Condensation principle[1]

Miroslav Balík

Department of Computer Science & Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Praha 2
Czech Republic

e-mail: balikm@fel.cvut.cz

Abstract. The most important contribution of this paper is the discovery and description of an effective implementation of a finite automaton that accepts all substrings of a given text. The ratio of the size of this automaton with respect to the size of the input text is in standard implementations 10 or more, see [AndNil95], in [Bal98] is described implementation with ratio less than 4. This implementation does not increase the time to search for a pattern, which is proportional to the length of the pattern.

The major approach introduced in this paper uses transformation of a text into a new alphabet that contains more symbols than the original alphabet and decreases text size. It operates over alphabets that contain small number of symbols. Such an automaton is suitable for example for processing of DNA sequences.

A new automaton and new search algorithms are presented for such a transformed text (patterns are still input in the original alphabet). Space requirements of such condensed automata are as low as the size of the input text.

This paper further introduces a new type of an automaton, a so-called $Identifier\ DAWG$, which uses special properties of the $DAWG$ automata and further decreases space requirements by introducing associations between states of an automaton and positions in a text. It is designed better searching algorithm using this association.

Key words: DAWG, Suffix DAWG, SDAWG, automata implementation, Suffix tree, finite automata.

1 Introduction

Let T be a text over a fixed alphabet Σ. Then an automaton can be created in a linear time (see [Cro94]) that accepts all *substrings* that occur in text T. This automaton is called DAWG (Directed Acyclic Word Graph) and its variant accepting only suffixes is called Suffix DAWG (SDAWG).

[1]This work was supported by grant GACR No. 201/98/1155.

Although some search automata have a linear size with respect to the length of the text, this size is high enough to disable practical implementation and usage. This size depends on implementation details, on the structure of the text and on the type of the automaton used. For *Suffix tree* the size is rarely smaler than $10n$ bytes, where n is the length of the text. Other types of automata are usually smaller, *suffix arrays* [GoBaSn91] (size $5n$ bytes), level compressed tries [AndNil95] (size about $11n$ bytes), *suffix cactuses* - a cross between suffix tree and suffix array [Kark95] (size $9n$ bytes), and *suffix binary search trees*[Irv95] (size about $10n$ bytes). Paper [Bal98] describes the method of implementation, which decreases the ratio to *4:1* .

This paper shows the new method, how to decrease this ratio using special transformation of the text. This transformation called condensation desreases text size. DAWG automaton constructed over transformed text saves good properties of DAWG, i.e. linear time and space comlexity for construction with respect to the size of text and linear time comlexity for matching a pattern with respect to its length.

2 DAWG, SDAWG

$DAWG(T)$ is a minimal automaton that accepts all substrings of a text T. An automaton that accepts all suffixes of a text T is denoted as $SDAWG(T)$. An algorithm of an incremental creation SDAWG automaton is described in [Cro94].

It holds that the number of states of the automaton $SDAWG(T)$ is greater or equal to the number of states of $DAWG(T)$. If we move all states of the automaton $SDAWG(T)$ to the set of final states, we could obtain a couples of equivalent states. For example, $DAWG(aabab)$ has six states, while $SDAWG(aabab)$ has seven states and two of them are final, see figure 1.

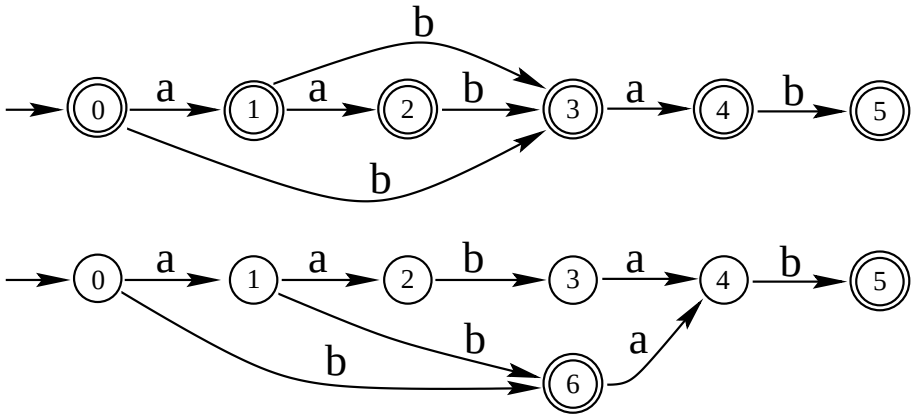

Figure 1: $DAWG(aabab)$ versus $SDAWG(aabab)$.

