Hardcoding and dynamic implementation of finite automata

Hardcoding and Dynamic Implementation of Finite Automata

Ernest Ketcha Ngassam[a], Bruce W. Watson[b], and Derrick G. Kourie[b]
[a] School of Computing, University of South Africa,
Pretoria 0003, South Africa
[b] Department of Computer Science, University of Pretoria,
Pretoria 0002, South Africa
E-mail: [a]ngassek@unisa.ac.za, [b]{bwatson, dkourie}@cs.up.ac.za

November 15, 2004

Abstract

The theoretical complexity of a string recognizer is linear to the length of the string being tested for acceptance. However, for some kind of strings the processing time largely depends on the number of states visited by the recognizer at run-time. Various experiments are conducted in order to compare the time efficiency of both hardcoded and table-driven algorithms when using such strings patterns. The results of the experiments are cross-compared in order to show the efficiency of the hardcoded algorithm over its table-driven counterpart. This help further the investigations on the problem of the dynamic implementation of finite automata. It is shown that we can rely on the history of the states previously visited in the dynamic framework in order to predict the suitable algorithm for acceptance testing.

1 Introduction

Previous work on Finite Automata (FAs) implementation revealed that the traditional table-driven (TD) algorithm may not be the sole approach for encoding a string recognizer. Another implementation approach using a hardcoded (HC) algorithm suggested by Knuth in [Kmp77] showed a time gain over the TD algorithm up to some threshold. In further experiments conducted in [Kwk04], it was shown that the processing time required to recognize a string largely depends on the structure of the string being recognized in relation to the overall structure of the automaton the recognizer is based upon. The possibility to improve the processing time of string recognizers using a dynamic algorithm called DIFAP[1] that handles both TD and HC algorithms simultaneously was

[1]Dynamic Implementation of Finite Automata for Performance

also suggested in [Kwk04]. However, the work only introduced the DIFAP algorithm without a complete analysis of the algorithm in terms of complexity and implementation. In this paper, we further the idea of DIFAP through analysis of its critical parts. The history of strings already processed by the recognizer is used to suggest the suitable algorithm to be used at run-time. Further improvements of DIFAP are suggested such as the extension of the original threshold of efficiency as well as a mixed-mode implementation of FAs using both HC and TD algorithms.

The structure of the remaining of this paper is as follows. In section 2 below, the implementation of FAs using both TD and HC algorithms is revisited. Section 3 reviews the experiments performed on string recognizers in order to capture the break-even variations of both algorithms. Various string patterns are investigated in this section showing the advantage of using HC over TD above the threshold of efficiency. In section 4, DIFAP is revisited followed by the analysis of its most critical section referred to as the knowledge table (KT). Additional improvements of DIFAP are suggested in Section 5, and we conclude and provide future directions to this work in Section 6.

2 Finite Automata Implementation

This section summarizes a review of automata implementation and the complexity of string recognizers already discussed in [Ket03]. Finite automata can be implemented using hardcode or softcode. The softcoded or TD algorithm requires a driver program made of few instructions to access the transition table during the entire recognition process. Algorithm 1 below depicts a TD algorithm that tests whether the string *str* is part of the language of the automaton represented by its transition matrix referred to as *transition*. The overall complexity of the recognizer is in the order of $O(len)$ where *len* is the length of the string being tested for acceptance. A hardcoded algorithm depicted in Algorithm 2 clearly shows that more instructions are required to represent the overall recognizer. Again, the complexity of the recognizer still remains in the order of $O(len)$ as for the TD algorithm. Both algorithms are different in terms of instructions and external data usage. The HC algorithm requires many instructions, which is not the case for the TD algorithm. The transition matrix is loaded into memory for the TD algorithm whereas only simple instructions are needed for its representation in the HC algorithm. Such observations clearly show that practical experiments are necessary to evaluate up to what extend the processing time of both algorithms differ. Section 3 below depicts various experiments on strings recognizers using both approaches.

