Finite Automata Implementations Considering CPU Cache

J. Holub

The finite automata are mathematical models for finite state systems. More general finite automaton is the nondeterministic finite automaton (NFA) that cannot be directly used. It is usually transformed to the deterministic finite automaton (DFA) that then runs in time $\mathcal{O}(n)$, where n is the size of the input text. We present two main approaches to practical implementation of DFA considering CPU cache. The first approach (represented by Table Driven and Hard Coded implementations) is suitable forautomata being run very frequently, typically having cycles. The other approach is suitable for a collection of automata from which various automata are retrieved and then run. This second kind of automata are expected to be cycle-free.

Keywords: deterministic finite automaton, CPU cache, implementation.

1 Introduction

The original formal study of finite state systems (neural nets) is from 1943 by McCulloch and Pitts [14]. In 1956 Kleene [13] modeled the neural nets of McCulloch and Pitts by finite automata. In that time similar models were presented by Huffman [12], Moore [17], and Mealy [15]. In 1959, Rabin and Scott introduced nondeterministic finite automata (NFA) in [21].

The finite automata theory is a well developed theory. It deals with regular languages, regular expressions, regular grammars, NFAs, deterministic finite automata (DFAs), and various transformations among the previously listed formalisms. The final product of the theory towards practical implementation is a DFA.

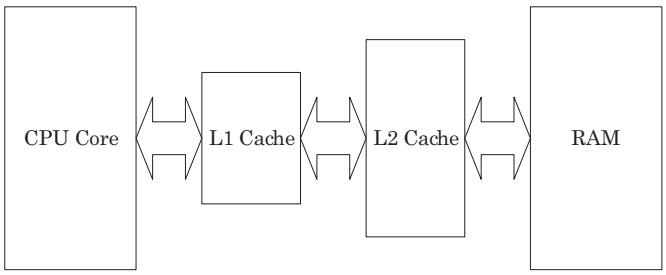

Fig. 1: Memory Cache Hierarchy

DFA then runs theoretically in time $\mathcal{O}(n)$, where n is the size of the input text. However, in practice we have to consider CPU cache that rapidly influences the speed. CPU has two level caches displayed in Fig. 1. The level 1 (L1) cache is located on chip. It takes about 2–3 CPU cycles to access data in L1 cache. The level 2 (L2) cache may be on chip or may be external. It has about 10 cycles access time. The main memory access takes 150–200 cycles and hard disc drive access takes even 10^6 times more time. Therefore it is obvious that CPU cache significantly influences DFA run. We cannot control the CPU cache use directly, but knowing the CPU cache strategies we can implement the DFA run in a way so that CPU cache would be most likely efficiently used.

We distinguish two kinds of use of DFA. For each of them we describe the most suitable implementation. In Section 2 we define nondeterministic finite automaton and discuss its usage. Section 3 then describes general techniques for DFA implementation. It is mostly suitable for DFA that is run most of the time. Since DFA has a finite set of states, this kind of DFA has to have cycles. Recent results in the implementation using CPU cache are discussed in Section 4. On the other hand we have a collection of DFAs each representing some document (e.g., in the form of complete index in case of factor or suffix automata). Such DFA is used only when properties of the corresponding document are examined. Such automaton usually does not have cycles. There are different requirements for implementation of such DFA. Suitable implementations are described in Section 5.

2 Nondeterministic finite automaton

Nondeterministic finite automaton (NFA) is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a set of input symbols, δ is a mapping $Q \times (\Sigma \cup \{\varepsilon\}) \mapsto \mathcal{P}(Q)$, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is a set of final states. *Deterministic finite automaton* (DFA) is a special case of NFA, where δ is a mapping $Q \times \Sigma \mapsto Q$.

