Part III

Performance of DFA-based String Processors

CHAPTER 9

INTRODUCTION AND MOTIVATIONS

This part of the thesis discusses the performances of some of the various algorithms investigated in previous chapters.

A possible approach to evaluate the performance of a string processing algorithm would be to test it in against various data pertaining to well defined problem domains such as network intrusion detection systems, DNA analysis, natural and computer virus scanning, spell checking, etc. Instead, we rely on artificially generated data for performance analysis. The main motivation for using artificial data was the scope of the thesis. In effect, the main purpose of this work was to investigate implementation strategies for DFA-based string processing leading not only to the construction of a taxonomy graph, but also to the design of a toolkit that would be later implemented for string matching purpose. This approach laid down the basis for constructing a generalized taxonomy, and a toolkit. However, while this work has established the basis for having a reservoir of potentially useful algorithms, it has not paid very much attention to the conditions under which a toolkit user should opt to make use of a particular type of string processing algorithm. To establish these conditions would require that a thorough and methodical sequence of benchmarking tests be carried out on the various algorithms. Such an investigation constitutes a complete and sound research theme on its own right and cannot be carried out in the context of this present work.

In this part of the thesis, we discuss the various experiments conducted on artificially generated data. In the sections below, we present our approach used for performance measurement and data collection as well as the hardware and software support structure on which we relied to conduct the experiments. Then follows in the next chapter experiments conducted on a selected number of the algorithms.

9.1 The Software and Hardware Context

Any number of hardware platforms can be used to perform experiments in DFA-based string processing. Each of them will produce different results according to their various cache configurations, clock speeds, memory sizes, and the like.

The particular platform available to perform experiments described here was an Intel Pentium 4 at 1.8 GHz with 512 MB of RAM and 20 GB of hard drive. This processor was introduced onto the market in 2001. Its features include branch prediction buffering as well as a 20-instruction pipeline. In addition, it has two levels of cache memory, designated L1 and L2 respectively. At the first level there is the

L1 data cache of size 8KB and the L1 instruction cache (also referred to as the *trace cache buffer*), able to hold up to 12KB of so-called *micro instructions*. At the second level, the 256KB L2 cache is used for both data and instruction caching. The computer being made of a superscalar processor, the way in which these various architectural components combine together and the conditions under which they maximize processing speed were discussed in Chapter 2. The reader could refer, for example, to [Cor02, Gov01] for more details on the processor. The machine was equipped with the Linux operating system under which algorithms were implemented and tested.

All programs were written in Netwide Assembler (NASM). However, the gnu C++ compiler was used to generate source code for each of the algorithms to be tested.

The pseudo-code in Figure 9.1 provides a high-level view of the process followed to generate a NASM source file. Each generation depended on the DFA's number of states and the alphabet size, which are provided as input. As suggested in the figure, a transition table and input string are first generated, and then the required NASM code. The first two mentioned components were generated based on the pseudo-random number generator from [PTVF02]:

- **An artificial transition table:** `genTable(t,n,a)` generates an artificial $n \times a$ two-dimensional transition table t, whose rows are in the range $[0..n)$ and represent the states of the automaton; and whose columns are in the range $[0..a)$ and hold the alphabet indices. An alphabet size of 10 was chosen for our experiments. Thus, for each $q \in [0..n)$ and each $c \in [0..a)$, a random integer in the range $[0, n)$ was generated to represent $\delta(q, c)$. For simplicity, each transition $\delta(q, c)$ was considered valid. The generated transition table was therefore 100% dense. Therefore the transition table of each automaton was randomly constructed in the following sense:

  - Firstly, for each row, $i : [0, n-2]$, a column $j : [0, a)$ (corresponding to some alphabet symbol) is randomly selected. The cell (i, j) is assigned the next state transition value $i + 1$. This ensures that there is at least one string of length $n - 1$ that will traverse every state of the FA. We shall refer to this string as the *root* string of the particular automaton.
  - Next, all remaining cells of the table are assigned a random value in the range $[0, n - 1]$.

  This means that each node in the FA graph has a transition to the next state on some random symbol, as well as a transition on each of the remaining $a - 1$ alphabet symbols to some randomly determined state.

- **An artificial accepting string:** Having generated t, it was now required to randomly generate an *accepting* string s associated with the artificial automaton defined by t. We specifically required an accepting strings, because it makes more sense to experiment with accepting strings rather than rejecting ones. The latter lead to the algorithm terminating after an unpredictable number of iterations. From the point of view of the experiments to be described later, this represents an unnecessary source of variation that adds nothing of value

to the experiments. In the algorithm in Figure 9.1, $genStr(t, s, l)$ was used to generate a string s of length l based on the transition table t. The length of the string was intentionally chosen to reflect the experiment being conducted. In general, most of the algorithms discussed in this thesis were designed to provide better results when processing large strings. Therefore, most of the strings generated were of length $4n$. The generation of an accepting string was done by randomly constructing a *string path* from t, and subsequently determining the symbols that fall in the randomly constructed string path in order to form the whole string. For our experiments, the first n symbols were chosen as previously described, and the remaining substring of length $3n$ was simply 3 replicas of the first substring of s. This approach of choosing our accepting strings helped ensured that when the first n symbols of the string are processed, most of the states that form part of the string path have already been visited.

- **The Assembly source code:** Having randomly generated both the accepting string and the transition table. A function $genCode(t, n, a, s)$ was used to generate the NASM source code of the algorithm under investigation. The algorithm could be any of the TD, HC or MM algorithms previously discussed. For every single algorithm corresponded a particular logic regarding its generation. Therefore, some algorithms required more parameters than the others, simply because of the chosen implementation strategy. Of course it should be noted that when invoked for TD-based algorithms $genCode()$ produced the assembly version of a transition table, whereas when invoked for HC-based algorithms, t was converted into directly executable instructions. For the MM-based algorithms the function was invoked not only with the transition table, but also with the number of hardcoded states and TD states in order to establish which states were supposed to be hardcoded and which were TD. For example $genCode(t, |\mathcal{Q}_t|, |\mathcal{Q}_h|, n, a, s)$ would produce a MM algorithm whereby the first $|\mathcal{Q}_t|$ states are table-driven and the remaining $|\mathcal{Q}_h|$ states are HC; in this case, $n = |\mathcal{Q}_t| + |\mathcal{Q}_h|$.