Algorithm 1. *Table-driven string recognition*

```
function recognize(str,transition):boolean
    state := 0;
    stringPos := 0;
    while(stringPos < len) ∧(state ≥ 0) do
        state := transition[state][str[stringPos]];
```

```
      stringPos := stringPos+1;
    end while
    if state ≤ 0
      return(false);
    else
      return(true);
    end if
  end function
```

Algorithm 2. *Hardcoded string recognition*

```
 function recognize(str):boolean
  $state_0$ :
   if $str[0] \notin validsymbol_0$ return(false);
   else if len = 1 return(true);
   else goto $nextStates_1$;
   end if
  $state_1$ :
   if $str[1] \notin validsymbol_1$ return(false);
   else if len = 2 return(true);
   else goto $nextStates_2$;
   end if
...
...
...
  $state_{numberOFStates-1}$ :
   if $str[numberOfStates-1] \notin validsymbol_{numberOfStates-1}$ return(false);
   else return(true);
   end if
 end function
```

3 String Recognition Experiments

In this section, we present the experiments carried out on various kind of strings using both HC and TD algorithms. The experiments where conducted on an Intel Pentium IV at 1.8 GHz with 512MB of RAM and 20GB of hard drive. The TD algorithm was implemented using the gnu C++ compiler, and NASM (Netwide Assembler) was used for the HC implementation. The experiments were conducted under the Linux operating system. Randomly generated strings were investigated as well as various kind of strings that may offer some time gain when the HC algorithm is considered as opposed to the TD algorithm. This section summarizes experiments carried out in [Kwk04]. The subsection below depicts experiments based on random strings.