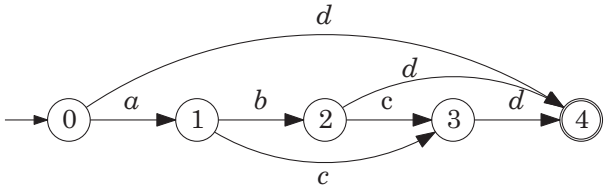

Fig. 2: A deterministic finite automaton

In the previous definition we talk about *completely defined DFA*, where there is for each source state and each input symbol *exactly one* destination state defined. However, there is also *partially defined DFA*, where there is for each source state and each input symbol *at most one* destination state defined. The partially defined DFA can be transformed to completely defined DFA introducing a new state (so called sink state) which has a self loop for each symbol of Σ and into which all non-defined transitions of all states lead.

There are also NFAs with more than one initial state. Such NFAs can be transformed to NFAs with one initial state introducing a new initial state from which ε-transitions lead to all former initial states.

NFA accepts a given input string $w \in \Sigma^*$ if there exists a path (a sequence of transitions) from the initial state to a final state spelling w. The problem occurs when for a pair (q, a), $q \in Q$, $a \in \Sigma$ (i.e., state q of NFA is active and a in the current input symbol) there are more possibilities how to continue:

1. There are more than one transitions labeled by a outgoing from state q. That is $|\delta(q,a)| > 1$.
2. There is an ε-transition in addition to other transitions outgoing from the same state.

In such a case NFA cannot decide, having only the knowledge of the current state and current input symbol, which transition to take. Due to this nondeterminism NFA cannot be directly used. There are two options:

1. We can transform NFA to the equivalent DFA using the standard subset construction [21]. However, it may lead to an exponential increase of number of states ($2^{|Q_{\mathrm{NFA}}|}$ states, where $|Q_{\mathrm{NFA}}|$ is the number of states of the original NFA). The resulting DFA then runs in linear time with respect to the size of the input text.
2. We can simulate the run of NFA in a deterministic way. We can use Basic Simulation Method [7, 6] usable for any NFA. For NFA with a regular structure (like in the exact and approximate pattern matching field) we can use Bit Parallelism [16,7,6,10] or Dynamic Programming [16, 8, 6] simulation methods which improve the running time of the Basic Simulation Method in this special case. The simulation runs slower than DFA however the memory requirements are much smaller. Practical experiments were given in [11].

3 Deterministic finite automaton implementation

Further in the text we do not consider simulation techniques. We consider only DFA. DFA runs theoretically in time $\mathcal{O}(n)$, where n is the size of the input text.

There are two main techniques for implementation of DFA:

1. Table Driven (TD): The mapping δ is implemented as a transition matrix of size $|Q| \times |\Sigma|$ (transition table). The current state number is held in a variable q_{curr} and the next state number is retrieved from the transitiontable from line q_{curr} and column a, where a is the current input symbol.
2. Hard Coded (HC) [22]: The transition table δ is represented as a programming language code. For each state there is a place starting with a state-label. Then there is a sequence of conditional jumps, where based on the current input symbol the corresponding `goto` command to the destination state-label is performed.

3.1 Table Driven

An example of TD implementation is shown in Fig. 3. For partially defined DFA one have to either transform it to a completely defined DFA or handle the case when a undefined transition should be used.

Obviously TD implementation is very efficient for completely defined DFA or DFAs with non-sparse transition table. It can be also very efficiently used in programs, where DFA is constructed from a given input and then it is run. In such a case it can be easily stored into the transition matrix. The code for the DFA run is then independent on the content of the transition matrix. TD implementation is also very convenient for a hardware implementation, where the transition matrix is represented by a memory chip.

3.2 Hard Coded

An example of HC implementation is shown in Fig. 4. The implementation can work with partially defined DFA in this case.

HC implementation may save some space when used for partially defined DFA, where the transition matrix would be sparse. It cannot be used in programs, where DFA is constructed from the input. When DFA is constructed, a hard coded part of the program has to be generated in a programming language, then compiled and executed. This technique would need calls of several programs (compiler, linker, the DFA program itself) and would be very inefficient.

