UNIVERSITATEA DIN BACĂU
STUDII ŞI CERCETĂRI ŞTIINȚIFICE
Seria: MATEMATICĂ
Nr. (2005) pag. 113 - 121

Adaptive DFA based on array of sets

Dan Popa
University of Bacău
popavdan@yahoo.com

Abstract: In this paper the author explains how the adaptive DFA based on array of sets was discovered, despite the fact that DFA and the theory of lexical analysis are considered closed domains for decades. The paper is introducing a data structure which is able to store together different automata. They are remaining able to work together but everyone uses its own way (i.e. path between states). Applications comes from the field of compiler construction. But other tools, subject of the future papers, may also be written using this technique. It is able to reduce the time needed to release a new compiler because it allows one part of it (which makes lexical analysis) to re-build itself every time when a compiler author needs.

Note: The algorithms used by the system during the learning phase and the data processing (lexical analysis) are not included.

Key words and phrases: adaptive, automata, DFA
MSC (2000) Mathematic Subject Classification 68N20, 68P05, 68Q70

The paper is a result of some research done during the process of preparation of a Ph.D. Thesis conducted by Prof. Hab. D. Todoroi (from ASEM, Chişinău, Rep. Moldova). Some results, including some practical examples were presented in a previous paper (Popa Dan, *Adaptable Tokenizer for Programming Languages*, Simpozionul Internațional al Tinerilor Cercetători, Chişinău 2004, p 55-57.). But two items were missing from that paper ; a complete description of the algorithms and *a description of the used data structure.*

The last one is presented in the following paragraphs.

Goal

The goal of the actual research was to create an adaptive system able to build DFA (able of accepting different kind of tokens). As a part of this project a data structure able to store such DFA making them usable together as a "tokenizer" was necessary. The property of the lexical analysis involved in our study was *the capacity of the program to classify tokens* in their corresponding classes. It was a challenging task because the classes of tokens accepted by an adaptive "tokenizer" are *not predefined* by its construction and had to be acquired on the run by the system, (usually on the base of some previous examples seen during the learning process).

The first example:

A decimal number is usually written as a sequence of digits, placed one after another, without spaces or other symbols between them. Let consider some numbers surrounded by two spaces, noted by "_" (underscore).

Example: 125 noted as _125_

Remark: A single-digit number is not a representative example. It did not reflects the idea of having digits one after an other. In our notation it looks like _1_. This is also visible when examining the grammar of the expressions:

number -> *digit*

number -> *digit number*

Because a single digit number can be generated by a rule like *number -> digit* it's obvious that a single digit number can not provide enough information.

The first set of conclusions

#1: Adaptive automata have to be trained using *relevant examples (like the. examples which are complicated enough to contains necessary information).* Good examples for every kind of tokens in which we are interested should be provided in order to get a well trained system.
#2: Practically, sometimes we can be forced to reunite to classes of tokens to get what we want. For examples we may be forced to explicitely state that numbers means 1-digit numers U more than one digits numbers. But grammars used the same idea for years.
#3: It's better to process the stream of characters with a function, in order to split it in pairs. Such pairs of neighbour-characters will gave us information and will be processed in order to get a sort of *signature of a token*.

Let be $v = (v_1, ...v_n) \quad i = 1..n$
We define the string of pairs $P(v) = (p_1,...p_n)$
where , for every i =1..n , $pi = (v_i, v_{i+1})$

Example: The number _125_ becomes (_,1)(1,2)(2,5)(5,_). External parentheses was removed. Two pairs of symbols reflecting the same rule ("digit and digit') can easily be seen: (1,2)(2,5). Both are matching a common pattern (c,c) where c means "a digit ". That is why we are interested in classes and factorization.

Classes

Usually, the automata theory are using not only singular symbols but classes. For example digits like 0,1,..9 had formed a class (we will denote it by "c"). Letters like 'A'...'Z' and 'a'...'z' are included in an other class, denoted with "l" by us. All kind of spaces, tab and blanc will be included in an other class, denoted by "_". All the symbols remaining are included in the last class, called "s".

Let be f a function which associate a character with its class. (*f('a')=l* , *f('1')=c* , *f(' ')=_* , *f('%')=s* etc.).

