary arrays:
  - *lcp*-table for bottom-up traversal
  - *child*-table for top-down traversal
- Can be implemented to take no more than 10 bytes per input symbol (at least 20 for suffix trees)

Enhanced suffix arrays

Table: Enhanced suffix array for string "XABXAC"

i	*suffix array*	*lcp-table*	*child-table*			*S[suff[i]:]*
			1.	*2.*	*3.*	
0	1	0		1	2	ABXAC
1	4	1				AC
2	2	0	1		3	BXAC
3	5	0			4	C
4	0	0				XABXAC
5	3	2				XAC

Enhanced annotated suffix arrays

- We need to store annotations for suffix tree nodes
- The number of nodes in a suffix tree cannot exceed $(2n - 1)$
- After preprocessing, all the leaves will be annotated with 1, so there is no need to store these annotations explicitly
- We are left with at most $(n - 1)$ numbers to store $\implies$ can introduce one more auxiliary array of length n (*annotation*-table)

Enhanced annotated suffix arrays

Table: Enhanced annotated suffix array for string "XABXAC"

i	*suffix array*	*lcp-table*	*child-table*			*annotation*	*S[suff[i]:]*
			1.	*2.*	*3.*		
0	1	0		1	2	6	ABXAC
1	4	1				2	AC
2	2	0	1		3		BXAC
3	5	0			4		C
4	0	0					XABXAC
5	3	2				2	XAC

Enhanced annotated suffix arrays

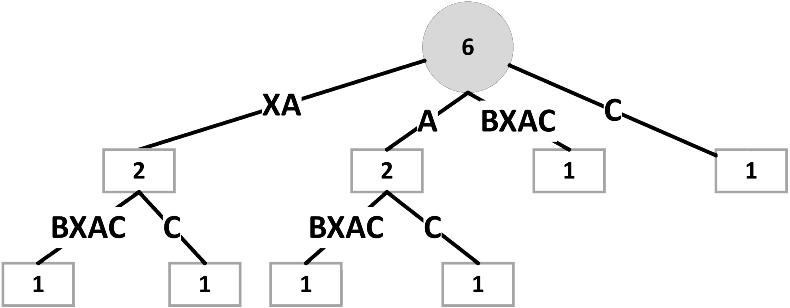


Figure: Annotated suffix tree for string "XABXAC"

Enhanced annotated suffix arrays

Node-to-array mapping – via **virtual lcp-trees**:

- Can be restored from the *lcp*-table
- Nodes correspond to the inner nodes of the suffix tree
- Nodes are represented as $\langle l, i, j \rangle$: the *lcp-value* l and the left and right boundaries of the *lcp-interval* (i, j)
- For each *lcp-interval* $v = \langle l, i, j \rangle$ there exists a unique index, $index(v) \in [0; n-1]$, which is equal to the smallest k, such that $k > i$ and $lcp[k] = l$. It is this mapping that we use to store the inner node frequency annotations.

Enhanced annotated suffix arrays

Algorithm **LinearEASAConstruction(C)**
Input. String collection $C = \{S_1, \ldots, S_m\}$
Output. Enhanced suffix array for C with substring frequency annotations.

1. Construct a string $S = S_1\$_1 + \cdots + S_m\$_m$, where $\$_i$ are unique termination symbols.
2. Construct a suffix array A for string S using a linear-time algorithm (e.g. the *Kärkkäinen-Sanders algorithm*) and two auxiliary arrays: *lcp-array* and *child-array*.
3. Simulate a postfix depth-first tree traversal on the suffix array A. At each of the *virtual inner nodes*, corresponding to an *lcp-interval* $v = \langle l, i, j \rangle$, where $i < j$, set $annotation[index(v)] = \sum_{u \in children(v)} annotation[index(u)] + \#(\langle l, i, j \rangle : i = j)$.