Note that we cannot use the recursive descent [1] approach from *LL*(*k*) top-down parsing, where each state could be represented by a function calling recursively a function representing the following state. In such a case the system stack would overflow since DFA would return from the function calls only at the end of the run. There would be as many nested function calls as the size of the input text. However, Ngassam's implementation [18] uses a function for each state, but the function (with the current input symbol given as a parameter) returns an index of the next state and then the next state function (with the next input symbol given as a parameter) is called.

transition_table:

	a	*b*	*c*	*d*
0	1	-	-	4
1	-	2	3	-
2	-	-	3	4
3	-	-	-	4
4	-	-	-	-

```
int DFA_TD(){
  int state=0,symbol;

  while((symbol= getchar())!= EOF ) {
    state= transition_table[state][symbol];
  }
  returnis_final[state];
}
```

Fig. 3: Table Driven implementation of DFA from Fig. 2

```
int DFA_HC(){
  int symbol;

  state0:if ((symbol= getchar())== EOF) return0;
         switch(symbol){
           case'a':goto state1;
           case'd':goto state4;
           default:return(-1);
         };
  state1:if ((symbol= getchar())== EOF) return0;
  switch(symbol){
    case'b':goto state2;
    case'c':goto state3;
    default:return(-1);
  };
  state2:if ((symbol= getchar())== EOF) return0;
  switch(symbol){
    case'c':goto state3;
    case'd':goto state4;
    default:return(-1);
  };
  state3:if ((symbol= getchar())== EOF) return0;
  switch(symbol){
    case'd':goto state4;
    default:return(-1);
  };
  state4:if ((symbol= getchar())== EOF) return1;
  return(-1);
}
```

Fig. 4: Hard Coded implementation of DFA from Figure 2

4 DFA with cycles

TD and HC implementations (and their combination called Mixed-Mode – MM) were heavily examined by Ngassam [20, 18]. His implementations use a data structure that most likely will be stored in CPU cache. For each of TD and HC implementationshe developed three strategies to use CPU cache efficiently: Dynamic State Allocation (DSA), State pre--ordering (SpO), and Allocated Virtual Caching (AVC).

DSA strategy has been suggested in [19] and was proved to outperform TD when a large-scale DFA is used to recognize very long strings that tend to repeatedly visit the same set of states. SpO relies on a degree of prior knowledge about the orderin which states are likely to be visited at run-time. It was shown that the associated algorithm outperforms its TD counterpart no matter the kind of string being processed. AVC strategy reorders the transition table at run time and also leads to better performance when processing strings that visit a limited number of states.

Ngassam's approach can be efficiently exploited in DFA, where some states are frequently visited (like in DFA with cycles). In both TD and HC Ngassam's implementations the transition table is expected to have the same number of items in each row (i.e., each state having the same number of outgoing transitions). Ngassam's implementation uses a fixed--size structure for each row of the transition table. Therefore for sparse transition matrix the method is not so memory efficient.

5 Acyclic DFA

Another approach is used for acyclic DFA. In these automata each state is visited just once during the DFA run. Suffix automaton and factor automaton (automaton recognizing all suffixes and factors of the given string, respectively) [3, 4] are of such kind. Given a pattern they verify if the pattern is a suffix or a factor of the original string in time linear with the length of pattern regardless the size of the original string.

An efficient implementation of the suffix automaton (also called DAWG – Direct Acyclic Word Graph) was created by Balík [2]. An implementation of the compact version of the suffix automaton called compact suffix automaton (also called Compact DAWG) was presented by Crochemore and Holub in [9].

Both these implementations are very efficient in terms of memory used (about 1.1–5 bytes per input string symbol). The factor and suffix automata are usually built over whole texts typically several megabytes long. Instead of storing the transition tableas a matrix like in TD implementation, whole automaton is used in a bit stream. The bit stream contains a sequence of states each containing a list of all outgoing transitions (i.e., sparse matrix representation).