The generated NASM source code (*source.asm*) was further compiled and translated into executable before being executed. Our goal in the execution of such programs was to collect the input string processing time for each generated algorithm. Therefore, each assembly source code was equipped with the Intel function that reads the time stamp counter at the beginning of acceptance testing, and also at the end, followed by the proper calculation of the number of clocks cycles required to accept the string. On Intel microprocessors, the so called time stamp counter keeps accurate count on every cycle that occurs in the processor. The time stamp counter is a 64 bit model specific register (MSR) which is incremented at every clock cycle [Cor02]. Whether invoked directly as a low-level assembly instructions, or via some high-level language instruction, the RDTSC instruction allows one to read the time stamp counter, and thus to determine approximately the time taken to execute critical sections of code. In the next section, we briefly discuss performance measurement of the generated programs.

```
GEN(n,a)
▷ Input: n: number of states, a: alphabet size
▷ Output: source.asm a NASM file
▷ Auxiliary: t[0 : n)[0 : a): table, s string
1 Begin
2    t[0 : n)[0 : a) ← −1
3    s ← nul
4    l ← value
5    genTable(t, n, a)
6    genStr(t, s, l)
7    genCode(t, n, a, s, source.asm)
8    Return source.ams
9 End
```

Figure 9.1. The structure of a NASM code generator

9.2 Performance Measurement

The measurement of the performance of programs was straightforward. Each program was run 50 times, and the minimum processing time in clock cycles (ccs) was recorded for randomly generated DFAs whose state size spanned different ranges. In general, up to 120 different DFAs were generated for each algorithm being tested. The size of the generated DFAs ranged from 100 states to 12000 states, with increment of 100. Each automaton generated was based on a 10 alphabet symbols. As mentioned earlier, associated with each generated automaton was an accepting string of length $4n$, such that the first n symbols drove the automaton through most of the states falling in the string path, an the remaining $3n$ symbols occasionally visited states that where not previously visited while processing the first n symbols. The data collected were then used for cross comparison with those collected for other algorithms under study. Our aim was to investigate the extent to which new algorithms may perform in relation to one another and, specifically, in relation to their core counterparts.

9.3 Summary of the Chapter

In this chapter, we have presented the software and hardware context under which the algorithms were investigated. We also presented, a generic function for the generation of NASM source code used to produce directly executable DFA-based recognizers, where it was shown that the kind of program to be generated depends on the parameters provided to the generator according to the implementation strategy in used, be it core algorithms or algorithms based on implementation strategies. Also briefly mentioned in this chapter was the kind of string used to conduct experiments. In effect for most of our experiments we relied on the string of the form $4n$ which in our opinion were good examples to exercise cache's temporal and spatial locality

of reference exploited by our algorithms. The experimental results of some selected algorithms are presented in the next chapter.

CHAPTER 10

EXPERIMENTAL RESULTS

10.1 Introduction

This chapter presents the experimental results of some of the algorithms whose descriptions were provided in previous chapters. The results are presented in the form of graphs representing the performance of the selected algorithms.

The chapter starts off by presenting the performance of all the new table-driven algorithms compared to their core TD counterpart. Then follow discussions on the performance of some selected HC algorithms, namely the bounded and unbounded HC-DSA, the HC-SpO and the HC-AVC compared to the core HC. Also briefly discussed in this chapter is the performance of the core mixed-mode algorithm where it is shown that the algorithm could be used as a performance *booster* when testing strings whose string path frequently falls in one of the HC or TD portion of the algorithm.

As a disclaimer to the reader, our intention in this chapter is not to provide a complete analysis of the algorithms investigated throughout the thesis. In effect, each algorithm could be viewed as having a best case behavior where it could be processed at optimum as long as we define appropriate kind of strings suitable for the algorithm. However, performing such investigation is a matter of intensive research on various computational mediums, operating systems and of course, programming languages. Also, since our algorithms were explicitly designed to exploit the notion of cache memory's spatial and temporal locality of reference, various other factors such as alphabet size, number of states, the threshold of the dedicated memory (or virtual cache) spaces, and also the appropriate replacement policies to be used are matters of intensive investigations and could not be dealt with in the scope of this thesis. We thus envisage to further such investigations as future directions to this research. Nevertheless, sufficient experimentation has been carried to show the viability of each of the strategies, and to suggest certain directions for further research to probe their optimal behaviour.

Experiments conducted on TD algorithms are provided in the next section.

10.2 The Table-driven Experiments

Various experiments were conducted in order to compare the performance of the derived implementation strategies relative to the core TD implementation approach. Again, as mentioned earlier, we merely relied on artificial data instead of real life data pertaining to well known DFA-based applications such as such as network intrusion

detection, natural and computer virus scanning, spell checking, tandem repeat finding, etc. This should be seen as a first approach to establish the drawbacks of our algorithms, a more detailed investigation being left for future work.

The TD algorithms were implemented using the Netwide Assembly language (NASM), under the Linux OS, on an Intel Pentium 4 machine. For each algorithm under investigation, 120 different automata were generated of size ranging from 100 to 12000 states, with increment of 100. Each DFA was based on a 10 alphabet symbols. Associated with each recognizer was an accepting string of length $4n$, where n is the number of states of the automaton. The strings generated were explicitly designed such that the state path due to the first n symbols was a sequence of randomly chosen automaton states. Those symbols were repeated 4 times as to *occasionally* take advantage of spatial and temporal locality of references at runtime since (as we will discuss in the paragraph below) the number of newly visited states declined from one segment of the string to the next.

Before discussing the timing results, consider the information presented in Table 10.1. The table gives an overview of the rate at which states are *visited for the first time* for each automaton generated within the the full data set of 120 automata. Data is given as a percentage of the total number of states in each particular run. The first column relates to the full string that was processed, the second column, to the first segment, etc. Thus, after processing the first segment, approximately 60% of the states have already been visited for the first time. Note that these observations lie in a fairly narrow band, between about 57% and 63%. As a matter of fact, when the number of visited states is plotted against the automaton size for the first segment (Figure 10.1-I), a very distinct linear trend is observed. However, in the case of segments 2 to 4 (Figure 10.1-II to Figure 10.1-IV), no obvious trend is observed in relation to the automaton size. Nevertheless, the average number of visited states declines steadily from about 16% in the case of segment 2, to almost 0% in the case of segment 4. Overall, about 80% of states were visited, on average.

