On Implementation and Performance of Table-Driven DFA-Based String Processors

Ernest Ketcha Ngassam[1], Derrick G. Kourie[2,3], and Bruce W. Watson[2,3]

[1] School of Computing, University of South Africa, Pretoria 0003, South Africa,
ngassek@unisa.ac.za

[2] Department of Computer Science, University of Pretoria, South Africa,
{dkourie,watson}@cs.up.ac.za

[3] FASTAR Research Group (www.fastar.org)
{eketcha,derrick,bruce}@fastar.org

Abstract. Table-driven (TD) DFA-based string processing algorithms are examined from a number of vantage points. Firstly, various strategies for implementing such algorithms in a cache-efficient manner are identified. The denotational semantics of such algorithms is encapsulated in a function whose various arguments are associated with each implementation strategy. This formal view of the implementation strategies suggests twelve different algorithms, each blending together the implementation strategies in a particular way. The performance of these algorithms is examined in against a set of artificially generated data. Results indicate a number of cases where the new algorithms outperform the traditional TD algorithm.

Keywords: deterministic finite automata, table-driven algorithms, performance, recognizer denotational semantic, string recognizer, string processor

1 Introduction

Most automata implementers rely on a well-known table-driven (TD) algorithm for string acceptance testing. The algorithm is a simple loop which accesses the transition table and scans through string's symbols in order to establish whether it is part of the language modelled by the DFA or not. To the best of our knowledge, not much has been done to explore alternative methods for implementing DFA-based recognizers that could outperform the conventional TD algorithm in specialised circumstances. Of course, the hardcoded approach suggested by Thompson in [8] has been studied, and experiments revealed that it outperforms TD for DFAs of relatively small size [2].

For each successive element of the input string being scanned, the TD algorithm has to access a row of the transition table. The overall performance of a TD recognizer is determined by the pattern of accesses into the transition table induced by the input string. If, for example, the string induces a fairly random pattern of accesses into the table, then there will be a relatively high probability of cache misses and a consequent degradation of performance [3]. It is thus of interest to investigate alternative table-driven approaches that somehow organize the transition table so as to minimize such effects. It should be noted at the outset that the conclusions to be expected from such an investigation must, of course, be tentative and probabilistically conditioned, since the characteristics of input strings are, in practice, at best only probabilistically known. Nevertheless, as the amount of hardware cache increases, and as finite automata technology is deployed in ever-larger applications, the need to explore effective cache utilization strategies is an important research theme.

In this paper, we propose three implementation strategies associated with the table-driven algorithm in order to minimize the overall latency of a recognizer. In each case, the revised algorithm outperforms the TD algorithm for an appropriate class of input strings. The first strategy, referred to as the dynamic state allocation (DSA) strategy, has already been suggested in [3] and was proven to outperform TD when a large-scale FA is used to recognise very long strings that tend to repeatedly visit the same set of states. The second strategy, referred to as the State pre-ordering (SpO) strategy, relies on a degree of prior knowledge about the order in which states are likely to be visited at runtime. It is shown that the associated algorithm outperforms its TD counterpart no matter the kind of string being processed. The last strategy, referred to as the Allocated Virtual Caching (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.

The remaining part of the paper is organized as follows: in section 2 below, we provide a unified formalism to describe table-driven recognizers that are based on the above mentioned strategies. Section 3 discusses each of the algorithms. Then follows in section 4 discussion on experimental results. The conclusion and further direction to this contribution are given in section 5.

2 Characterization of Table-driven Recognizers

In this section, we describe the various table-driven string recognizers mentioned in the introduction in a formal fashion—specifically, in terms of mathematical functions—where arguments correspond to the strategy according to which the TD recognizer is implemented. The next section gives the pseudo-code for these algorithms.

Consider an automaton M $= (\mathcal{Q}, \mathcal{V}, \delta, s_0, \mathcal{F})$, where: $\mathcal{Q}$ is the set of states; $\mathcal{V}$ is the set of the alphabet symbols; δ is the transition function; s_0 is the start state; and $\mathcal{F}$ is the set of final states of the automaton. We denote by $\mathcal{T} = \mathcal{P}(\mathcal{Q} \times \mathcal{V} \times \mathcal{Q})$ the power set of $\mathcal{Q} \times \mathcal{V} \times \mathcal{Q}$. Since $\mathcal{V}^*$ is the set of all strings over the alphabet, including the empty string; the language of M is denoted $\mathcal{L}(\mathrm{M}) \subseteq \mathcal{V}^*$. We also consider the set $\mathbb{B} = \{T, F\}$ of boolean.