If $v = (v_1, ...v_n)$ $i = 1..n$

we can extend f to vectors of characters.

$f(v) = f(v_1),..f(v_n)$

Applying P, the vector of pairs is produced:

$P(f(v)) = p_1,..p_n$

where $p_i = (f(v_i), f(v_{i+1}))$ for each $i = 1..n$

So: $P(f(v)) = (f(v_1), f(v_2)), (f(v_2), f(v_3)), ..., (f(v_i), f(v_{i+1})), ..., (f(v_{n-1}), f(v_n))$

Remark: The extended version of f can also be applied to pairs. let ba a generic pair $p=(a,b)$. We have $f(p)=(f(a), f(b))$. Considering the indexed pairs $p_i = (v_i, v_{i+1})$ and applying f we have:

$f(p_i)=(f(v_i), f(v_{i+1}))$.

It's easy to notice that $p(f(v)) = f(p(v))$.

So, in practice, we can replace characters by their classes and group them in pairs or we can produce the string of pairs and replace the elements of that pairs by the name of the corresponding classes.

Example:

1)_125_ becomes _ccc_ then (_,c)(c,c)(c,c)(c,_)

2)_125_ becomes (_,1)(1,2)(2,5)(5,_) then (_,c)(c,c)(c,c)(c,_)

The data structure

We had studied *sequences of pairs* (like the previous one) because we have remarked them as *a sort of signatures of the tokens*. In fact, this kind of signatures can be used to build automata. We are specially interested in the storage of the signatures in a structure able to make all this stored automata running together.

Let denote by A be the alphabet of a computer language. Because the set A is finite, the cardinal number $q = | \{ (d,e) \mid \text{where } d,e \text{ belongs to } f(A) \} |$ is also finite.

Remember: Only a few number of classes was found for the common programming languages !

In this paper only 4 classes (l , c , s , _) will be used. So, we will have at least 4^2 = 16 ordered pairs of classes.

The data structure introduce by this paper is an array of q x q elements. The elements are subsets of integers. Every integer from such a subset corresponds to a distinct automaton stored in the structure.

Basically, as we will show, if n is the number of a specific automaton, all the numbers n from the sets of the structure was entered during the process of its creation.

Remark: The maximum cardinal of the sets is limited in every common programming language (for example, it is limited at 255 or 256 elements by different implementations of Oberon). This limitation did not affect the solutions. Practical "tokenizers" (i.e. lexical analyzers) have never ever have to recognize 255 different kinds of tokens. Usually, common programming languages are using a small *set of kinds* of tokens. So, despite this limitation, Adaptive "tokenizers" based on Adaptive DFA can be implemented in every usual programming language.

The place of the stored automata

It's time to show haw can a sequence of pairs become an automaton stored in our array of sets.

Let's suppose we have i-1 automata already stored and we are in process of building and storing the new one (automaton number i).

Remark: Only few lines and columns of the array will be altered during this process. The changes are made in some places strictly determined by the set of pairs, because every pair will indicate a cell in the structure. (How do they do this ? The elements of the pair are interpreted as the label of the line and the label of the column, respectively. The element which belongs to the intersection between that line and that column will be modified.)

Example: The *string* of pairs:

(_,c)(c,c)(c,c)(c,_) is producing a *set* of pairs, by eliminating the duplicates:

{ (_,c), (c,c), (c,_) }

Consequently, the changes will be made in the small area determined by the columns

{ (_,c), (c,c), (c,_) } and by the lines { (_,c), (c,c), (c,_) }. In fact the area is, sometimes, a bit smaller as you will see.

Adding information

Procedure: For every pair of classes from the string of pairs (or it's set of pairs), the element of the array which is indicated by the pair - which is a set ! - will receive i as a new element included by the set.

Let (p,q) be a pair: Only M[p,q] is affected for this pair, in this way:

M[p,q] : = M [p,q] U {i}.

