IMPLEMENTATION OF DICTIONARY LOOKUP AUTOMATA FOR UNL ANALYSIS AND GENERATION

Igor Zaslawskiy, Aram Avetisyan, Vardan Gevorgyan

***Abstract**: In this paper we present some research results and propose solutions for natural language string lookup techniques. In particular a fast method is suggested for searching dictionary entries for possible matches of sentence words without using relational databases or full dictionary load into machine random access memory. Such approach is essential for minimizing the speed dependency from dictionary size and available machine resources as well as for the scalability of the analyzer software. The mentioned is based on an implementation of Aho-Corasick [Aho, Corasick, 1977] automata with a number of optimizations in the indexing and lookup algorithm.*

***Keywords**: UNL, natural language processing, dictionary lookup, indexing, search, XML, pattern matching machine, string matching algorithm, information search*

***ACM Classification Keywords**: F.2.2 Non-numerical Algorithms and Problems – Pattern matching, Sorting and searching, I.2.7 Natural Language Processing - Text analysis, Language parsing and understanding, I.7.2 Document Preparation – Index generation, Markup languages, H.3.1 Content Analysis and Indexing – Dictionaries, Indexing Methods, H.3.3 Information Search and Retrieval - Retrieval models, Search process.*

Introduction

UNL (Universal Networking Language) is a meta-language representing semantic information *[Uchida, Zhu, 2005]*. Its main purpose is to store "the meaning" of natural language texts in a language independent format. Each sentence in UNL is a directed linked graph. Unlike natural languages, UNL expressions are less ambiguous. In the UNL semantic networks, nodes represent concepts, and arcs represent relations between concepts. These concepts are called "Universal words" (UWs). The UWs' connections are called "relations". They specify the role of each word in the sentence *[Uchida, Zhu, 2005].*

Many UNL centers and the UNDL Foundation itself have created a number of tools for working with UNL and Natural Language (NL) resources. In this paper we are highlighting some aspects of the development of tools for natural language text analysis and generation. The core tools of the project are the NL Analyzer and NL Generator developed by the UNDL Foundation *[Uchida, Zhu, 2005].*

The aim of the paper

The NL analyzer uses several types of resources to build semantic UNL graphs of NL sentences: NL-UNL dictionary, transformation and disambiguation rule sets. These resources are being used for semi or fully

automatic transformation during the analysis process. Many tools were created for working with UNL-NL and NL-UNL dictionaries, rules and UNL documents, however in the work process of mentioned tools some difficulties arise connected with the increase of the volumes of resources. As the amount of data is growing, the queries to relational databases require more time to return. In this paper we present a specific dictionary lookup tool which was built based on some search algorithms tuned to match specificity of UNL resources and to get better performance.

Finite-state Machine Implementations

Nowadays the finite-state machine implementation algorithms for string lookup are considered giving comparatively effective solutions for the better lookup performance. In particular machines based on algorithms like Boyer–Moore, Knuth–Morris–Pratt and Aho–Corasick are considered, they are widely used for string lookup purposes.

One of the most efficient and known string lookup algorithms is the Knuth–Morris–Pratt string searching algorithm (KMP) *[Knuth, Morris, Pratt, 1977]*. This algorithm searches for occurrences of strings in text. The interesting point in this algorithm is that when a mismatch occurs, the word itself embodies information to determine where the next match can begin, thus bypassing re-examination of previously matched characters.

Another efficient solution is the Aho–Corasick string matching algorithm invented by Alfred V. Aho and Margaret J. Corasick. It has been proved that the running time of the Aho-Corasick Algorithm (ACA), where m is the length of the text T, n is the total (cumulative) length of all patterns in P, and k is the total number of matches of P in T, is $O(n + m + k)$. If we compare the work of ACA with the native exact matching algorithm then allowing that T is searched once for each of the z patterns in P and a single search can run in $O(m)$ time, there are z iterations of T, and $O(n)$ amount of work spent looking at the patterns. This results in a total running time of $O(n + mz)$, which is significant amount of time compared to the linear search time of the ACA. Clearly, the ACA is more efficient than naive exact set matching algorithms *[Spreen, Van Slyke, 2010]*.