Automaton $DAWG(T)$ can be simulated by automaton $SDAWG(T)$, the reverse is not possible without an additional information.

3 Identifier Trie, Identifier DAWG

An *identifier trie* is modification of an automaton that accepts a set of patterns $Suff(T)$. A pointer to the text is added to some states. There are states where there

is only one sequence of transitions leading to some final state. Labelling of such a sequence is a suffix of a text T and it can be represented by a pointer to the text, which marks the beginning of this suffix. This pointer will replace all transitions to the first state of such a sequence.

An example of an $identifier\ trie(acagac\#)$ using new type of states is shown in Figure 2. New type of states are drawn as squares. The number in this square denotes starting position of corresponding suffix in the text. For example states denoted as 3' and 3 point to the suffix $gac\#$.

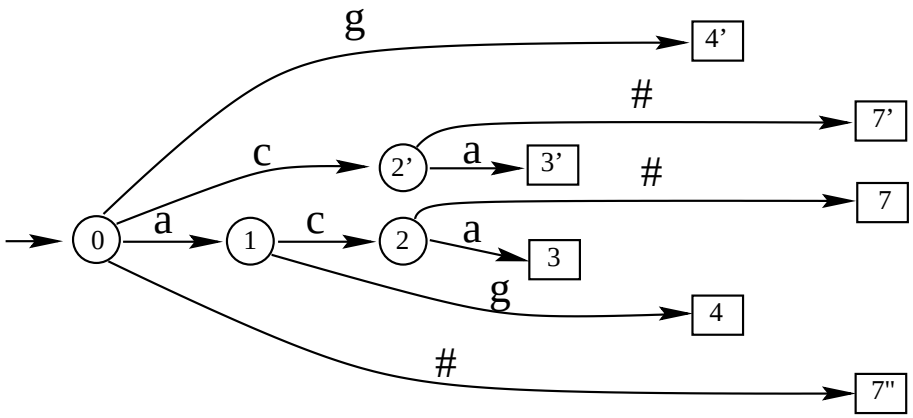

Figure 2: An $identifier\ trie(acagac\#)$ using new type of states.

Another approach does not create a special category for transitions, but it creates a new type of states. Because each state of a *suffix trie* corresponds to some suffix of a text T, we can link this suffix and its corresponding state with the position of its beginning in the text T, thus we create a new type of states that represent a reference to the text and that will act in places where we have used special transitions in the first case.

An example of an $identifier\ trie(acagac\#)$ using new type of transitions is shown in Figure 3. New type of transitions are without labeling and point to sguares representing positions in text. Each of these numbers denotes the starting position of the corresponding suffix in the text. For example, positions denoted as 3' and 3 point to the suffix $gac\#$.

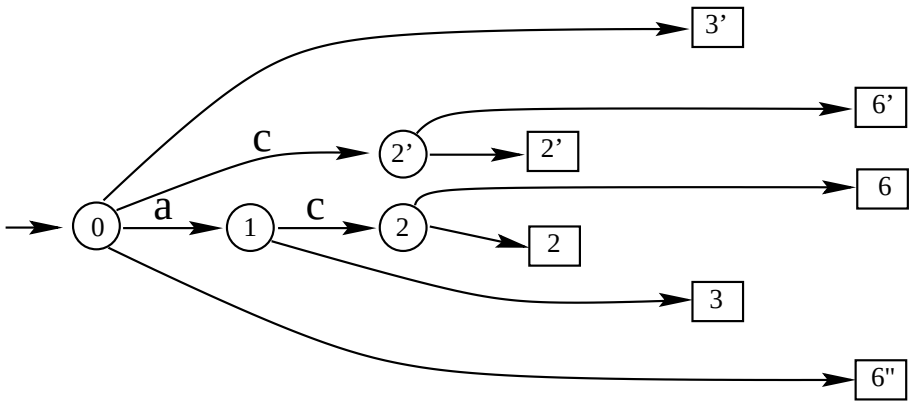

Figure 3: An $identifier\ trie(acagac\#)$ using new type of transitionss.

We can minimize such an automaton. We denote it as an *identifier DAWG* denoted as $IDAWG$. An example of an $IDAWG(acagac\#)$ is shown in Figure 4.