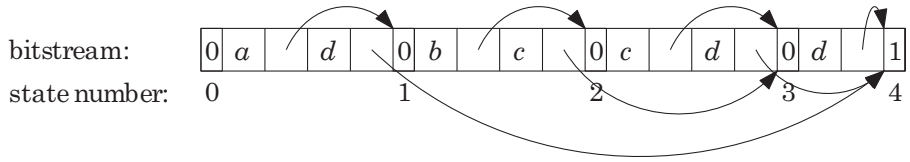

Fig. 5: A sketch of bitstream implementation of DFA from Fig. 2

The key feature of both implementations is a topological ordering of states. It ensures that we never get back in the bit stream when traversing the automaton. This minimizes main memory (or hard disc drive) accesses.

Balík's implementation is focused on the smallest memory used. It uses some data compression techniques. It also exploits the fact that both factor and suffix automata are homogeneous automata [5], where each state has all incoming transitions labeled by the same symbol. Therefore the label of incoming transition is stored in the destination state. The outgoing transition then only points to the destination state, where the corresponding transition label is stored.

On the other hand Holub's implementation considers also the speed of traversing. Each state contains all outgoing transitions together with their transition labels like in Fig. 5. (However, the DFA represented in Fig. 5 is neither suffix nor factor automaton.) It is not so memory efficient like Balík's implementation but it reduces main memory (or hard disc drive) accesses. It exploits the locality of data – principle used by CPU cache. When a state isreached during the DFA run, whole segment around the state is loaded into CPU cache (from main memory or hard disc drive). The decision which transition to take is done based only on the information in the segment (in the CPU cache) and no other accessesto other segments (i.e., possible memory/HDD accesses) are needed. While in Balík's implementation one needs to access all the destination states to retrieve the transition labels of the corresponding transitions. Holub's implementation uses at most as many main memory/HDD accesses as many states are traversed.

6 Conclusion

The paper presents two approaches to DFA implementation considering CPU cache. The first approach is suitable for DFA with cycles where we expect some states are visited frequently. HC and TD implementations for DFA with non-sparse transition table were discussed.

On the other hand the other approach is suitable for acyclic DFA with a sparse transition table. This approach saves memory used but it runs slower than the previous one – instead of direct transition table access (coordinates given by the current state and the current input symbol) a linked list of outgoing transition of a given state is linearly traversed. However, reducing the memory used for the transition table increases the probability that the next state is already in the CPU cache which also increases the speed of DFA run.

The first approach is suitable for the DFAs that are running all the time like for example an anti-virus filter on a communication line. On the other hand the second approach is suitable for a collection of DFAs from which one is selected and then it is run. That is for example a case of suffix or factor automata build over a collection of documents stored in hard disk. The task is then for a given pattern find all documents containing the pattern.

Acknowledgment

This research has been partially supported by the Ministry of Education, Youth and Sports under research program MSM 6840770014 and the Czech Science Foundation as project No. 201/06/1039.

References

[1] Aho, A. V., Sethi, R., Ullman, J. D.: *Compilers – Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] Balík, M.: DAWG versus Suffix Array. In: J.-M. Champarnaud, D. Maurel (eds.): *Implementation and Application of Automata*, number 2608 in Lecture Notes in Computer Science, p. 233–238. Springer-Verlag, Heidelberg, 2003.

[3] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M. T., Seiferas, J.: The Smallest Automaton Recognizing the Subwords of a Text. *Theor. Comput. Sci.*, Vol. **40** (1985), No. 1, p. 31–55.

[4] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., McConnel, R.: Complete Inverted Files for Efficient Text Retrieval and Analysis. *J. Assoc. Comput. Mach.*, Vol. **34** (1987), No. 3, p. 578–595.

[5] Champarnaud, J.-M.: Subset Construction Complexity for Homogeneous Automata, Position Automata and ZPC-Structures. *Theor. Comput. Sci.*, Vol. **267** (2001), No. 1–2, p. 17–34.