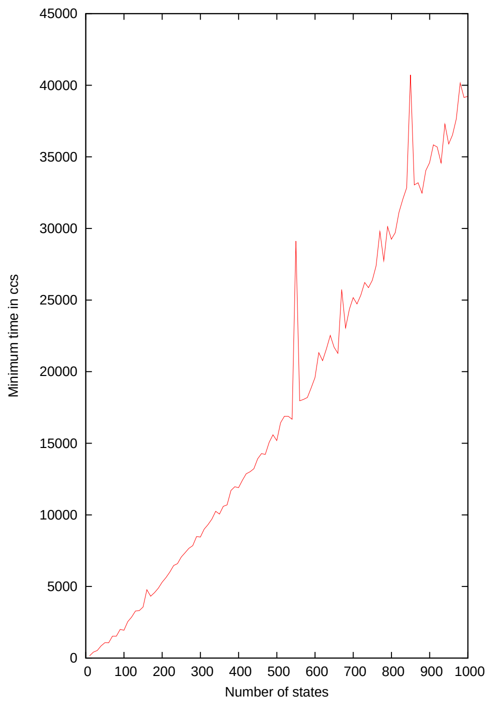

Figure 1: TD performance: 25 symbols

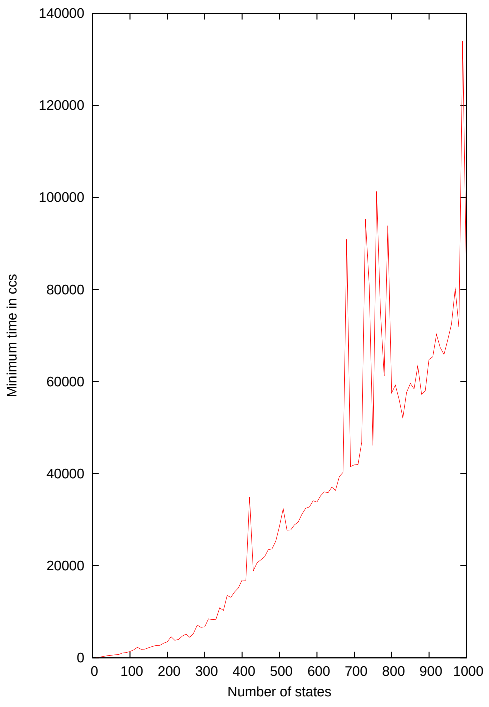

Figure 2: HC performance: 25 symbols

3.1 Experiments based on Random Strings

Experiments were based on alphabet size varying between 10 and 50 symbols with an increment of 5. For each alphabet size under consideration, 100 random automata of sizes ranging from 10 states to 1000 states with an increment of 10 were generated. For each case, a random accepting string of length $n - 1$ was also generated for acceptance testing. Figures 1 and 2 depict the performance for both TD and HC algorithms for automata based on 25 alphabet symbols. It is observed that both graphs show a superlinear growth on the number of states. However, the HC experiment shows a slow growth in the region between 10 states and about 400 states. A plausible explanation to this may lies on the effect of cache on automata of smaller sizes. For such automata, the entire code size can fit into cache and reducing therefore the processing time since the probability of cache misses is very low in that region. Above the 400 states, the HC processing becomes inefficient due to the high probability of cache misses. A comparison between the two algorithms is depicted in Figure 3. It is clearly observed that the HC algorithm outperforms the TD algorithm up to the region between 300 and 400 states. As a result to this, FAs implementers should consider using the HC algorithm when solving computational problems based on automata of size less than about 360 states. However, above that threshold, the TD algorithm

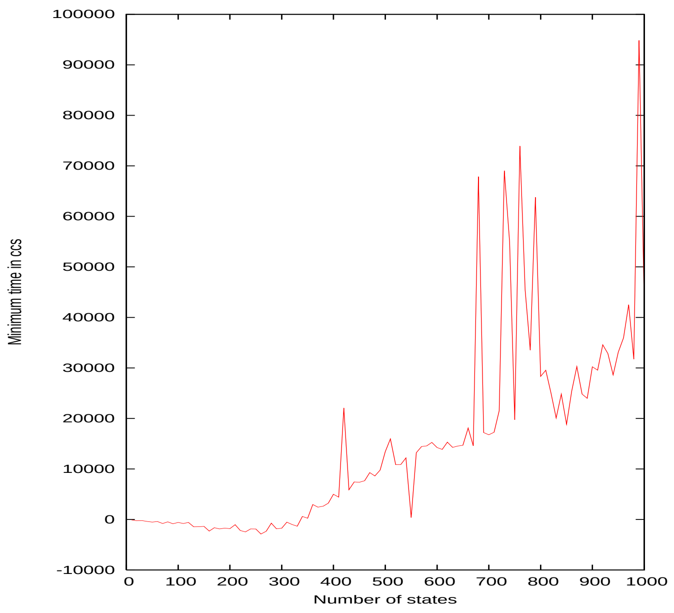

Figure 3: HC-TD performance aphabet size = 25

may be the suitable implementation approach. Since in practice, various strings of same size may require different processing time, it is of importance to conduct experiments in order to cross-compare the time efficiency for both algorithms. The following subsections depict various such experiments.

3.2 The single jump experiments

In this subsection, experiments were performed on strings that keep the FA only on a single state. The recognizer only jumps once on a single states and remains there for the entire recognition process. Thus the title *single jump experiment* suggested by the HC algorithm. The section summarizes experiments already suggested in [Kwk04]. Let consider the automaton modelled in Figure 4 having 5 states with two accepting states 3 and 4. The strings *abab* and *cdef* of size 4 are both part of the language modelled by the automaton. In theory, the total time required to accept or reject each string should be roughly the same. However, we notice that for the string *abab*, once the device reads the first symbol *a*, it jumps to the final state 3 and remains there until the entire string is processed. On the other hand, the recognizer will transverse several different states in order to accept the string *cdef*. This observation indicates that in practice, the time required to accept strings of same size with different patterns may differ considerably from one another. For the experiments, we randomly generated various automata of sizes between 10 and 1000 states using alphabet size varying between 10 and 50. Figure 5 depicts the difference in time between