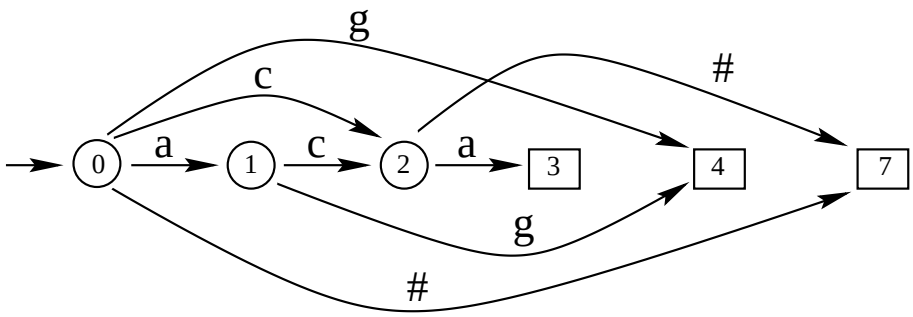

Figure 4: $IDAWG$ for the text $T = acagac\#$

The new type of an automaton presented in this section requires a modification of a standard matching algorithm. When it encounters a reference pointing to the text, it must check whether the rest of the pattern is the same as the suffix of the text being pointed to. The moment when it transfers to the text corresponds to a prefix having been matched with some sistring. If string matching terminates sooner, the string is a prefix of several sistrings, so it is a substring of a text T, the number of its occurences in the text is greater than one and it corresponds to the number of sistrings it is a prefix of. For a clear definition of sistrings it is necessary to concatenate the text with the symbol # that denotes the end. A more detailed description of the matching algorithm is shown in the following example.

Let $T = acagac$ be a text. $IDAWG(T\#)$ is shown in Figure 4. We will show the process of matching a input string (pattern) on two examples:

1. $P_1 = caga$. We start at the initial state q_0 denoted as 0. We match symbol c and we move to state 2. Then we match symbol a and we move to state 3 and the continuation of matching is in the text at position 3. Since now we are matching the suffix of the text gac with the suffix of the pattern ga. As ga is the prefix of gac, pattern P_1 is the substring of a text T. Because we have been matching the text and it has been done successfully, the number of occurences of P_1 is equal to one. If the rest of the text (here ga) is the same as the suffix of the text (here it is not, suffix is gac), the pattern P would be the suffix of the text T. The position of the pattern in the text corresponds to the position of the symbol in the text that was matched with the last symbol of the pattern decreased by the length of the pattern - 1, i.e., $Pos(P_1, T) = Pos(caga, acagac) = \{5-4\} = \{1\}$.

2. $P_2 = ac$. We start in the initial state q_0. We match symbols a and c and we move to state 2. As we have matched the whole pattern and in the same time we have not encountered a reference to the text, the number of occurences of this pattern is greater than one. This number corresponds to the number of sistrings whose prefix is formed by the pattern, i.e., to the number of paths leading from the last visited state (here 2) to the text (here the transition a and the second path is a transition #). If there is a transition leading from the last visited state (here 2) labeled by the symbol # (here it is), then the given pattern is a suffix

of text T. The position of the pattern corresponds to the beginning of sistrings that they are a prefix of, in this case $Pos(P_2, T) = Pos(ac,\ acagac) = \{0, 4\}$.

We see that the automaton $IDAWG(T\#)$ is suitable to find out whether a pattern P is a substring of a text T as well as it is its suffix. It is not necessary to create another type of automaton representing suffixes of the text. It is suitable to determine the number of repetitions of the pattern in the text together with the positions of all its occurences.

As it is not important if the information about the reference to the text will be represented as a special transition or as a new type of state, we will further consider the case with states that have an additional information - a reference to the text. $IDAWG(T\#)$ does not contain final states, thus we could declare that final states correspond to accepting some sistring.

The simplest way of constructing an identifier trie is deletion of redundant states and transitions from a *suffix trie*. This construction is following: Each state whose right language contains just one word (there is only one labelling on the path to a final state) can be excluded. All transitions from excluded states are redundant (these are transitions between non-existing states), thus they can be excluded. Transitions to excluded states from non-excluded states are replaced by references to the text (using new type of transitions or states). If we apply this way of exclusion and replacement of redundant states to a minimized *suffix trie*(T), i.e., $SDAWG(T)$, we create a minimized type of an identifier trie, $IDAWG$. This automaton has significantly better properties than an identifier tree, the number of its states is linearly proportional to the length of the input text, in contrast to a quadratic complexity of the non-minimized version.