	Full String	Segment 1	Segment 2	Segment 3	Segment 4
Maximum	95.20%	62.67%	24.80%	9.17%	2.78%
Minimum	62.42%	56.80%	0.40%	0.00%	0.00%
Average	79.06%	61.29%	16.11%	1.60%	0.06%

Table 10.1. Rate at which states are visited for the first time

These results are broadly in line with expectation. They imply the following: that in processing the first n symbols, the roughly 60% of visited states will be located contiguously in memory in order of first usage when either the DSA or AVC strategy (or any combination of those strategies) is used. To this extent, the data is optimized in terms of the spatial locality of reference principle. In the processing of later segments, fewer states are newly visited, and consequently, when the algorithm involves DSA or AVC, more time is spent in the *hot-spot* part of the code. If these later segments were to traverse the previously visited states in *exactly* the same sequence as the first segment, then the probability would be relatively high of accessing spatially

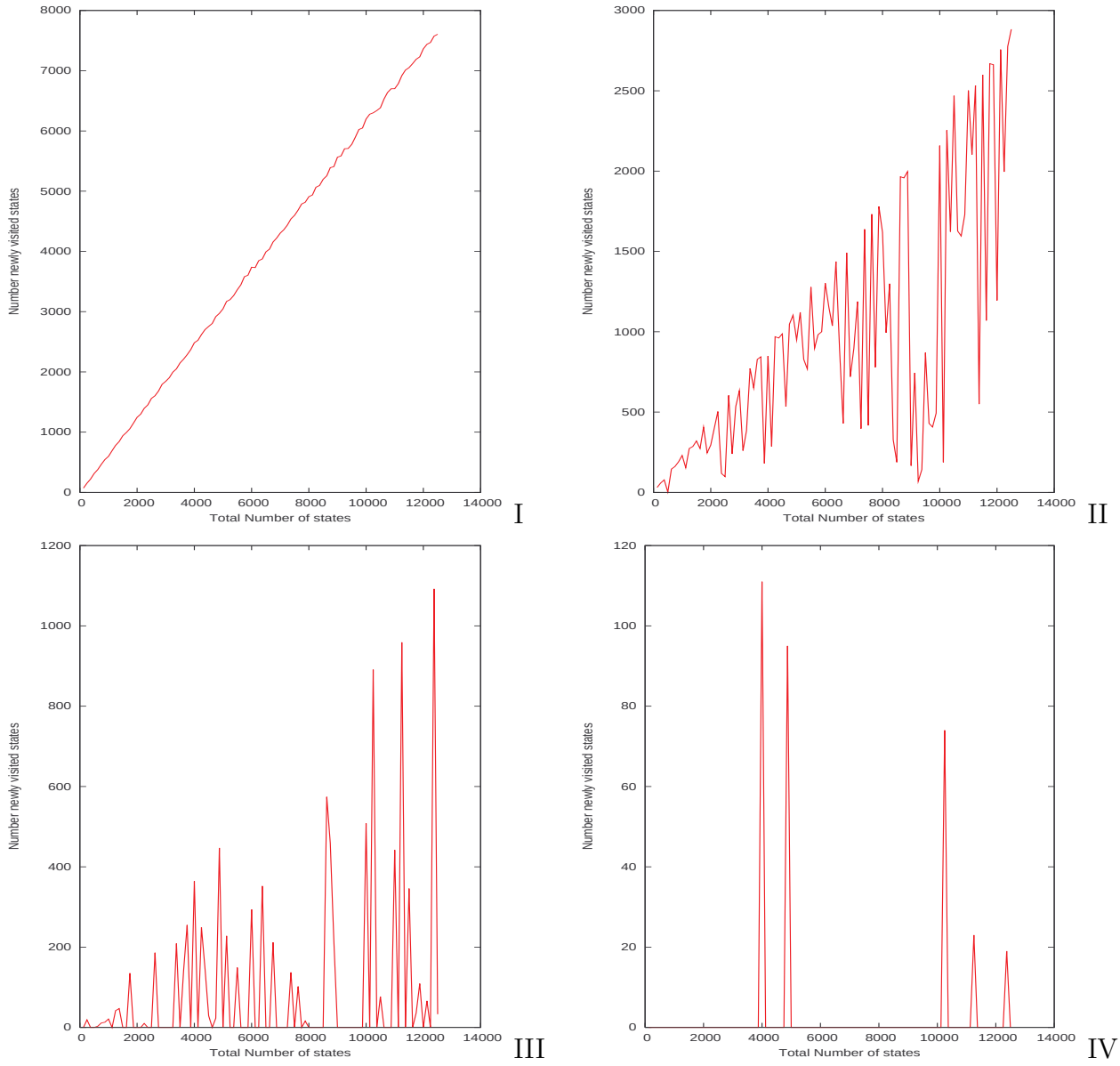

Figure 10.1. Total number of states vs: number of newly visited states in segment 1, 2 , 3 and 4 (I, II, III & IV)

localized data, and hence of triggering few cache swaps. Of course, this will only happen in the unlikely event that segment 2 (and therefore also 3 and 4) happen to start off in state 0.

The experiment above has not been designed to specifically generate this "best case" scenario. Rather, it is far more likely that these later segments (i.e. segments 2, 3 and 4) will start off in some other random state. Nevertheless, on the evidence of Table 10.1, an increasingly large proportion of segment 2 to 4 processing is via the

hot-spot. In fact, even in the case of the first segment's processing, about 40% of the iterations were through the hot-spot. Whether this translates into time gains as a result of frequently accessing spatially localized data (and therefore having fewer cache swaps), cannot be predicted *a priori*. To this end, we require the timing data derived from running the respective algorithms.

For the purposes of recording timing data, each experiment to be described below was repeated 50 times for each set of input data, and the processing time was recorded in clock cycles (ccs). Furthermore, each processing time was divided by the corresponding automaton's number of states in order to report on *time per states* instead of the overall processing time. For further analysis, we relied on the *minimum time per state* of these 50 observations[1].

For each algorithm involving the SpO strategy, the array of states' positions $p_{[0..n)}$ was randomly generated to represent the "prior knowledge" of the position of the states before acceptance testing.

In order to evaluate the algorithms performance against the core TD algorithm, each data item collected was plotted against the data of the core TD algorithm using Gnuplot.

For the bounded TD-DSA and the TD-AVC algorithms, we somewhat arbitrarily chose a bound of 50% of the number of states. That is, for an automaton of size n, up to $\lceil n/2 \rceil$ states were processed in the allocated dynamic memory or in the virtual cache. It is a matter of future research to search for heuristics that indicate what a reasonable bound size will be when implementing these strategies.

The graphs in Figure 10.2 depict the performance of the core TD algorithm against the bounded TD-DSA, the unbounded TD-DSA, the TD-SpO, and the TD-AVC algorithm respectively.