$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[r, "c"] \arrow[rrr, bend left, "a"] & 1 \arrow[r, "d"] & 2 \arrow[r, "e"] & 3 \arrow[r, "f"] \arrow[loop above, "{a,b}"] & 4
\end{tikzcd}$$

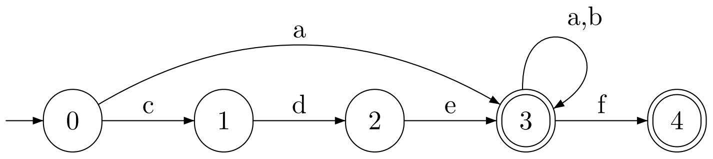

Figure 4: A state diagram: accepts the strings *"abab"* and *"cdef"*

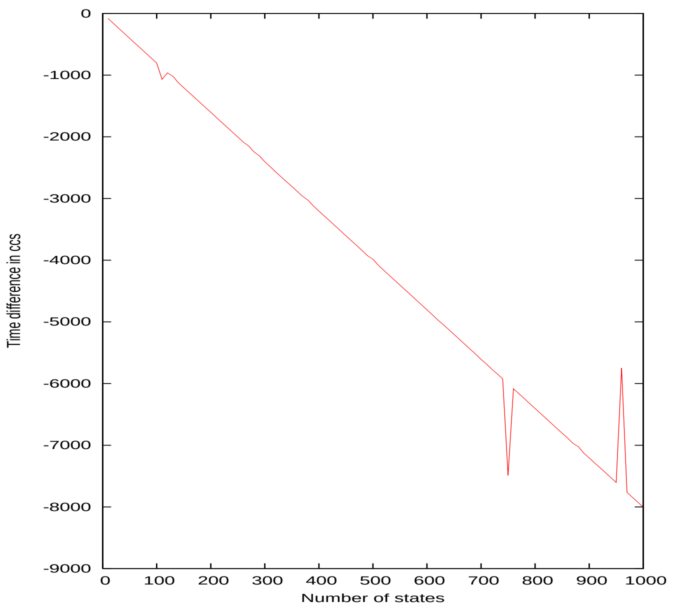

Figure 5: HC-TD performance for single jump

the table-driven and hardcoded implementation. It clearly shows that the HC algorithm is superlinearly faster than its TD counterpart. The average time efficiency per symbol is about 8 ccs. That is, for an automaton with n states, the HC algorithm is $8n$ times faster than the TD algorithm. These results illustrate that for strings following the pattern above described, the processor has sufficient space in its cache to hold the code relating to a single state. Since it always visits the same state over and over, there is a very low probability of cache misses. The experiment is therefore the best case scenario for both algorithms although the HC algorithm appears to be the most efficient. In the next subsection we explore another variant of such strings whereby the automaton visits only the starting state and the final state.

3.3 The far jump experiments

The experiments was suggested by the HC algorithm in the sense that from its initial state, the recognizer jumps to the final state and loops between the two states during the entire recognition process. The state diagram in figure 6 depicts such strings. We consider two strings *ab...ab* and *cd...ef* of length $n-1$ that are both accepting strings. The recognition process of the second string requires that the recognizer visits several states in the automaton whereas the first string only visits two states. Unlike the fact that far jumps are required to

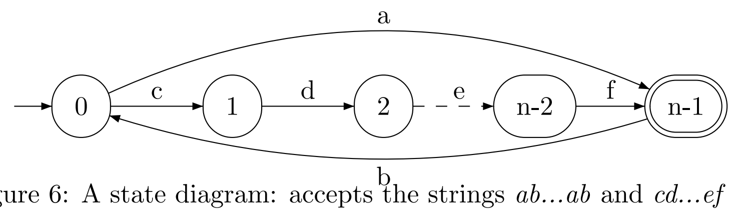

Figure 6: A state diagram: accepts the strings *ab...ab* and *cd...ef*

move from the initial states to the final state for the first string, only limited number of states are visited. Therefore, we expect that such strings are tested at a very efficient time due to the very low probability of cache misses the processor is subject to. As show in figure 7, experiments revealed that the HC algorithm still outperforms its TD counterpart in such context. As a result to this, although in an average case behaviour there is a threshold of efficiency of HC over TD, there is still room for improvements above that threshold when some string patterns are considered. In the next section, we use the original