Enhanced annotated suffix arrays: Experimental results

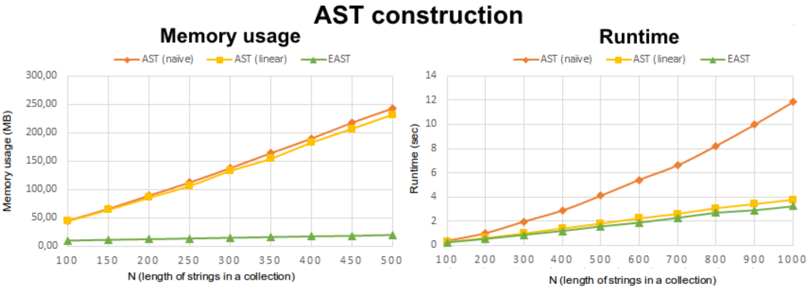


Figure: Experimental results

- Implementation: Python 2.7
- 10x less memory – due to suffix arrays + the *Numpy* library

Table of Contents

Package EAST

- EAST = *"Enhanced Annotated Suffix Trees"*
- Open-source:
  `https://github.com/msdubov/AST-text-analysis`
- Registered in Python Package Index and is easy to install:
  `$ pip install EAST`

Package EAST

- Provides command-line user interface:

```
$ east keyphrases table <keyphrases_list.txt>
  <path/to/the/text/collection/>
```

- Can be used as a Python library:

```
>>> from east.asts.base import AST
>>> ast = AST.get_ast([''XAB'', ''XAC'', ''CAB''])
>>> ast.score(''ABC'')
0.314814814814149
```

Synonym extraction

- **EAST** implements one laguage-dependent feature: **synonym extraction**
- **Motivation:** Relevance scores should be similar, say, for *"plant taxonomy"* and *"plant classification"*, even if the latter can be rarely found in the text collection.
- **Algorithm:** distributional synonym extraction algorithm based on that by Lin [4], which employs the so-called dependency triples (w_1, r, w_2) (idea: *similar texts appear in similar contexts*)
- **Domain-specific synonyms** are likely to be found with this context-based approach

Synonym extraction

- Dependency triples extraction is done by **Yandex Tomita parser** (based on grammatical templates like *"adjective + substantive"* or *"verb + arverb"*)
- Grammar:

```
S -> adj_mod_of interp (Relation.adj_mod_of::...) |
     adv_of interp (Relation.adv_of::norm="inf") |
     adv interp (Relation.adv::norm="inf") |
     ...

adj_mod_of -> Adj<gnc-agr[1]> Noun<gnc-agr[1]>;
adv_of -> Adv Verb;
adv -> Verb Adv;
...
```

Synonym extraction

- Synonyms extracted from a text collection from the "Izvestia" newspaper:
  - *"head" ("глава")* $\Leftrightarrow$ *"CEO" ("гендиректор")*
  - *"high" ("высокий")* $\Leftrightarrow$ *"low" ("низкий")*
  - ...
- Low precision is not very critical: among synonimous key phrases we chose the one that has $\max_{w \in syn(S)} SCORE(w)$
- To extract synonyms before computing relevance scores:

```
$ east -s keyphrases table <keyphrases_list.txt>
  <path/to/the/text/collection/>
```

Table of Contents

LM Monitor: Concept

LM Monitor = *"Latent Meaning Monitor"* [2]

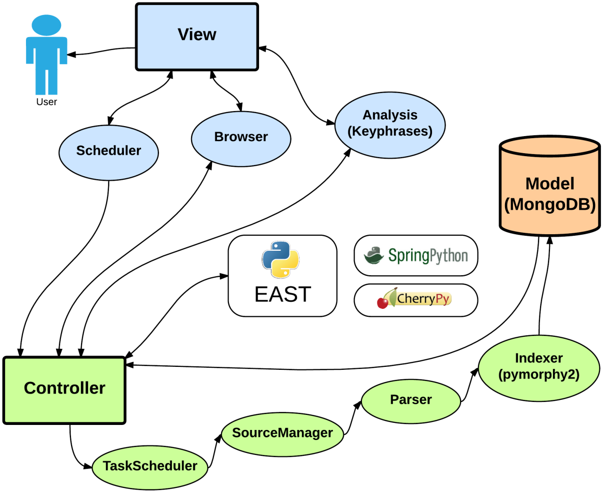
1. Web crawling
   - **RuNeWC:** Russian Newspaper Web Corpus
   - 5 sources available now
2. Keyphrase analysis
   - Keyphrases are provided by the user
   - Using the AST relevance scores for these keyphrases, a **keyphrase reference graph** is built
   - Text visualization tool