The performance of the unbounded TD-DSA algorithm depicted in Figure 10.2-II, shows that the unbounded TD-DSA algorithm competes with its core TD counterpart. In facts there is an improvement of roughly 21 ccs of the TD-DSA algorithm over the core TD algorithm for automata of size less than 2000 states; also an improvement of about 18 ccs is observed between about 7700 states and 1200 states. However, the core TD algorithm is roughly 87 ccs faster than the unbounded TD-DSA algorithm for automata of size greater than 2000 states, but less than 7700 states. A plausible explanation for this is of course the fact that the string is large enough as previously described, and the number of dynamically allocated states decreases from one segment of the string to the next, resulting in the processing of the string at hot-spot and hence, better performance of the unbounded DSA strategy. In contrast, the TD-algorithm is

[1]Earlier studies had shown that there is in fact quite a lot of variation in the number of clock cycles used from one observation to the next. It is not within the scope of this study to establish the precise reason for these variations. We assumed that they are due to operating system and CPU overheads: for example round robin checking for context switching on the Linux OS, the effects of the pipelined architecture and the branch predication on the CPU chip, etc. Nevertheless, in these earlier studies it had been found that occasionally outlier data is generated. To incorporate this outlier data into an average would drastically distort results. Hence, it was decided to rely on the minimum of the 50 values obtained, instead of on their average, to characterise each particular experiment undertaken [NWK03b].

about 300 ccs faster than the bounded TD-DSA algorithm when processing the same kind of strings. A justification could indeed be the fact that the bounded nature of the algorithm requires frequent state replacement during acceptance testing when the dynamically allocated space is full. Therefore, the following scenario may be envisaged for defining a best case scenario of the bounded TD-DSA algorithm:

- *Replacement policy*: We have chosen to use the direct mapping policy in order to swap a state in and out of the allocated free memory space when no more space is available. Such policy may not always yield better cache placement and data organization since in certain circumstances, some states are frequently swopped in and out of the cache, resulting in poor performance of the algorithm. A policy such as the associative mapping or the LRU policy are likely to be better alternatives for avoiding such drawback [PH05]. Moreover, a replacement policy could be avoided totally by performing acceptance within the transition table when reference is made to a state out of the allocated dynamic space, provided that the threshold for state allocation has been reached.

- *Kind of string*: Table 10.1 shows that up to 80% of the automaton's number of states were accessed during acceptance testing, when processing the kind of strings used for the experiment. Therefore, dedicating only 50% of those states to the dynamic memory space does not guarantee better data organization in the sense that a significant number of states would have not been visited before reaching the threshold. Thus, when defining the best case scenario for the bounded TD-DSA, both the threshold used to defined the allocated dynamic memory and the frequency at which new states are visited should be considered.

In an attempt to take into account the fact that replacement policy is a performance bottleneck, the TD-AVC algorithm was implemented such that no replacement was made when the virtual cache was full. This approach resulted in TD-AVC competing with its core TD counterpart as shown in Figure 10.2-III; the graphs show that for automata less than 2500 states, the TD-AVC algorithm is roughly 64 ccs faster than the core TD, whereas above that region the core TD algorithm outperforms the TD-AVC algorithm only with about 35 ccs. Again, since up to 80% of the automaton's state are visited, we merely have 30% of the states that are processed out of the cache whiles the remaining are processed in cache. This observation shows that the strategy is of interest. However, the kind of data that should provide better cache utilization must be further investigated. Moreover, even better performance of the TD-AVC over the core TD is expected if the cache size increases at about 80% provided that the kind of strings being tested are made of segments whereby after processing the first one, all the states that fall in the string-path are already available in the cache (since the remaining segments of the string would be processed at hot-spot).

The graphs in Figure 10.2-(IV) also reveal that the TD-SpO is approximately 195 ccs faster than its TD counterpart. The result is indeed as expected since we are not taking into account the time taken to reorder the states. Moreover, the same result is expected no matter the kind of string or automaton to be processed, since once

data is well organized, processing will always be at optimum because the cache enjoys spatial and temporal locality of references.

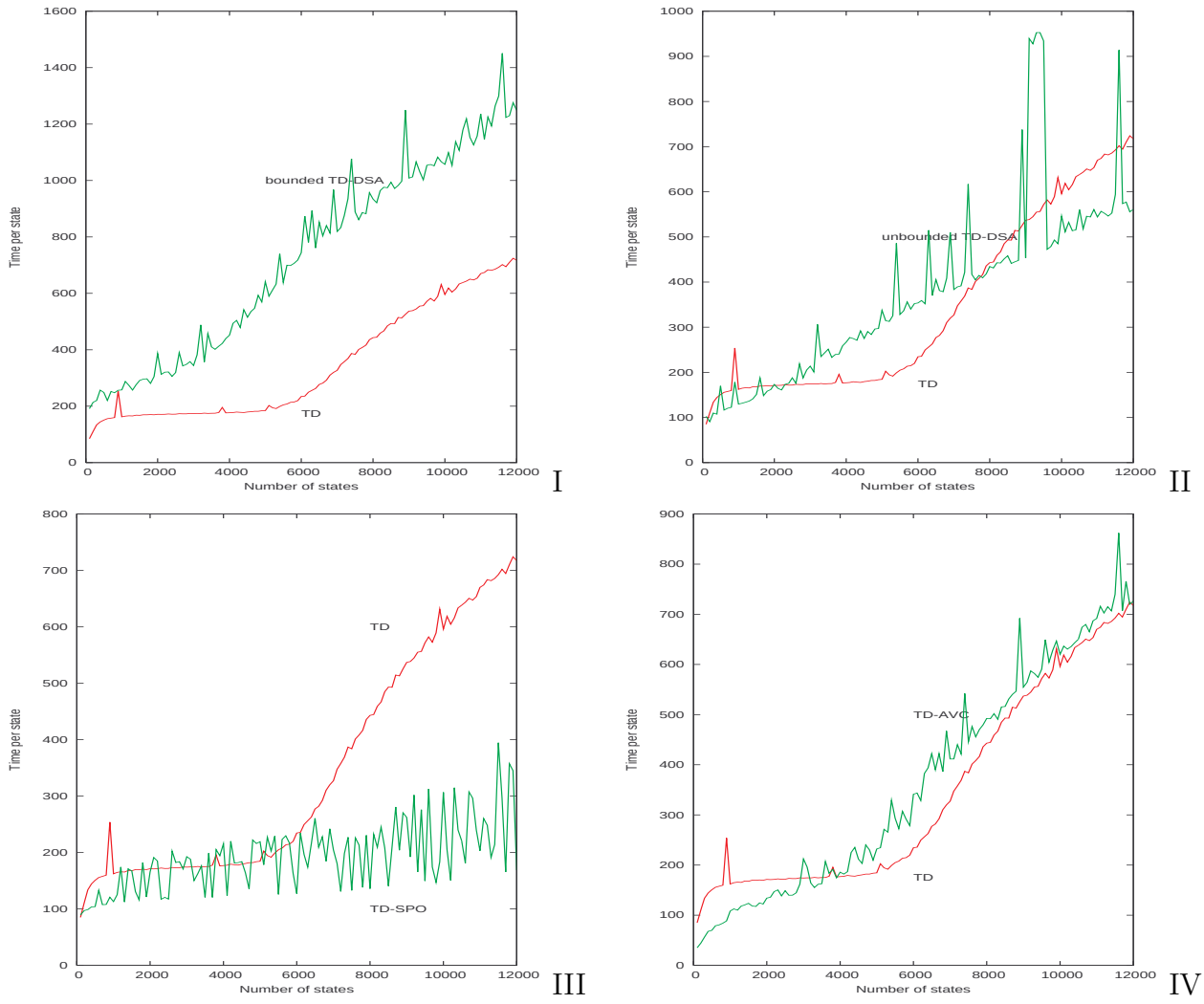

Figure 10.2. Performances of TD vs: DSA (I & II), SPO (III), and AVC (IV)