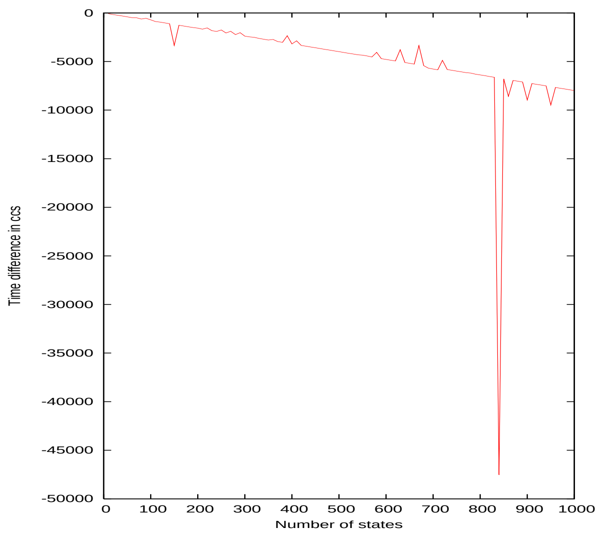

Figure 7: Performance HC-TD for strings that loop on two states

threshold of efficiency to conduct a particular experiment whereby the string remains on a single state after visiting some random states below the threshold.

3.4 A random string experiment followed by single jump

This section summarizes experiments already discussed in [Kwk04]. The random string experiments suggested that the HC outperformed the TD algorithm up to some threshold. In this section, we combine the threshold of efficiency with the experiment conducted in subsection 3.2. Consider the automaton modelled in figure 8. State t depicts some "sink state" in which the FA will remain after some other arbitrary set of states within the automaton have been visited. The state t is also assumed to be an accepting state. Based on the previous experiment, we would expect that the longer the FA remained in state t, the more the hardcoded implementation would enjoy an advantage over the table-driven version.

To verify this observation, and as a sort of sanity check on our results up to this point, hardcoded and table-driven implementations were set up to test strings of length *n-1* in FAs with n states. The experiment was designed so that the behaviour in processing the first 300 states was random - in the same sense as previously described. However, thereafter the FA remains in the same state – i.e. the best case scenario prevails.

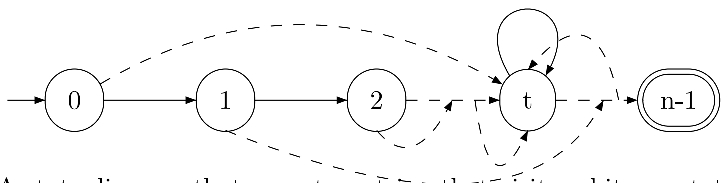

Figure 8: A state diagram that accepts a string that visits arbitrary states and remains on state t for some time

Figure 9 depicts the graphs obtained from the experiment. Unsurprisingly, it shows that hardcoding generally outperforms the table-driven implementation. However, there is also a suggestion in the data that in the longer term, the asymptotic improvement tends towards the 8ccs improvement observed in figure 5.

Of course, many more experiments similar to those described above could be run. An overall and general observation in regard to all these experiments is that they enable us to identify various ways in which the hardcoded implementation of FAs may outperform the traditional table-driven implementation. The next section considers how such information could be used to capitalize on the advantages offered by both approaches.