Similar to KMP algorithm Boyer-Moore(BM) string matching algorithm is widely used in string lookup purposes. The execution time of the BM algorithm, while still linear in the size of the string being searched, can have a significantly lower constant factor than many other search algorithms: it doesn't need to check every character of the string to be searched, but rather skips over some of them. Generally the algorithm gets faster as the key being searched for becomes longer. Its efficiency derives from the fact that with each unsuccessful attempt to find a match between the search string and the text it's searching, it uses the information gained from that attempt to rule out as many positions of the text as possible where the string cannot match. In 1991 it was proved that the BM time complexity O(m) can have in worst case $3m$ comparisons, while in best case only m/n comparisons *[Boyer, Moore, 1977]*.

However it must be noted that both Boyer–Moore and Knuth–Morris–Pratt string search algorithms perform lookup in plain text strings, while Aho-Corasick algorithm is capable to search in text documents represented as lists of strings (which in our case can be the NL-UNL Dictionary). Thus the ACA implementation was chosen as a more suitable solution for NL-UNL Dictionary lookup.

Aho-Corasick Automaton

Let $D = \{y_1, y_2, \dots, y_k\}$ be a finite set of strings that we shall call ***words*** which will be the headwords of dictionary entries that will be included in the pattern matching machine. And let x be an arbitrary string that we shall call ***text string*** (natural language sentence). Our problem is to find all the substrings of ***text string*** which are ***words*** in ***D***. Substrings may overlap or include each other.

The pattern matching machine for ***D*** is a machine that takes as input the ***text string*** and returns all occurrences of text string substrings that are ***words*** in ***D***. Thus, if we create a pattern matching machine for a dictionary and give to its input a natural language sentence, we shall receive all the dictionary entries that the NL Analyzer may need for the further UNL generation. *[Aho, Corasick, 1977]*

NL-UNL Dictionary structure

Like the conventional dictionaries, NL-UNL dictionaries have simple structure, e.g. [HEADWORD]{UW ID} "UW" (ATTRIBUTES) <L,P,F>; (more simplified : [HEADWORD]<WORD DESCRIPTION >). *[Uchida, Zhu, 2005][UNDL, 2007]*

In NL Analysis the keywords for lookup can be any combinations of letters, digits and special characters, which means that the string matching machine must be constructed to provide maximum speed for searching any type of character combinations. For compilation we created a custom XML-like syntax, the example below illustrates the compiled dictionary syntax.

```
<root>
    <_h_00000182>
        <_e_00000164>
            <e>[he]{UW ID} "UW" (ATTRIBUES) <L,P,F>;</e>
            <_r_00000106>
                <e>[her]{UW ID} "UW" (ATTRIBUES) <L,P,F>;</e>
                <_s_00000047>
                    <e>[hers]{UW ID} "UW" (ATTRIBUES) <L,P,F>;</e>
                </_s>
            </_r>
        </_e>
    </_h>
    <_s_00000082>
        <_h_00000064>
            <_e_00000046>
                <e>[she]{UW ID} "UW" (ATTRIBUES) <L,P,F>;</e>
            </_e>
        </_h>
    </_s>
    <_u_00000222>
        <_s_00000204>
            <e>[us]{UW ID} "UW" (ATTRIBUES) <L,P,F>;</e>
            <_h_00000146>
                <_e_00000128>
                    <_r_00000110>
                        <e>[usher]{UW ID} "UW" (ATTRIBUES) <L,P,F>;</e>
                        <_s_00000049>
                            <e>[ushers]{UW ID} "UW" (ATTRIBUES) <L,P,F>;</e>
                        </_s>
                    </_r>
                </_e>
            </_h>
        </_s>
    </_u>
</root>
```

Example 1.

Example 1 illustrates a finite state string matching machine compiled from a dictionary containing entries:

[he] {UW ID} "UW" (ATTRIBUTES) <L,P,F>;

[her] {UW ID} "UW" (ATTRIBUTES) <L,P,F>;

[hers] {UW ID} "UW" (ATTRIBUTES) <L,P,F>;

[she] {UW ID} "UW" (ATTRIBUTES) <L,P,F>;

[us] {UW ID} "UW" (ATTRIBUTES) <L,P,F>;

[usher] {UW ID} "UW" (ATTRIBUTES) <L,P,F>;

[ushers] {UW ID} "UW" (ATTRIBUTES) <L,P,F>;

There are several reasons why we use “XML-like” syntax instead of valid XML. The first reason is that a valid XML document cannot contain special characters or digits as node elements (e.g. < ’ > or < 1 > are invalid). The second reason is that in a valid XML document the opening and closing tag contents should be identical, while in our document each opening tag contains an eight digit number describing the count of characters inside the tag; this number enhances the keyword lookup speed by allowing the program to identify the position of the closing tag. It is unreasonable to keep that number in closing tag, because it makes the document larger.