Figure 10.3 depicts graphs representing the performance of the various algorithms obtained by combining implementation strategies. As we may observe, the combination of the SpO strategy with the unbounded TD-DSA yields even better performance over the core TD algorithm (more than 54 ccs improvement). This also applies to the algorithm t_{23} (i.e. the combination of the SpO and the AVC strategies) —This version used direct mapping for replacement.— For those algorithms, the pre-ordering of states has resulted in better data organization, and hence better cache utilization. However, for algorithm t_{u123}, the AVC strategy relied on direct mapping for replacement, resulting in poor cache utilization, and hence poor performance. It is expected that by choosing suitable strings, a better performance may be observed.

For algorithm t_{b123}, the bounded nature of both DSA and AVC strategies required replacement, which is a source of overheads; this explains its poor performance in relation to the TD algorithm. Again, suitable data set would produce better results. The combination of both the DSA strategy and the AVC strategy also suffered from the overheads caused by the direct mapping replacement policy. Therefore, in order to improve the performance, a better replacement policy should be chosen or we may even avoid it totally.

The performance of some of the hardcoded algorithms is discussed in the next section.

10.3 The Hardcoded Experiments

Based on the results obtained from the TD-experiments, it seemed reasonable not to repeat experiments on *all* the HC algorithms. In effect, the TD experiments revealed the following:

- The SpO strategy appears to be the best of all (although in some circumstances the unbounded DSA and AVC strategies seemed to improve the time per state quite significantly), and its association with any of the other strategies or combination of strategies further improves the performance of the derived algorithm;
- The performance of most of the algorithms obtained by combining the various strategies (SpO-AVC, DSA-SpO-AVC, DSA-AVC) need to be further investigated. Such future research should seek to establish whether and under what conditions —i.e. for which kinds of data sets— would these combined strategies be advantageous.

Thus, the algorithms that most interesting candidates for further experimentation in relation to HC are: the bounded and unbounded HC-DSA, the HC-SpO and the HC-AVC. The collection of data pertaining to the hardcoded experiments was similar to that of the TD experiments discussed in the previous section. The data collected for the chosen algorithms were plotted against that of the core HC algorithm. As depicted in Figure 10.4, each new algorithm in general outperforms its associated core counterpart.

Figure 10.4(I), depicts the graphs showing the performance of the core HC algorithm and that of the unbounded HC-DSA algorithm. The graphs clearly show that the unbounded HC-DSA outperforms the core HC algorithm for automata of size greater than 4000 states. In effect, since we are concern in this experiment with large strings, it seems plausible that for smaller automata, the cost of overhead remains noticeable and could not be absorbed by that of the processing of the entire string. However, when dealing with automata of considerable size (more than 4000 states), the cost is now absorbed by that of the processing of the whole string, resulting therefore in better performance of the unbounded HC-DSA algorithm. Moreover, the core HC algorithm is highly affected by the number of instructions that make up the whole automaton. Caching effects are noticed because of the high probability of cache misses that could occurs, resulting therefore in poor performance.

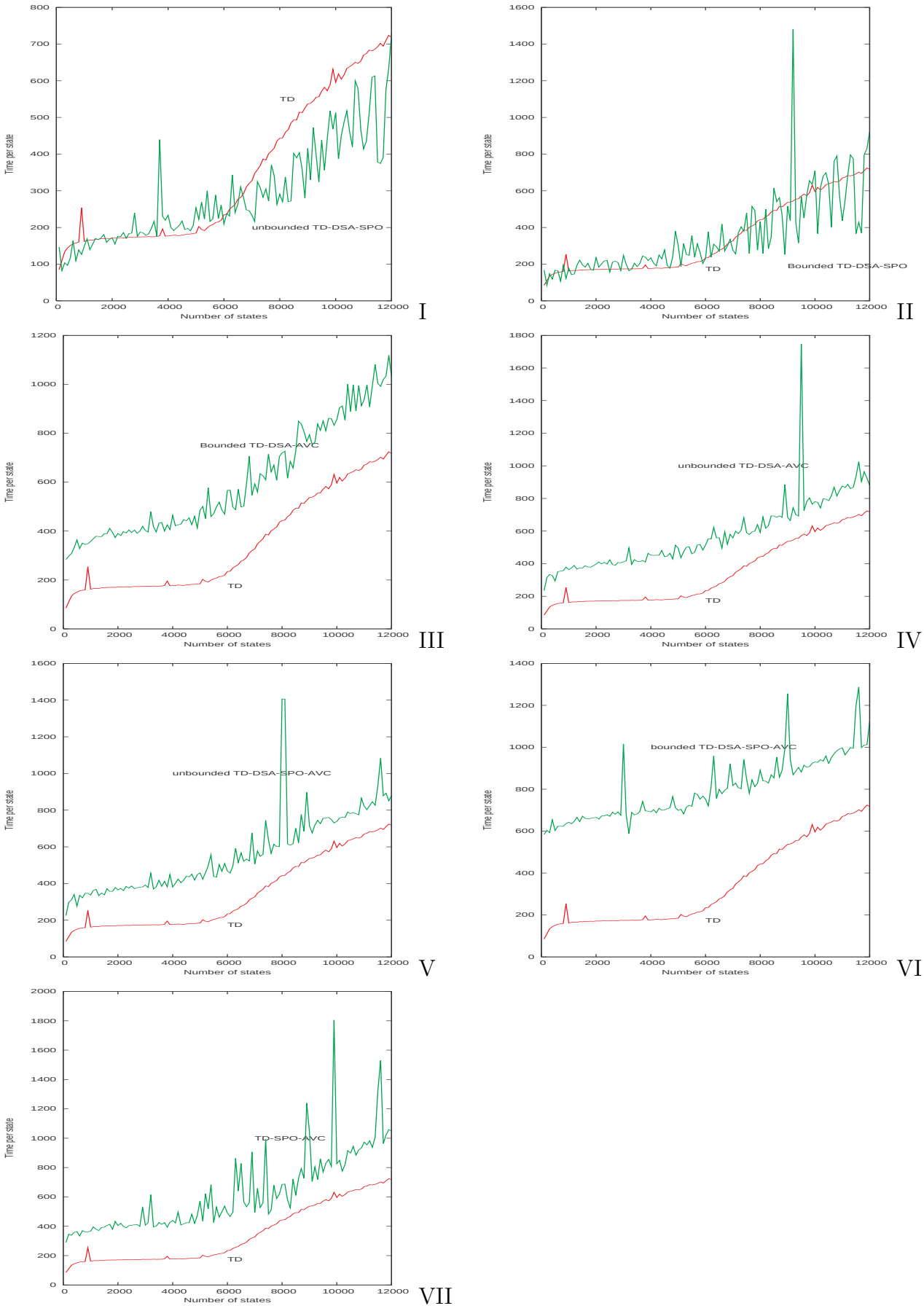

Figure 10.3. Performances of TD vs: DSA-SpO (I & II), DSA-AVC (III & IV), DSA-SpO-AVC (V & VI), and SpO-AVC (VII)