4 Text transformation

Let T be a text of length n over some alphabet Σ. A transformed text with a coefficient of condensation k, T^k is constructed so that we group the input text by k symbols and we replace each such a group by one symbol of a new alphabet Σ^4. An example of such a transformation for $k = 4$ is shown in Figure 5.

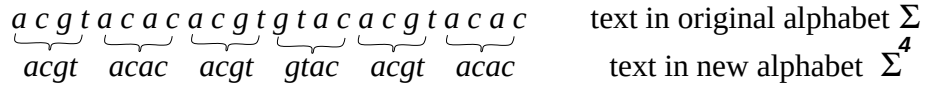

Figure 5: Transformation of a text $T = a\ c\ g\ t\ a\ c\ a\ c\ a\ c\ g\ t\ g\ t\ a\ c\ a\ c\ g\ t\ a\ c\ a\ c$ to $T^4 = acgt\ acac\ acgt\ gtac\ acgt\ acac$.

This example shows a transformation of a text T of length 24, to a text T^4 and which uses a new alphabet. The original alphabet $\Sigma = \{a, c, g, t\}$ contains four symbols, the new alphabet $\Sigma^4 = \{aaaa, aaac, aaag, \ldots, tttt\}$ contains $|\Sigma|^4 = 256$ symbols. These relations can be generalized, it holds for a text T, alphabet Σ and a degree of condensation k:

- A text T of length n can be transformed to a text T^k of length $\frac{n}{k}$.
- If the original alphabet is Σ, then the new alphabet is Σ^k of size $|\Sigma|^k$.

If we use the transformed alphabet, the degree to which a text can be reduced depends on the factor of condensation. In the example above a suitable usage of a condensation factor $k = 4$ for a text over an alphabet of DNA sequences, the transformed text uses 256 symbols, which can be encoded by 8 bits, i.e., using a standard ASCII code. Problems connected with condensation of a text whose length is not a natural multiplication of the condensation factor are reflected in the necessary modification of a pattern matching algorithm. These problems as well as the algorithm are dealt with in section 5.

To demonstrate the reduction of memory requirements we show an example of a $SDAWG$ for the text in Figure 5.

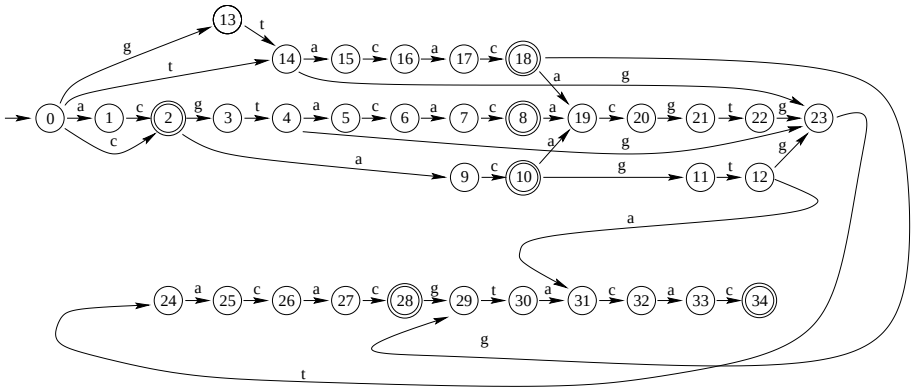

Figure 6: $SDAWG$ for text $T = a\,c\,g\,t\,a\,c\,a\,c\,a\,c\,g\,t\,g\,t\,a\,c\,a\,c\,g\,t\,a\,c\,a\,c$.

Figure 6 shows an example of a $SDAWG$(T) automaton. This automaton contains 35 states and 43 transitions.

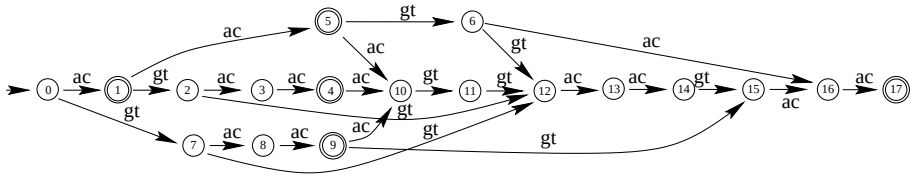

Figure 7: $SDAWG$ for text $T^2 = ac\ gt\ ac\ ac\ ac\ gt\ gt\ ac\ ac\ gt\ ac\ ac$.