ACA Construction

The first step of ACA implementation is the creation of the Data Object Model (DOM); that is a tree containing all the dictionary entries in a structure described below (*Fig1*). The second step is the exporting of DOM into a “XML-like” text form file (*Example 1*).

DOM constructing algorithm (DOMCA) iterates through the dictionary entries fed in a text file or database. All entries are being inserted into the same DOM by the *DOMCA*. During the process *DOMCA* receives a new entry and separates the headword from the whole entry string, splits the headword into characters and starting from the root element of the current DOM inserts the characters into the tree. If the character of headword has not been inserted into that level of the tree in previous steps, a new node object representing that character is being inserted into that level. After that the pointer shifts to the next character of the headword and repeats same steps considering as a start level element the previous character node that was inserted. If the node already exists, nothing is being inserted, the program only shifts the pointer to the next character of the headword and repeats the same steps considering as a start level element the previous character node that was found inserted in previous level. The program repeats these steps recursively until the pointer reaches the end of the headword and after that inserts a new node containing information about the whole entry string. As a result, the final output will be a single DOM tree with characters as intermediate nodes and dictionary entries as final nodes (*Fig1*).

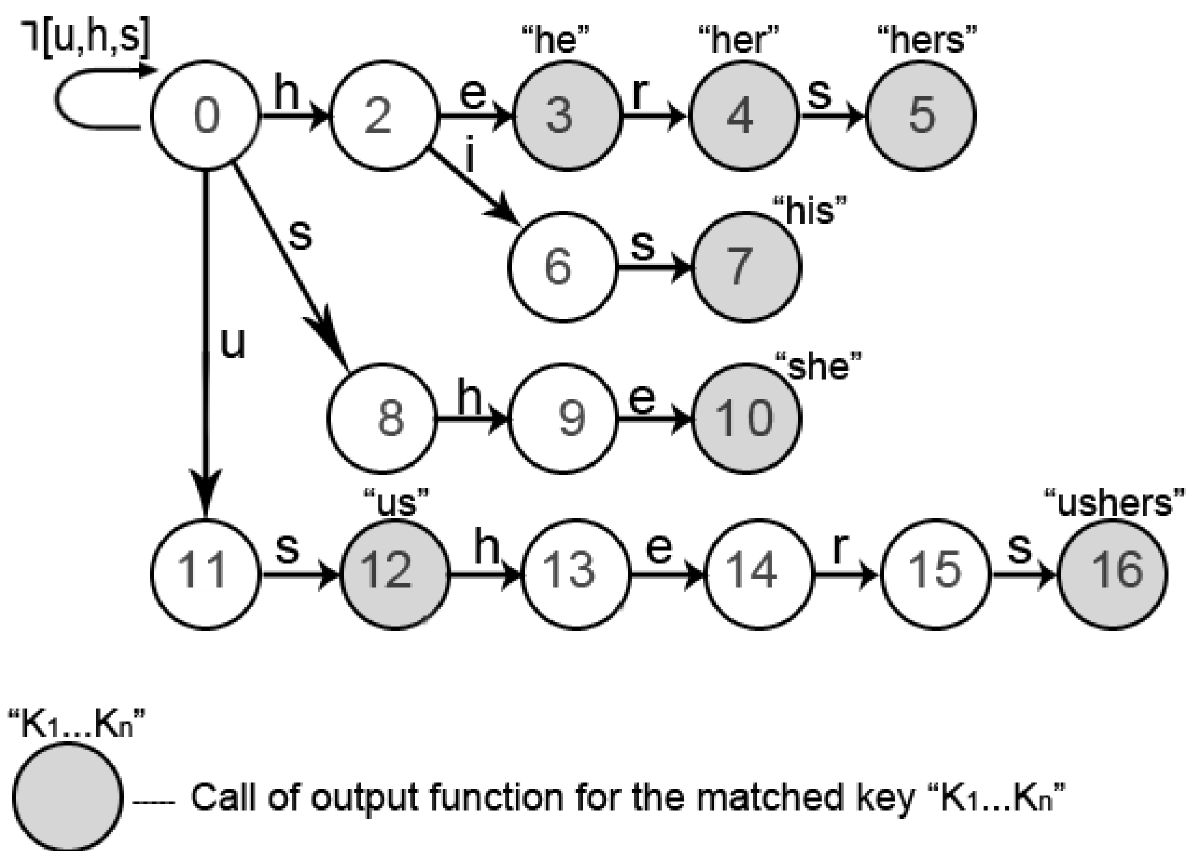


Fig1

.