While the unbounded HC algorithm does not provide any restriction on the number of states to be dynamically allocated in the new memory space, the bounded HC-DSA makes provision for the threshold for dynamic allocation. In our experiment, the bounded algorithm was designed such that only 50% of the total number of states could be dynamically allocated. Recall that the experiment was designed such that an average of 80% of new states were visited during acceptance testing. Since it was already known from the table-driven experiment that the bounded DSA was hampered by the replacement policy adopted, we choose for the present experiment not to use a replacement policy. Therefore, when the dynamically allocated memory was full, no replacement was carried out and acceptance testing simply took place in the table. Figure 10.4(II) depicts the graphs for both HC and bounded HC-DSA. The graphs clearly show that the bounded algorithm now outperforms the core HC algorithm for automata of size greater than 2000 states. This is explained by the fact that the bounded nature of the algorithm has freed us from incurring more overheads during acceptance testing, which results in an improvement of the DSA strategy in terms of processing time. The graphs show that under such conditions although both algorithms (bounded and unbounded DSA) outperform their HC counterpart for an FA made up of a large number of states, the bounded DSA algorithm now has become more efficient than its unbounded counterpart simply because we have chosen to avoid the use of a replacement policy. This approach raises the problem of optimality of the replacement policy to be chosen during acceptance testing in that, if not well chosen, we could end up with inefficient algorithms. Therefore, further investigations need to be conducted on not only the size of the dynamic memory to be allocated in the bounded case, but also on the appropriate replacement policy to be used when the reserved memory is full. However, all these issues are left to future work.

The comparison of the HC algorithm and the HC-SpO is depicted is Figure 10.4(III). The random access nature of the hardcoded states when processing strings using the core HC algorithm usually results in high probability of cache misses and hence poor performance. However, when the implementer has some prior knowledge on the order in which states would be visited, the SpO strategy yields even better performance as shown in the graphs. As for the table-driven case, the association of the SpO strategy with either of the DSA strategies would definitely result in better performance in that, the derived algorithm would exploit the strengths of both strategies as previously explained.

Figure 10.4(IV) depicts the performance of both HC and HC-AVC algorithms. Unlike the TD-AVC algorithm whose performance was closer to that of the core TD algorithm, a threshold of efficiency of the HC-AVC algorithm over its HC counterpart is observed as from about 8000 states. A plausible explanation of this observation lies in the number of operations required to perform states swapping whenever this is necessary. In effect, unlike its TD counterpart, that requires states swapping by a complete interchange of data, in hardcoding, only what we referred to as *necessary fields* (see Section 6.4 of Chapter 6) are modified resulting in the minimization of the overall time penalties due to overheads.

Acceptance testing is handled by a driver function that ensures that the state being processed is available in the cache when the cache is not full. When the cache

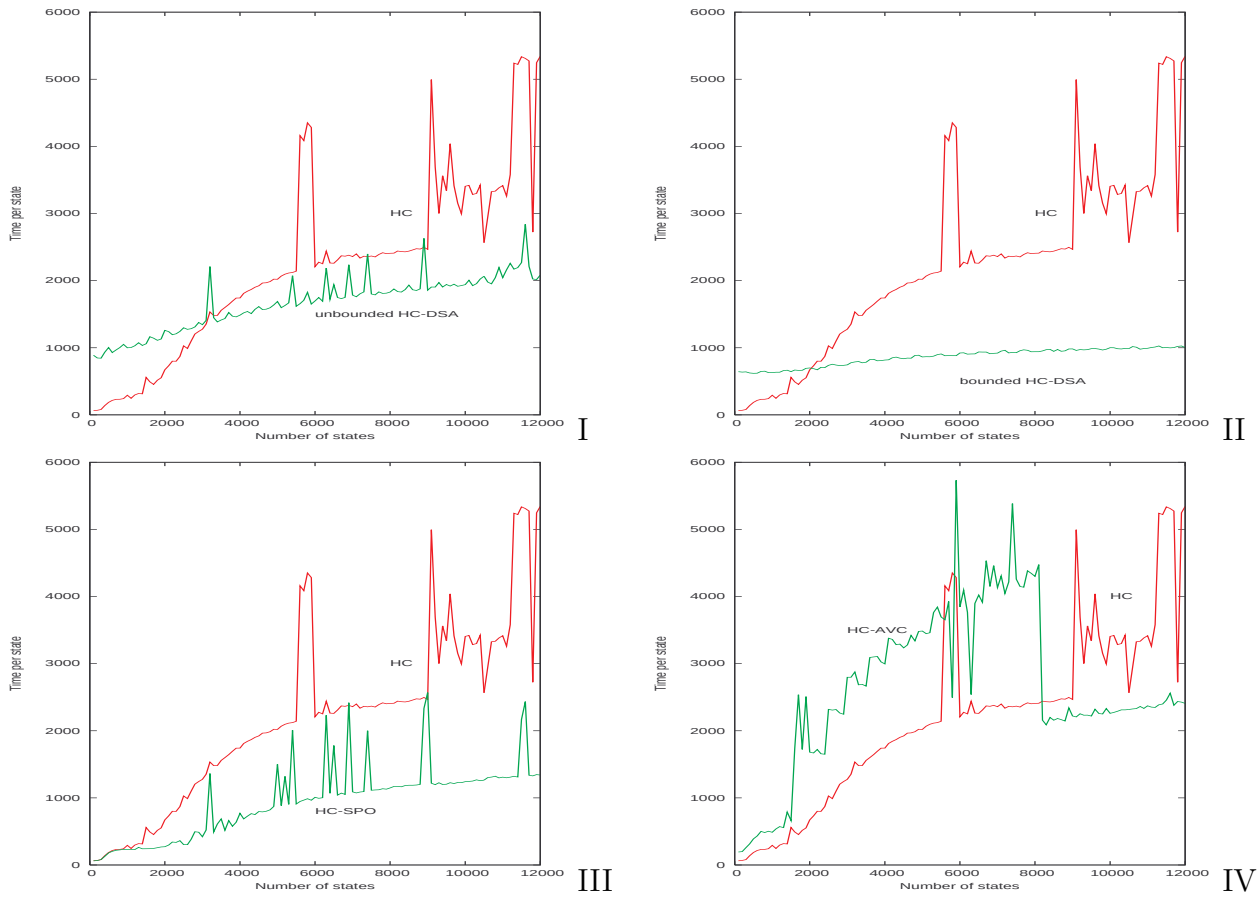

Figure 10.4. Performances of HC vs: HC-DSA (I & II), SpO (III), and AVC (IV)

is full, acceptance testing takes place in the hardcoded portion of the state, be it out of the cache or not. Therefore, we have again chosen not to adopt any replacement for this algorithm since further investigations need to be conducted in order to define the appropriate algorithm to be used for state replacement. Since the SpO strategy has proven to improve the performance when associated with other strategies, it is thus expected that the HC-SpO-AVC strategy would yield even better performance compared to that of the core HC algorithm.