Figure 7 shows an $SDAWG$ automaton for condensed text
$T^2 = acgtacacacgtgtacacgtacac$, with condensation factor $k = 2$. This automaton contains 18 states and 24 transitions. Using a condensation factor of $k = 4$ we will get an automaton shown in Figure 8. This automaton, which is constructed for text
$T^4 = acgt\ acac\ acgt\ gtac\ acgt\ acac$, has 7 states and 9 transitions and accepts all suffixes of the text T^4. These suffixes are written in the new alphabet and correspond to the suffixes of the original text that have four times the length of the suffixes in the new alphabet.

We can construct all automata mentioned in section 2 for condensed texts. The method is very simple. It is enough to state that an elementary unit of an alphabet is not one symbol, but a k-tuple of symbols. Then a standard construction algorithm is used. The constructed automaton accepts factors, or suffixes of the transformed text.

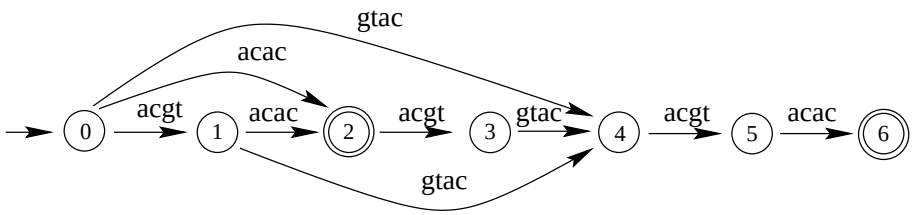

Figure 8: $SDAWG$ for text $T^4 = acgt\ acac\ acgt\ gtac\ acgt\ acac$.

5 String matching algorithm

The string matching algorithm is similar to the methods used in [KaUk96]. Thus we will use the same terminology to denote an occurence of a pattern in a text, i.e., the position in the original text where a k-tuple starts that forms a symbol of the transformed alphabet. That is why we will use the same terminology for referencing occurences of patterns in the original text where a k-tuple starts that is used for condensation - suffix points.

If a pattern starts at a position in the original text that is the same as a position of some suffix point, the occurence of its condensed representation in a condensed text is found directly. If it starts at a position that does not match any suffix point, let us call a pattern prefix that corresponds to a substring of a text and that starts at the position of the pattern and ends at the nearest suffix point as the head. The following part of the pattern that ends at the last suffix point of the occurence of the pattern we will call the body and the last part of the pattern as the tail. An example for a pattern $P = acacgtgtacac$ is shown in Figure 9.

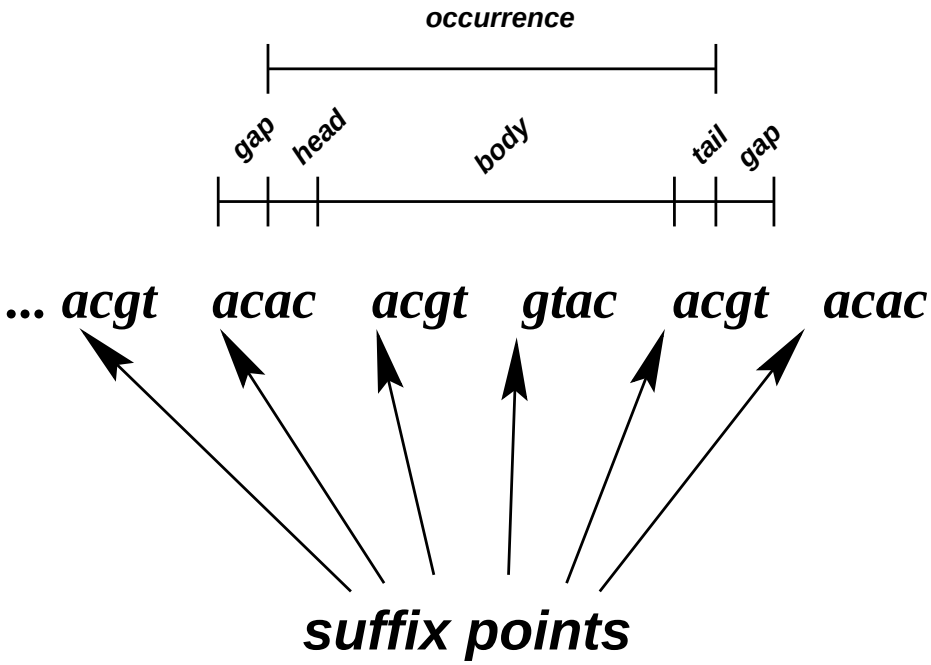

Figure 9: An occurence of a pattern $acacgtgtacac$ in a text.