Exporting DOM into a Text Document

After the pattern matching machine has been created and presented as a DOM, we need to export it into a text document for storing and further usage. We use a function that receives a DOM node element to its input and returns its content as a string including the string values of sub-nodes which in their turn are presented as text string by calling the same function recursively until there are no sub-level nodes left. The final string is an XML-like document (*Example 1*) that represents the DOM (*Fig1*) in a final text form, which will be later used in keyword lookup process.

Dictionary Lookup Using the ACA

String lookup algorithm is applied to a text string ***T*** and a dictionary DOM which is constructed as a directed graph described above.

Let ***T*** be a word ***T*** = $\sigma_1, \sigma_2, \ldots, \sigma_L$. We define a *goto function* giving for every state *k* and every letter *σ* a new state ***g****(k, σ)* obtained by applying the letter *σ* (graph edge) in the directed graph DOM to the state *k* (graph node). For example in *Fig. 1*, the edges labeled from 0 to 11 indicate that ***g****(0, u)* = 11, ***g****(0, s)* = 8 and so on. Whenever there is no edge *σ* for the node k in the graph the *goto* function *g*(*k*, *σ*) reports *fail* result.

In our Machine we also define an ***output*** function returning the dictionary entries that match the word generated by the current state (may be empty).

At the beginning the string lookup algorithm considers the words $g(\ldots g(g(0, \sigma_1), \sigma_2)\ldots,\sigma_k)$ such that $k \leq L$ while the symbol *fail* does not arise during the process. Every time when the function ***output***(s_k) ≠ *empty* we add its returned entry to the output list of entries. If the failure symbol *fail* has appeared during the word lookup process $g(\ldots g(g(0, \sigma_1), \sigma_2)\ldots,\sigma_L)$ the lookup process for the word ***T*** = $\sigma_1, \sigma_2, \ldots, \sigma_L$ stops and starts again considering as the lookup word ***T*** = $\sigma_2, \ldots, \sigma_L$, and so on. The list of all output entries obtained during these iterations will be the final output of the String Lookup Algorithm.

Initially, the current state of the machine is the start state ***0*** and the first symbol of the keyword string is the current input symbol. The machine then processes the keyword string by making one operating cycle on each symbol of the keyword string. *[Aho, Corasick, 1977]*

Development of Dictionary Lookup Program Using ACA

The text file of the created pattern matching machine may be very large depending on dictionary size, thus, instead of loading the whole document into the machine’s memory, the program reads the document bytes one by one.

Let us assume that we have a keyword *w* and need to find all the dictionary entries with headwords that match as a whole word or as substring to the *w*.

The lookup program receives the keyword *w* and splits it into character array $K = \{ k_1, k_2, \ldots , k_n \}$. On each iteration of $i = \{1, 2, \ldots , n\}$ loop the program searches for an opening tag “<_k_i>” or “<e>” (“<e>” tag indicates a dictionary entry or entry set and is different from “<_e>” tag) . When an opening tag “<e>” is found the program calls the ***output(****“$k_1k_2\ldots k_{i-1}$”****)*** function to print the content of the “<e>…</e>” tag which is a match for the keyword substring “$k_1k_2\ldots k_{i-1}$”, and continues search, if a tag “<_k_i>” is found, meaning that the *i*-th character of the keyword has matched, the next character of the keyword becomes the current character ($i = i+1$) and the operations repeat (e.g. for the keyword “ushers”, if the “<e>” was found in tag representing the second symbol of the keyword, ***output(****“us”****)*** function will be called and the found entries in <e> tag will be printed as the matches for a sub-keyword “us”). If no tag “<_k_i>” is found on the i-th step, the iteration stops and returns the printed values.