In general, the graphs in the figure have shown that the new algorithms outperform their core hardcoded counterpart. However we have restricted ourselves in these experiments to the processing of strings made of long sequences. Furthermore, the strings were designed only to reflect the good, if not the best, case scenarios for the DSA strategies. It turns out that they are also good case scenarios for the other strategies. We have thus chosen to leave experimentation with other kinds of strings for future work. Such an investigation would require intensive analysis of each individual algorithm on its own before drawing appropriate conclusions.

In the next section, we discuss the performance of the core mixed-mode algorithm.

10.4 The Mixed-mode Experiments

The mixed-mode algorithm is an algorithm explicitly designed to take advantage of the capabilities offered by the TD and HC algorithms. As previously discussed, the algorithm is subdivided in a TD part used to processed states that are table-driven, and a HC portion made of directly executables states. When reference is made to a given state, a test is first made as to check whether the state is hardcoded or not. If hardcoded, the directly executable instructions that make up the current state is invoked; otherwise, the state is processed through the table. It follows that for the mixed-mode algorithm, if carefully designed, it may further improve performance, depending on the kind of string under consideration. Several matters should be taken into account when implementing the mixed-mode algorithm:

- *The number of table-driven states*: Previous experiments revealed that the table-driven algorithm enjoys better performance when the states being visited are contiguously organized so as to take advantage of temporal and spatial locality of reference. Therefore, if the number of TD states are such that states are contiguous and it happens that those states are frequently in the cache, the overall performance of the string would be optimal in that the string-path falls within the TD portion of the algorithm.

- *The number of hardcoded states*: As previously discussed, the performance of the hardcoded states are usually hampered by the number of directly executable instructions. It follows that if the mixed-mode algorithm contains a large number of hardcoded states, the performance would be negatively affected in that, frequent accesses to those states would result in several cache missed and hence poor performance. In order to have a mixed-mode algorithm that takes advantage of hardcoding, the implementer should ensure that the number of hardcoded states is able to fit in the instruction cache. In this case, no matter the kind of string being tested, the algorithm would be processed at optimum.

- *The rate at which HC/TD states are visited*: In order ensure optimal performance in mixed-mode, it is of importance to have a balanced definition between hardcoded states and table-driven states. If the number of hardcoded states is very large, the program would experience several cache misses in the instruction cache. However, if that number is reasonable such that all the states fit into instruction cache, the hardcoded states would be processed at optimum and would thus result in better performance. Moreover, the portion of the code related to TD must be organized as to take advantage of temporal and spatial locality of reference in order to ensure fast processing. It follows that, the kind of string being tested for acceptance is a matter of interest when dealing with mixed-mode. In effect with a mixed mode algorithm whose string path frequently falls in the hardcoded portion, if those instructions are able to fit in the cache, the algorithm would enjoy optimum performance. In the same way, when the string path frequently falls in the TD portion, then the algorithm

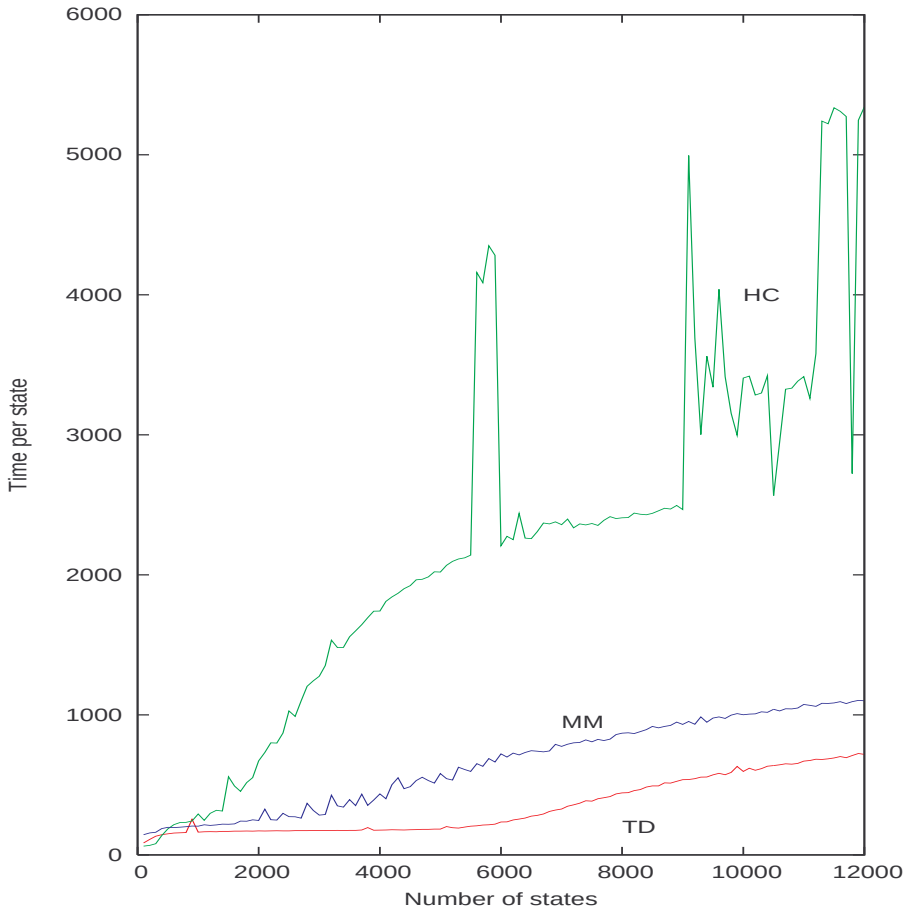

Figure 10.5. Performances of HC, TD and MM (where 25% of the states are HC)

would enjoy optimum performance provided that the TD states are visited on a contiguous fashion.