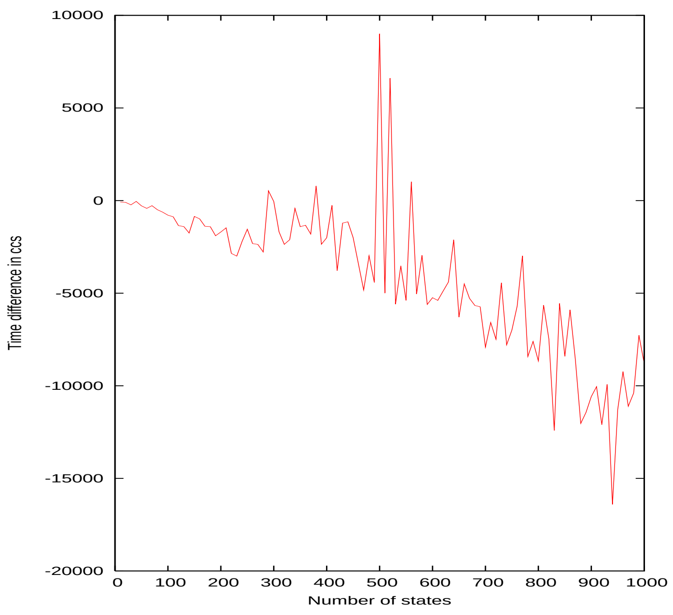

Figure 9: Comparison based on limited access on states

4 The Dynamic Implementation of FAs: DIFAP

The experiments depicted in the previous sections clearly show that the efficiency of a string recognizer is highly dependent on the nature of the string being recognized. This suggests that if likely patterns of strings to be input are known in advance – at least in some probabilistic sense – then it may be possible to put in place a time optimizing mechanism to carry out the string recognition. Consequently, the idea of dynamically adapting the implementation strategy of the FA according to the expected input (or partially inspected) string may be considered. We use the acronym DIFAP to refer to this notion, designating Dynamic Implementation of FAs for Performance enhancement. Figure 10 depicts the overall design of a DIFAP system. When it is first invoked, the implementer provides the specification of the automaton to be used, regardless the type of string to be recognized. DIFAP then analyzes the specification and choose the appropriate way of implementing the automaton depending of the size of the derived automaton. In terms of the currently available data, if the size is less than 360 states, this means that hardcoding is likely to be the optimal approach in representing the automaton irrespective of the kind of string to be tested. On the other hand, if the size of the automaton is above 360 states, as suggested in Section 2, a hardcode implementation might be indicated if long term behaviour is likely to tend towards best case behaviour, a table-driven implementation will be the appropriate choice in the absence of such information. However, in the latter case, DIFAP relies on the kind of string received as input to adapt itself progressively to an implementation approach that is optimal in some sense (e.g. optimal in relation to the history of strings processed to date), resulting in improved average processing speed.

The figure indicates that for bigger automata size, a knowledge table (KT) is first checked. At this stage, we do not prescribe what information should be kept in the KT. We merely observe that the current input string could, in principle, undergo some preliminary scan to identify whether its overall structure conforms to some set of general patterns that favour hardcode over table-driven. If that is not the case, the table-driven version of the automaton specification is generated and is used by the recognizer to check whether the string is part of the language described by the FA or not. Otherwise, the hardcoded version of the FA's specification is generated and used for recognition.

Not indicated in the figure is the possibility of post-processing: after a string has been tested, the string and the test outcome could be used to update information in the KT. As a very simple example, we might decide to concretely implement the KT as a table of the FA's states, in which a count is kept of the number of times a state has been visited. This information could be used to rearrange the order of rows (which represent states) in the transition matrix used by the table-driven approach, in the hope of minimizing data cache misses when this implementation strategy is used. Alternatively, the same information could be used to dictate the blocks of hardcode that should preferentially be loaded into cache, in circumstances in which hardcoding is indicated. However, the foregoing should not be construed as the only way in which the KT can be implemented. We conjecture that there are many creative possibilities within this broad model that merit deeper investigation in the future.