After each circle while $|w| > 1$, we remove the first character of *k* keyword and run lookup program with the new keyword *w’* = *“$k_2k_3\ldots k_n$”* again.

For example if the base keyword is "ushers", the keywords that will be sent to the lookup program input are {*w* = "ushers", *w'* = "shers", *w''* = "hers", *w'''* = "ers", *w''''* = "rs", *w'''''* = "s"}. See *Table 1* for the returned results after each call of lookup program.

Keyword	Matches
"ushers"	**"us", "usher", "ushers"**
"shers"	**"she"**
"hers"	**"he", "her", "hers"**
"ers"	**-**
"rs"	**-**
"s"	**-**

Table 1.

Lookup Time Optimization

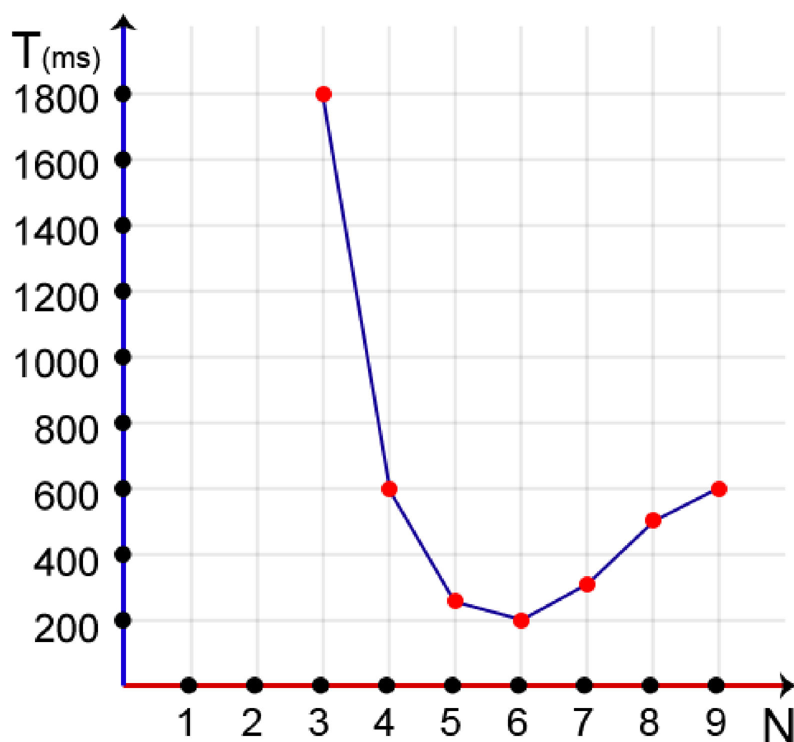
Several enhancements were done in order to minimize the time required for lookup.

1. It was decided to create a mapping file that will contain the byte addresses of the first 6 characters of all headwords of dictionary entries used in pattern matching machine creation. Thus for a keyword $K = \{ k_1, k_2, \ldots ..., k_n \}$ $(n>6)$, the program can find the byte indexes for sub-keywords {"k_1" , "k_1k_2" , "$k_1k_2k_3$" , "$k_1k_2k_3 k_4$" , "$k_1k_2k_3k_4k_5$" , "$k_1k_2k_3k_4 k_5 k_6$"} from the mapping file and retrieve all dictionary entries that may be allocated in those byte addresses of the dictionary file. After that the program starts the dictionary lookup starting from the "$k_1k_2k_3k_4k_5k_6$" index position with a keyword "$k_7k_8 \ldots k_n$". If $n \le 6$, no lookup will be called at all, this significantly minimizes the time required for process. The limit of 6 characters was chosen for mapping as the most optimal number in mapping file size and lookup time relation. By increasing the length of mapped keyword strings more time is required for loading the mapping file, less time for lookup and vice versa.

   Graph 1 illustrates the efficiency of lookup algorithm in required time in milliseconds (T) and mapping characters limit (N) relation. This graph is based on the average of test results held on a PC with dual core CPU of 2.0 GHz frequency and 2 GB of RAM. 10 sentences in English of average 180-200 characters each were chosen as keyword strings for testing with English-UNL dictionaries containing 100'000 to 400'000 entries. According to the graph, mapping character limit of 6 is the optimal number producing the best speed results.