LM Monitor: Development

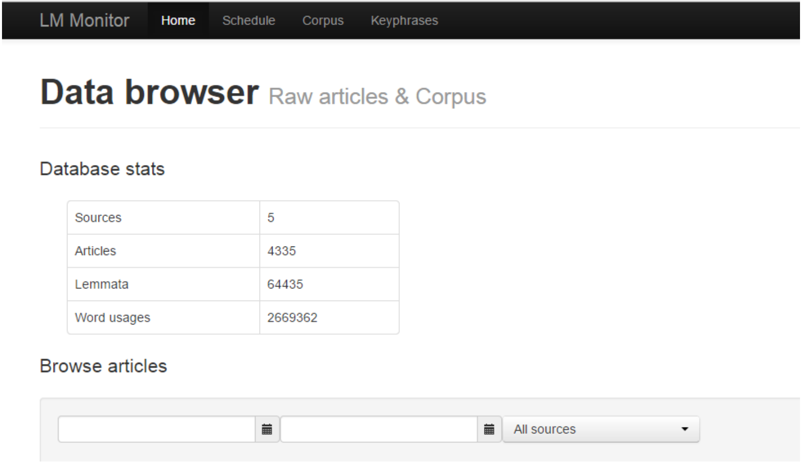
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ

1. Research group “Text analysis and visualization methods“
2. Head: Boris Mirkin (Sc.D, prof.)
3. Staff: Bachelor/Master/PhD students

LM Monitor: Architecture



LM Monitor: System

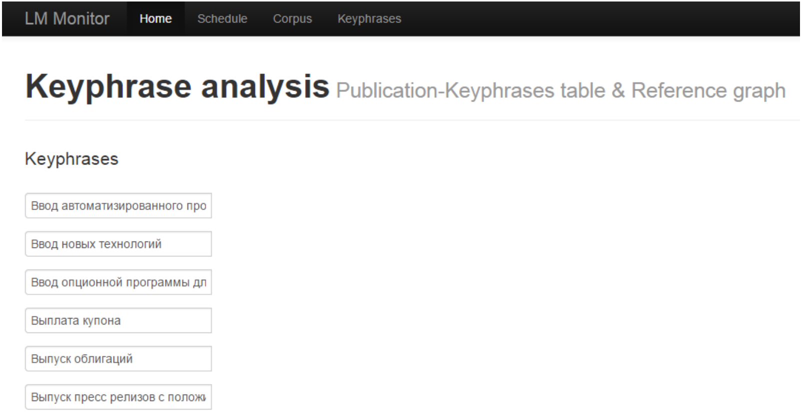


Keyphrase reference graphs

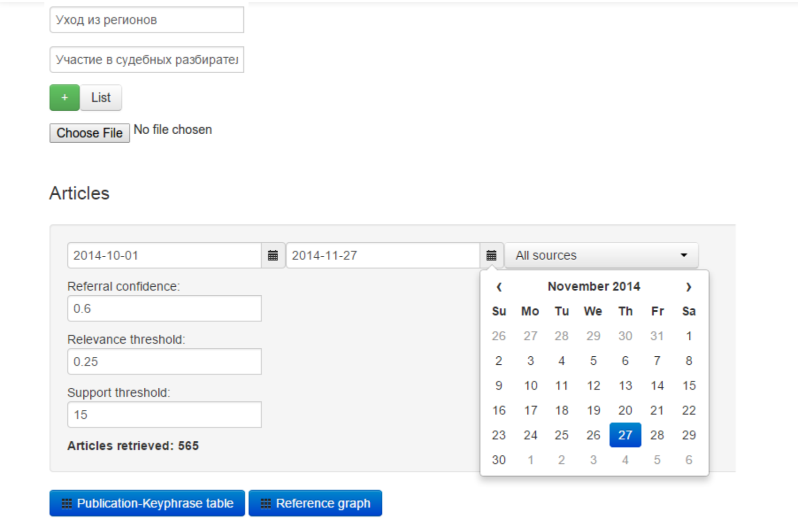
Keyphrase reference graphs:

- Model **directed relations** between keyphrases
- Nodes are keyphrases
- For a keyphrase A,
  - $r \in [0;1]$ is a *relevance threshold*: if $SCORE_{AST(T)}(A) > r$, then A is considered to be relevant to text T (usually $r = 0.2$)
  - $F(A) = \{T : SCORE_{AST(T)}(A) > r\}$
- $c \in [0;1]$ is a *confidence threshold*
- For keyphrases A and B, if $\frac{|F(B) \cap F(A)|}{|F(A)|} \geq c$, then there is an edge in the graph from keyphrase A to keyphrase B (usually $c = 0.6$)
- $A \rightarrow B$ is like an *associative rule*

LM Monitor: Keyphrase reference graphs



LM Monitor: Keyphrase reference graphs



LM Monitor: Keyphrase reference graphs

Id	Name
4	Выпуск облигаций
12	Новый CEO генеральный директор
21	Погашение кредита
22	Погашение облигаций
23	Присвоение кредитного рейтинга
29	Публикация финансовой отчетности
34	Смена генерального директора
35	Смена финансового директора

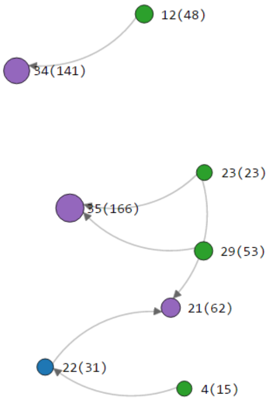
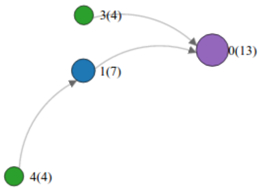


Figure: Keyphrase reference graph built for Oct/Nov 2014

LM Monitor: Keyphrase reference graphs

Id	Name
0	Congress of the United States
1	President of the United States
3	Legislative Power
4	Executive Power



Id	Name
0	Le Président de la République
1	Le Gouvernement
2	Le Parlement
3	Le Conseil constitutionnel

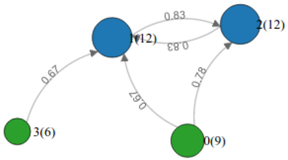


Figure: Keyphrase reference graph built USA/France constitutions

Future work

- Automated graph analysis (central nodes visualization etc.)
- Temporal graph analysis (how do graphs change over time?)
- Better support for synonyms

Abouelhoda, M. I. Replacing Suffix Trees with Enhanced Suffix Arrays / M. I. Abouelhoda, S. Kurtz, E. Ohlebusch // Journal of Discrete Algorithms, Amsterdam: Elsevier. – 2004. – № 2. – pp. 53-86.

Dubov, M. Automatic Russian Text Processing System / M. Dubov, B. Mirkin, A. Shal // Open Systems. DBMS – 2014. – v. 22 № 10 – pp. 15-17

Gusfield, D. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology / D. Gusfield. – Cambridge University Press, 1997.

Lin, D. Automatic Retrieval and Clustering of Similar Words. / D. Lin // Proceedings of the 17th International Conference on Computational Linguistics. – 1998. – pp. 768-774.

Mirkin, B. Method of annotated suffix tree for scoring the extent of presence of a string in text / B. Mirkin, E. Chernyak, O. Chugunova // Business-Informatics – 2012. – № 3(21). – pp. 31-41.