For the present study, we have chosen to show the reader the advantage of using the mixed-mode algorithm as a performance *booster*. To this end, we relied on the same kind of strings and automata used in previous experiments. Since for about 80% of the states are frequently visited, we dedicated 75% of the total number of states to the TD portion of the mixed-mode and 25% to the HC part. However, the states (HC or TD) were not specifically organised in a way that would take advantage of spatial and temporal locality of reference. Figure 10.5 depicts the graphs for the performance of each of the core algorithms (TD, HC, and MM), with the MM algorithm designed as previously described. As we may observe, for automata made of large number of states, the TD algorithm tends to be the best but the gap between MM and HC is fairly narrow. An obvious approach that could be used to improve MM in this context would be to use MM-SPO (not discussed here) since the states would be organized as to enjoy cache's locality of reference.

An attempt to improve the performance of the mixed-mode algorithm under the same conditions was also investigated. To this end, we still assigned 25% of the entire automaton to the HC part of the algorithm and 75% to the TD part. However, frequently visited states ranged between 0 and $75 \times n/100$. Therefore a limited number of HC states were part of the string path whereas the majority of states that form part of the string path belonged to the TD portion of the algorithm. As depicted by the graphs in Figure 10.6, MM algorithm is on average more than 150 ccs faster than the core TD algorithm. The experiment clearly shows that the MM algorithm may indeed act as a performance booster when implementing recognizers, provided that the kind of string being tested is well investigated and that there is a suitable split between the states to be table-driven and those to be hardcoded.

It should also be noted that the experiment designed as such was not meant to test the best case behaviour of the MM algorithm. In order to indeed investigate its best case scenario, attention should be given on the kind of the data being processed as well as the order in which both HC and TD states are organized in memory. Such investigations inevitably involve the study of the various MM algorithms which rely on various implementation strategies. Since the scope of this thesis does not provide for the complete study of the various MM algorithms, the issue is left as a matter of future work.

10.5 Summary of the Chapter

In this chapter, we have discussed the performance of some selected algorithms studied in the previous parts of the thesis. The chapter was not intended to fully study algorithms performance, but rather to show that all algorithms investigated throughout the thesis are of interest and thus to suggest that further studies should be carried out in order to investigate the best case behaviour of each. Experiments on the TD algorithms revealed that most of the algorithms based on individual implementation strategies could be more efficient than their core TD counterparts as long as the suitable kind of string to be processed are use in the acceptance testing. Moreover, several constraints (such as the threshold to be used for the size of the different ad-hoc memory utilized in the algorithms as well as the so-called replacement policy) need to be further investigated .

Experiments conducted on HC algorithms were in-line with expectations in the sense that almost the same results were obtained for the TD experiments. Therefore, in general the HC algorithms to which strategies or combination of strategies are associated outperform their core HC counterpart. However, an observation of the HC graphs and that of TD shows the magnitude of the time it takes to accept the kind of strings under study when HC algorithms are used is far higher than that of the TD algorithms for large automata. Of course previous studies ([Nga03]) have already revealed that HC outperforms TD for automata in the order of hundred states. This clearly suggests that the TD algorithms remain the best algorithms to be used when processing the kind of strings considered for the present experiments. However, we are confident that the performance of hardcoding could be improved if the associated

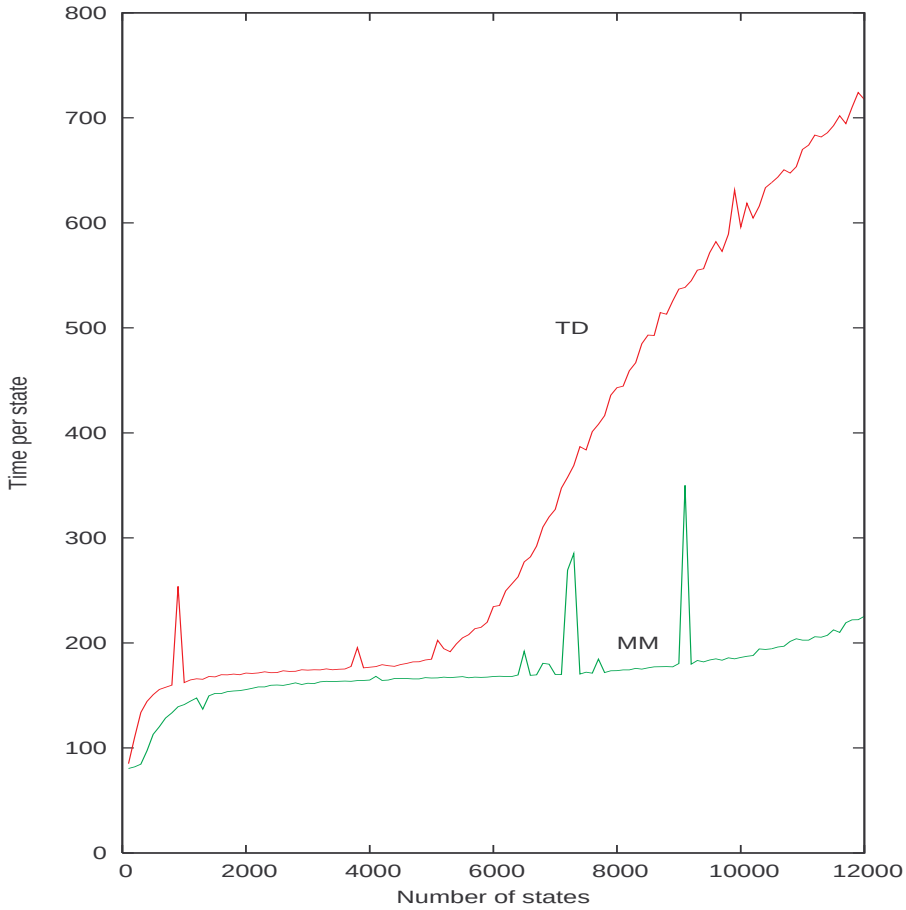

Figure 10.6. Performances of TD and MM (where 25% of the states are HC, and the first 75% of states are frequently visited)

code written in assembler are fully optimized. But these issues will not be further investigated in this context.

The mixed-mode experiment was conducted only on a single algorithm. This was because we believe that the study of all the mixed-mode algorithms derived from this work should constitute a complete research topic on its own right. However, experiments clearly revealed that the core MM algorithm remains the best algorithm between all the core algorithms provided that the string path followed during acceptance testing is fairly balanced between HC and TD states.

The experiments conducted in this chapter were far from being an exhaustive task; they were merely aimed at showing the importance of the algorithms discussed throughout the thesis. Much remains to be done, for example by exploring the use and performance of the various algorithms on different platforms, by applying real-life data pertaining to existing problem domains, etc.

The conclusion and further direction to this thesis are provided in the next part.