In the previous example, the effect is:

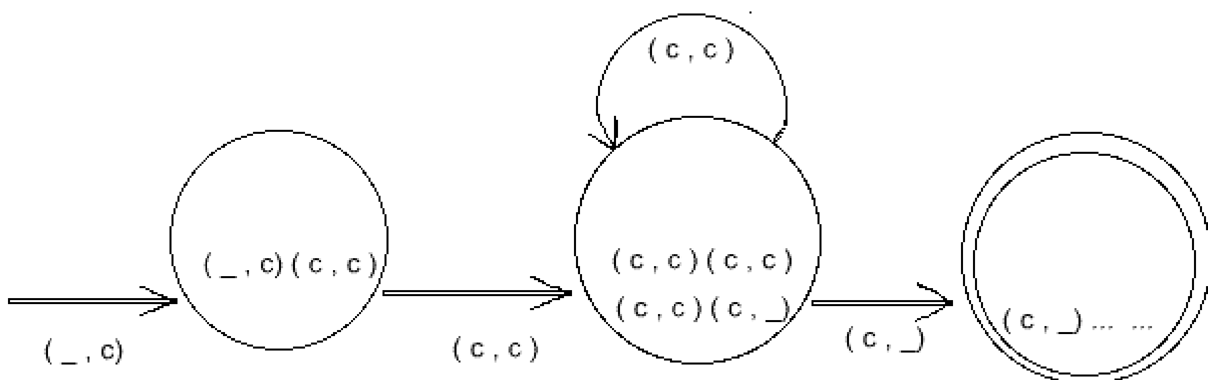
M[(_,c), (c,c)] : = M [(_,c), (c,c)] U {i} because of presence of *(_,c), (c,c)* in the input string.

M[(c,c), (c,c)] : = M [(c,c), (c,c)] U {i} because of presence of *(c,c), (c,c)* in the input string.

M[(c,c), (c,_)] : = M [(c,c), (c,_)] U {i} because of presence of *(c,c), (c,_)* in the input string.

Remark: Human intuition may notices that all three positions affected are corresponding with three nodes from a classic DFA. This is not far away from truth.

- The first place, *M[(_,c), (c,c)]* corresponds to the first node of the DFA , the place where a space followed by a digit is accepted.
- The second place, *M[(c,c), (c,c)]* corresponds to the second node, where a digit followed by an other digit is accepted.
- The third state corresponds to the end of the number, where a digit followed by a space is accepted.



Advantage

The DFA (see a trained part of it conveniently figured in the picture) can accept a no matter how many digits number. Even if the analyzer was trained using only a 2 digits number, it can accept other longer numbers. This property, which we calls "generality" was found for every kind of usual tokens (including the subsets of the identifiers set), but due to space limitations it is not included herein.

Disadvantage

The figured automata do not recognize a 1-digit number yet. A 1-digit number (which means an other kind of numbers, an also has a special rule in the grammar) wold have to be given as a supplementary example, during the training of the system.

An other disadvantage: The adaptive DFA can theoretically recognize sets of tokens which can be bigger than what a nonspecialist intended to be. (If somebody instructs the system to recognize, let's see, four letter words, the system will recognize any sequences formed by any two or more letters.) It's acceptable because the computer can not actually guess what's in the human mind. Also, *from the point of view of the programming language user, it is good to have all kind of identifiers accepted*, even if the system was trained only with four letter words.

Complexity

Being nothing more than usual automata stored together in a set and able to work together, in the same time, Adaptive automata do not slow down the system. The gain consists in adaptability, not in speed. The system will have a speed of the same order as a classic one.

Conclusion

Despite the opinion of the scientists which declared lexical analysis a closed domain, we have been able to introduce *Adaptive DFA based on array of sets.*

Such Adaptive DFA are able to learn the forms of the tokens from the text and classify them, even if the system have seen only one or two specimens during the training. This leads to an other way

of specifying tokens for a programming language, based by well chosen examples, which is far simple as regular expressions or DFA.

References

Aaby Anthony A, Popa Dan; *Construcția compilatoarelor cu Flex şi Bison*, EduSoft, Bacău, 2006

Popa Dan, *Adaptable Tokenizer for Programming Languages*, Simpozionul Internațional al Tinerilor Cercetători, Chişinău 2004, p 55-57.

Serbănați, Luca Dan, Limbaje de programare şi compilatoare, Editura Academiei, Bucureşti 1987