One of the advantage of using such a dynamic algorithm is that the structure of the automaton does not always remains in the system after processing. Each automaton is always regenerated into its executable when the system is invoked. The only structure that permanently remains in the system is the algebraic specification of the automaton. This results therefore in some degree of minimization of memory load for automata of considerable size. However, there is no need to always regenerate automata of size less than 360 states since they will always be implemented in hardcode. That is the reason why in the figure no deletion of the generated hardcode is indicated when the "size less than 360" path is followed.

In an implementation of DIFAP, attention should be given to the following parts of the algorithm to minimize latencies:

- *Time to generate the recognizer:* Unless directly implemented by hand, any FA-related problem always requires a formal specification of the grammar that describes the automaton before its corresponding automaton is encoded. This is a general problem, and one specific to DIFAP. The DIFAP implementation could therefore use generator techniques similar to those used in efficient code generator tools such as YACC[2] which as been proven to be amongst the best tool available to create directly executable

[2]Yet Another Compilers Compiler

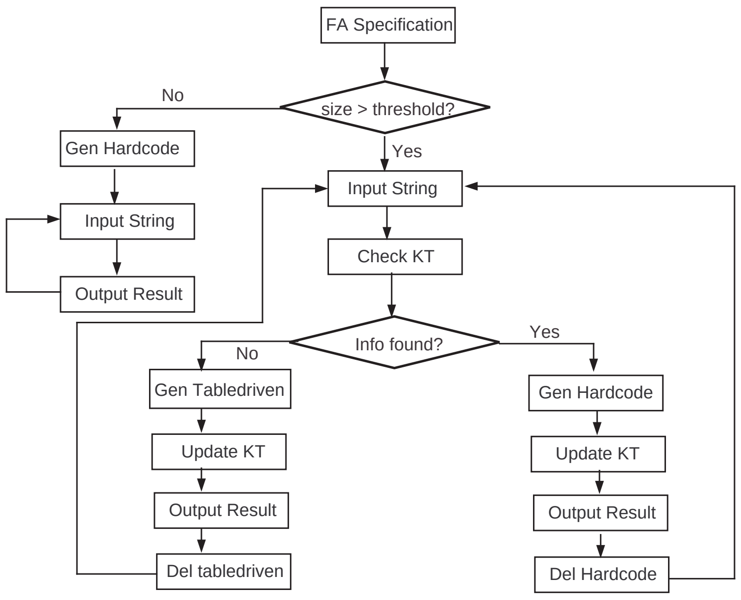

Figure 10: A Preliminary design of DIFAP

parsers. Unlike parsers, DIFAP's code generator will generate directly executable string recognizers.

- *Time taken to check the knowledge table:* One should take care to ensure that the matter of checking the KT does not degenerate into a time-inefficient exercise that negates any benefit from using the optimal string recognition strategy. Efficient algorithms should be devised that take minimal time to access the table and to chose the appropriate path to follow. This part of DIFAP may constitute a bottleneck. Intensive investigations will be made to provide an efficient approach to access the table and retrieve appropriate information.

- *Time required to update the knowledge table:* Although the precise nature and scope of the KT have not been identified here, it is envisaged that it will, itself, be a dynamic structure, changing over time in relation to the history of strings analyzed to date. However, there does not appear to be any reason for adapting the KT prior to processing the input string. Its update is something that can happen at a post-processing stage, and does not appear to be time-critical.

4.1 Structure of KT

The entire DIFAP algorithm relies on the KT in order to provide optimal information to the entire framework for efficient processing of a given recognizer. Various strategies can be used for its representation. Of course the most important notion to bear in mind must be efficiency. Since intensive investigations have not yet been made in order to test any of the strategy we have in mind. We have chosen to solve the problem in a step by step fashion so that each idea is eliminated from our list of choice once its weakness is noticeable. Up to date, we have identified three approaches by which the KT can be represented for optimal processing:

- *The KT is a single variable with recent nodes visited:* Our DIFAP algorithm heavily rely on the number of states visited by the automaton in order to take relevant action on whether to use TD or KT algorithm. In some applications, the kind of strings input might be predefined. We chose therefore to use a single variable to represent the KT since it will only contain the most recent number of states visited by the recognizer. In other words, during the recognition process, the number of states visited is recorded and then saved in the KT variable at the *update* of the KT. We then use this information for later processing. The Algorithm checks the value in the KT and takes relevant decision. If the value is zero, it means that no state have been visited and therefore the suitable algorithm should be the TD. If the number of states is less than or equal to the threshold of efficiency established above, then HC is the suitable algorithm. Otherwise, we use the TD algorithm. This approach seems to be straightforward and very simple. However, one of its major drawback is that it is highly unpredictable as may result to a very inefficient framework.

- *The KT is a single variable with average nodes visited:* This approach is an alternative to the above. Using average node visited instead of most recent nodes visited reduce the probability of unpredictability but does not solve the entire problem. The approach is still therefore highly unpredictable.
- *The KT is an history structure:* This approach is still under investigation. We envisage it to be a structure containing detailed information on the history of each nodes of the automaton. The structure is aimed to be dynamic in the sense that, not all states can have a history if they have not yet been visited. The overall idea is to define a number of category of nodes such as *most likely to be visited*, *likely to be visited* and *unlikely to be visited*. The rate at which a state is visited is updated on a regular basis during the recognition process whether TD or HC was chosen. For each calculated rate of *visits*, the state can fall under each of the above defined category at any stage of the recognition. The role of the *check KT* routine will then be to probabilistically evaluate the number of states that are likely to be visited and take relevant action. Of course, the routine is still at a brainstorming stage and requires more though. However, this might be a better way to overcome efficiency problem of DIFAP as defined above.

5 Conclusion and Future Work

In this paper, we have made a review of the performance evaluation of both HC and TD algorithms already present in the literature. The HC algorithm outperforms the TD algorithm up to some threshold. Moreover, unlike the fact that the theoretical complexity of a recognizer is linear to the length of the input string. We have shown that the way the states of the automaton are visited at run-time plays a major role on the overall processing time. The more a state is visited the less he cache misses and the less the processing time. This observation helped us to have an idea on the way the KT of DIFAP can be implemented. We use a probabilistic concept to calculate the overall rate at which states are visited. This therefore yield to the actual implementation of DIFAP. However, many other challenges are still under investigation in order to make DIFAP a very flexible and efficient framework. The most important is the problem of mixed mode implementation of FAs whereby TD and HC are use simultaneously in order to provide a balance execution environment that uses small tables and small code.

References

[Kim02] Paul Kimmel. The Visual Basic .Net Developper's Book. Addison-Wesley, 2003.

[Ket03] E. Ketcha Ngassam. Hardcoding Finite Automata. MSC Dissertation. University of Pretoria, 2003.

[Kmp77] D. E Knuth and J.H Morris, Jr and V. R. Pratt. Fast Pattern Matching in Strings. SIAM J. Comput. Volume 6, 323-350, 1977.

[Kwk03a] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, Preliminary Experiments in Hardcoding Finite Automata, Poster paper, CIAA, Santa Barbara, 299-300, September 2003.

[Kwk03b] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, Hardcoding Finite State Automata Processing, SAICSIT, Johannesburg, 111-121, September 2003.

[Kwk04] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, A Framework for the Dynamic Implementation of Finite Automata for Performance Enhancement, PSC, Prague, September 2004.

[Nwk03] Noud De Beijer, Bruce W. Watson and Derrick G. Kourie, Stretching and Jamming of Automata, SAICIST, Johannesburg, 198-207, September 2003.

[Nr02] Gonzalo Navarro, and Mathieu Raffinot. Flexible Pattern Matching In String: Practical on-line search for texts and biological sequences. Cambridge University Press 2002.

[Wat95] Bruce W. Watson. Taxonomies and Toolkits of Regular Languages Algorithms. PhD Thesis. Technical University of Eindhoven, 1995.