The figure shows an occurence of a pattern P with head ac, body $acgtgtac$ and tail ac. The length of the body is a multiple of the condensation coefficient and starts at a

position of some suffix point. The condensation directly maps the body to a substring in a condensed text. The length of the head and the tail of the pattern is less than the value of the condensation factor. Thus it is necessary to match all posible extensions of the head (or tail) up to the magnitude of the condensation coefficient so that a condensation of this extensions is possible. The head must end at some suffix point, that is why it is extended with a prefix, similarly the tail is extended with some suffix.

Pattern matching that uses a condensed automaton is done on many levels. On each level we try to match a pattern shifted by one symbol with respect to the suffix points. On level zero we try to locate occurences from some suffix point, on level one we try to find possible occurences that start at some successor of a suffix point etc., on the last level we inspect positions that are shifted by k-1 symbols with respect to a preceding suffix point. On each level a pattern can be divided into a head, a body and a tail. The length of the head on level zero is zero, on level 1 its length is k-1, ..., on level k-1 its length is equal to one. To execute each level we can use either of these two methods:

- Metod 1 : Find a state that matches the body of the pattern. For each transition that starts at this state find out whether the non-condensed form of its labeling has not a prefix identical with the tail of the pattern. For each such a pattern find positions in the text. For these positions it holds that they connect the body of the pattern to its tail, so we have to check only whether it is preceded by a symbol whose suffix (after a decomposition) is the head of the pattern.

- Metod 2: For a non-zero length of the head generate all symbols of the new alphabet whose head after the decondensation is the same as its suffix. We will concatenate each such a symbol with the body of the pattern. Next find such a state in the automaton that corresponds to the concatenation. Determine whether every transition that starts from this state has not a prefix that is the same as the tail of the pattern.

We can use any of these methods for any level of matching, the matching can be modified according to the expected speed of the finishing phase. For a long body with a small (expected) number of occurences Method 1 is more suitable, for a short body and a relatively long head it is better to use Method 2.

While Method 2 is more universal, Method 1 uses an association between a state of the automaton and the position(s) in the text. Thus it is meaningful only for automata that keep this association, a typical example is a *suffix tree* and an *identifier trie*.

To make the algorithm working efficiently it is necessary to introduce a suitable mapping of k symbols (k-tuples) of the original alphabet to the symbols of the new alphabet. This mapping, condensation, together with an inverse mapping, decondensation must be chosen with respect to the number of symbols of the original alphabet. For example each symbol of the DNA alphabet can be encoded using two bits, then each *k-tuples* of a DNA sequence can be encoded by a sequence of bits where each pair is a code of some (DNA) symbol.

The matching algorithm can be modified so that it takes into account specific properties of a particular matching task, for example in the case when we want to discover whether a pattern is present in a text we can terminate the search after the first successful match.

Find a pattern *acacgtgtacac* (symbol ? denotes any symbol of $\{a, c, g, t\}$) Let us suppose we chose Method 2.

Let us consider the text and the pattern from Figure 9. Since the task is pattern matching in a condensed text with the condensation coefficient $k = 4$, the search will be executed on four levels. Level zero is to match occurences that start at suffix points, i.e., we inspect one condensed pattern. Level one inspects four patterns, the original pattern is extended with all possible prefixes of length 1, i.e., all symbols of the original alphabet. Four extensions will be condenseded and will be matched using automaton. It is necessary after matching each pattern to check if any transition that starts at the current state matches the symbol that corresponds to the tail of the pattern complemented with any suffix. Other levels are executed similarly. The different approach to matching heads and tails is due to the number of transitions from the current state. While transitions from the initial state are matched during matching extensions of a head, these transitions exist for each symbol of the input alphabet that is used in the text. For DNA sequences and the degree of condensation $k = 4$ the condensed alphabet contains 256 symbols. To match a tail we compare transitions starting at some non-initial state, there are usually less than three of them. This is an average number, an exact number is proportional to the number of different symbols that follow some occurence of a matched part of the pattern in the text.