[6] Holub, J.: *Simulation of Nondeterministic Finite Automata in Pattern Matching*. Ph.D. Thesis, Czech Technical University in Prague, Czech Republic, 2000.

[7] Holub, J.: Bit Parallelism – NFA Simulation. In: B. W. Watson, D. Wood (eds.): *Implementation and Application of Automata*, number 2494 in Lecture Notes in Computer Science, p. 149–160. Springer-Verlag, Heidelberg, 2002.

[8] Holub, J.: Dynamic Programming—NFA Simulation. In: J.-M. Champarnaud, D. Maurel (eds.): *Implementation and Application of Automata*, number 2608 in Lecture Notes in Computer Science, p. 295–300. Springer-Verlag, Heidelberg, 2003.

[9] Holub, J., Crochemore, M.: On the Implementation of Compact DAWG's. In: J.-M. Champarnaud, D. Maurel (eds.): *Implementation and Application of Automata*, number 2608 in Lecture Notes in Computer Science, p. 289–294. Springer-Verlag, Heidelberg, 2003.

[10] Holub, J., Iliopoulos, C. S., Melichar, B., Mouchard, L.: Distributed String Matching Using Finite Automata. In: R. Raman, J. Simpson (eds.): *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, p. 114–128, Perth, WA, Australia, 1999.

[11] Holub, J., Špiller, P.: Practical Experiments with NFA Simulation. In: L. Cleophas, B. W. Watson (eds.): *Proceedings of the Eindhoven FASTAR Days 2004*, TU Eindhoven, The Netherlands, 2004, p. 73–95.

[12] Huffman, D. A.: The Synthesis of Sequential Switching Circuits. *J. Franklin Institute*, Vol. **257** (1954), p. 161–190, 275–303.

[13] Kleene, S. C.: Representation of Events in Nerve Nets and Finite Automata. *Automata Studies*, (1956), p. 3–42.

[14] McCulloch, W. S., Pitts, W.: A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bull. Math. Biophysics*, Vol. **5** (1943), p. 115–133.

[15] Mealy, G. H.: A Method for Synthetizing Sequential Circuits. *Bell System Technical J.*, Vol. **34** (1955), No. 5, p. 1045–1079.

[16] Melichar, B.: Approximate String Matching by Finite Automata. In: V. Hlaváč, R. Šára (eds.): *Computer Analysis of Images and Patterns*, number 970 in Lecture Notes in Computer Science, p. 342–349. Springer-Verlag, Berlin, 1995.

[17] Moore, E. F.: Gedanken Experiments on Sequential Machines. *Automata Studies*, 1956, p. 129–153.

[18] Ngassam, E. K.: *Towards Cache Optimization in Finite Automata Implementations*. Ph.D. Thesis, University of Pretoria, South Africa, 2006.

[19] Ngassam, E. K., Kourie, D. G., Watson, B. W.: Reordering Finite Automatata States for Fast String Recognition. In: J. Holub, M. Šimánek (eds.): *Proceedings of the Prague Stringology Conference '05*, Czech Technical University in Prague, Czech Republic, 2005, p. 69–80.

[20] Ngassam, E. K., Kourie, D. G., Watson, B. W.: On Implementation and Performance of Table-Driven DFA-Based String Processors. In: J. Holub, J. Žďárek (eds.): *Proceedings of the Prague Stringology Conference '06*, Czech Technical University in Prague, Czech Republic, 2006, p. 108–122.

[21] Rabin, M. O., Scott, D.: Finite Automata and Their Decision Problems. *IBM J. Res. Dev.*, Vol. **3** (1959), p. 114–125.

[22] Thompson, K.: Regular Expression Search Algorithm. *Commun. Assoc. Comput. Mach.*, Vol. **11** (1968), p. 419–422.

Ing. Jan Holub, Ph.D.
e-mail: holub@fel.cvut.cz

Department of Computer Science and Engineering

Czech Technical University in Prague
Faculty of Electrical Engineering
Karlovo nám. 13
121 35 Prague 2, Czech Republic