2. When exporting the dictionary string matching machine into a text file, all character nodes are being ordered alphabetically. During the lookup process, when a character tag is found but does not match the

current keyword character, the program is supposed to find the next opening tag in that level. But considering the fact that the tags are stored in alphabetical order, the program compares the byte codes of the current keyword character and the found tag character, if the tag character code is less than the keyword character code, we can be sure that there will be no further character tags in that level of tree that will match the current character of keyword, thus the failure function is being called, avoiding the further lookup actions that will not return any match. Also to ensure that the program will not miss any entry tag by performing this action, the entry tags are being placed before all character tags of each level of the tree.



Graph 1.

Results and Conclusion

As it was mentioned before there are other alternative ways of dictionary lookup to the string matching machines and those methods were also tested and compared to the implemented ACA results. One of the most robust search engines nowadays is the Apache Lucene: an open source project created in Java and currently being released under the Apache Software License. Currently, Lucene is considered of the leading tools for search and is being used as a basis in many powerful search engines. It uses similar search indexing and lookup approaches, thus was chosen for our comparison. *Table 2* illustrates the lookup results gained by using MySQL

database, Apache Lucine and our ACA implementation. Below are listed 5 randomly selected sentences that were used in comparison results (*Table 2*). These results were later confirmed by a bigger test corpus.

1. “The flexibility and opportunities that UNL gives are enormous, so we decided to create a project that will be a pioneering effort to invest on this technology into real world applications.”

2. “The flexibility and opportunities that UNL gives are enormous.”

3. “The flexibility and opportunities.”

4. “Cyanogenmod is free of charge, but let's face it - it takes time and effort from Cyanogen to make it happen, time he could be using to work a salaried position, but instead is working on getting you the ROM you love, and doing it without asking anything in return.”

5. “Depending on the current state of your handset, there are basically three different ways to upgrade to the latest CyanogenMod version.”

Sentence No.	Matched entries	ACA lookup time	Lucine lookup time	MySQL lookup time
1	111	355 ms	1093 ms	2512 ms
2	55	154 ms	152 ms	215 ms
3	36	42 ms	50 ms	162 ms
4	120	131 ms	789 ms	7300 ms
5	86	90 ms	149 ms	2048 ms

Table 2.

Thus, we can say that the implemented search engine performance is robust and resolves the performance issues on standard desktop machines.

Bibliography.

[Uchida, Zhu, 2005] Uchida H., Zhu M., “The Universal Networking Language (UNL) specifications” version 7, UNDL Foundation, June 2005.

[Avetisyan, 2009] Avetisyan A., “Some approaches to the generation of sentences in natural language from UNL” Institute of Informatics and Automation Problems of NAS of RA, 2009.

[Aho, Corasick, 1977] Alfred V. Aho, Margaret J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search” Communications of the ACM, 1977.

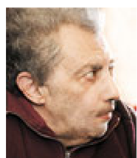
[Knuth, Morris, Pratt, 1977] D. Knuth, James H. Morris Jr, V. Pratt, "Fast pattern matching in strings", SIAM Journal on Computing, 1977.

[Spreen, Van Slyke, 2010] T. Spreen, A. Van Slyke, "Exact-Set Matching with the Aho-Corasick Automaton" University of Victoria, 2010.

[Boyer, Moore, 1977] R. S. Boyer, J. S. Moore, "A Fast String Searching Algorithm" Communications of the ACM, 1977.

[UNDL, 2007] UNDL Foundation, www.unlweb.net/wiki/, 2010

Authors' Information



Igor Zaslawskiy *– Head of lab., Chief sci. worker, Institute of Informatics and Automation Problems of NAS RA, 1, P. Sevak str., Yerevan, Armenia, 0014, e-mail: zaslav@ipia.sci.am,*

Major Fields of Scientific Research: Mathematical logic and the automated logic conclusion

Aram Avetisyan *– Junior sci. worker, Institute of Informatics and Automation Problems of NAS RA, 1, P. Sevak str., Yerevan, Armenia, 0014, e-mail: a.avetisyan@undlfoundation.org*

Major Fields of Scientific Research: Mathematical logic and the automated logic conclusion

Vardan Gevorgyan *– CEO, "VTGSoftware" Ltd., 52a, Sose str., Yerevan, Armenia 0019,*

e-mail: vgevorgyan@vtgsoftware.com