The following table shows the number of matched patterns on each matching level. The total number of patterns is the sum of given patterns, i.e., 85 in this case. While the length of the original pattern was m, patterns that are input to the algorithm (in condensed form) have a length $\frac{m}{k}$.

level	length of head	patterns	#of patterns
0	0	*acac gtgt acac*	1
1	1	*?aca cgtg taca c???*	4
2	2	*??ac acgt gtac ac??*	16
3	3	*???a cacg tgta cac?*	64

So far we have assumed that the length of the text is a natural multiple of the condensation factor k. In many cases this condition is not true. These cases are dealt with in a different way. The simplest way is to place this remainder at the beginning of the text and to condens the text without this remainder. Matching all occurences in the text using the algorithm described above, except the occurences that start at some position in the remainder of the text, i.e., that precede the first suffix point. These positions ($k - 1$ positions at most) can be checked later. Method 1 processes these positions more effectively, because the reference to the text where a head of a pattern should be checked is negative, it points in front of the condensed text. It points to the places where there is the rest of the text that was not condensed. It is sufficient to match the head of the patern with this remainder to solve all possibilities. Similarly we can leave the non-condensed remainder at the end of the text.

The principle of condensation is proposed to decrease the amount of memory required to store data structures that are used to simulate the automaton. The search time is proportional to the length of the pattern. If the automaton is too big (for a long text) and the simulator works a lot with a slow memory during parsing, then it is possible that a search that is complex on one hand, but that uses some sophisticated data structures in main memory on the other hand may lead to achieve

a desired result in a shorter time, this is still an open question.

6 Experimental results

The main task solved in this report is the proposal and design of a new type of an automaton over condensed text. The implementation of the resulting automata created using a condensed text with different degree of condensation is more efficient than the original automata. The size of the implementations is shown in Table 1.

Degree of condensation	Size	$SDAWG$	$IDAWG$	$\frac{SDAWG}{origSIZE}$	$\frac{IDAWG}{orig.SIZE}$
Original	230195	946973	898728	4.11	3.90
2	115097	484681	411332	2.10	1.79
3	76731	307393	241535	1.34	1.05
4	57548	250639	191168	1.09	0.83

Table 1: Condensed automata

We can see that using the degree of condensation equal to 4 (i.e., four-tuples of symbols are used to label transitions) leads to an implementation whose size is comparable with the size of the original text.

7 Future work and open questions

Open questions:

- The implementation of condensed automata decreases memory requirements when compared to other types of search automata. The change of search algorithms and the decrease of processing speed thus created could counteract this improvement.
- The aim of this work is to find an effective implementation of a search automaton. Space requirements to store the resulting data structures are not so important, but a very high processing speed of these automata is. Large data structures and a bad implementation can result in excessive memory accesses, which decreases the qualities of the implementation. Thus the most important question is to choose the correct automaton and to implement it correctly so as not to decrease the resulting processing speed.

References

[Bal98] Balík, M. Diploma Thesis, CTU Prague 1998.

[Cro94] Crochemore, M. and Rytter, W. Text algorithms, Oxford University Press, New York 1994.

[GoBa91] Gonnet G.H., Baeza-Yates R., Handbook of Algorithms and Data Structures - In Pascal and C. Addison - Wesley, Wokingham, UK, 1991.

[KaUk96] Juha Kärkkäinen and Esko Ukkonen: Sparse Suffix Trees. Proc. Second Annual International Computing and Combinatorics Conference (COCOON '96), Springer 1996.

[AndNil95] A. Anderson and S. Nilson, Efficient implementation of suffix trees, Software-Practice and Expirience, 25(1995), pp129-141

[GoBaSn91] G.H.Gonnet, R.A. Baeza-Yates, and T. Snider, Lexikographical indices for text: Inverted files vs. PAT trees, Technical report OED-91-01, Centre for the new OED, University of Waterloo, 1991.

[Irv95] R.W.Irving, Suffix binary search trees, Technical report TR-1995-7, Computing science Department, University of Glasgow, Apr.95

[Kark95] Kärkkäinen, J. Suffix cactus: A cross between suffix tree and suffix array, in Proc. 6th Symposium on combinatorial Pattern Matching, CPM95, 1995, pp191-204. U. Manber and G. Myers Suffix arrays: a new method for on-line string searches Proceedings of the 1st ACM-SIAM Annual Symposium on Discrete Algorithms, pp. 319-327, 1990.