Traditionally, FA-based string recognizers are implemented using the table-driven algorithm. In this case, the transition function is represented in the form of a table (two-dimensional array) whose columns represent the symbols of the alphabet and rows the states of the automaton. The table is therefore an implementation of the function $\delta(q, c_j)$ that is associated with the FA at issue[4].

The traditional TD algorithm does not account for the way in which the transition table is accessed. If the input string results in transitions to states that are arbitrarily located in the table, then there are likely to be many cache misses and consequent performance degradation. On the other hand, if transitions are to states that are contiguously stored in the transition table, then the cache utilisation will be optimal, with consequent performance gains. The following subsection discusses the strategies investigated to date aimed at exploiting this insight.

[4] For convenience, and without loss of generality, it will be assumed that states are integers in the ranges $[-1, |\mathcal{Q}|)$. State -1 corresponds to the sink state that indicates rejection, and need not be represented as a row in the table. Each remaining state, q, corresponds to the qth table row. By convention, 0 corresponds to the start state.

2.1 Dynamic State Allocation (DSA)

Implementation of FA-based string processors that rely on the dynamic state allocation principle requires that a dynamically allocated space be created in memory which is used during acceptance testing. At runtime, as each state is encountered that falls for the first time within the *string path*[5], it is allocated a memory block into which the state's transition information (i.e. a row in the original transition table) is copied. Subsequent references to such a state's transitions are then made via this new piece of memory, rather than via the original transition table. Furthermore, the memory blocks allocated to states on the string path are contiguous, and arranged in the order in which the states are encountered. The DSA strategy was first introduced in [3] and further improvements on the algorithm were suggested in [4].

The TD algorithms based on various strategies are to be described as a mathematical function in subsection 2.4. In this function, we will rely on an argument to represent the DSA strategy. The argument, D is a natural number that indicates the extent to which the strategy has been adopted. This can range from not having been adopted at all, in which case the argument should be 0; to having been adopted for every possible state visited along the state path, in which case the argument should be set to $n = |\mathcal{Q}|$. Two scenarios are distinguished:

- In the *unbounded dynamic state allocation* scenario, the relevant strategy variable is equal to the maximum number of states (i.e. $D = n$). In this case, dynamic allocation occurs as new states that have not yet been dynamically allocated in the new memory space are encountered. In a worst case situation it may be necessary to have the size of the newly allocated memory equal to that of the originally used memory. All that has changed is that the state ordering in the newly allocated memory is organized in a contiguous fashion with respect to the sequence of states in the string's state path.
- In a *bounded dynamic state allocation* scenario, a relevant strategy variable is strictly less than the maximum number of states, but also greater than zero ($0 < D < n$). In this case, the algorithm only has a limited number of states to be allocated dynamically in memory. The restriction means that not all states need necessarily be represented in the new memory location when processing a string whose string path requires more states than those allocated.

Note that a bounded DSA strategy requires a *replacement policy*—i.e. a policy about whether and how to replace states in the dynamically allocated space. In this paper, we shall assume the *direct mapping* replacement policy. In terms of this policy, when the dynamic space is full and reference is made to a state that has not yet been visited, the new state is assigned an address in the dynamically allocated space based on the modulus operation used to identify the state to be removed from the dynamic space. Of course, there could be various other replacement policies such as: the least recently used (LRU) policy whereby, state in allocated memory is removed, replacing it with the least recently invoked state; or associative mapping and the set associative mapping [1, 7]. Alternatively, we may simply chose not to do any replacement at all within the dynamically allocated space. However, these various policy options will not be further explored here.

In the next subsection, a new TD-based implementation strategy referred to as the state pre-ordering strategy is discussed.

[5] String path is construed to mean the set of visited states that are encountered during the processing of the input string.

2.2 State Pre-ordering (SpO)

It may sometimes happen that the percentage of visited states is far below the overall number of states that make up the automaton. Moreover, those frequently visited states may be scattered throughout the transition table. To optimize performance in such a situation, a mechanism is needed to reorganize the transition graph such that frequently accessed states are grouped together in memory, thus reducing the probability of cache misses. The SpO strategy addresses this issue. The strategy requires that a function be run before acceptance testing. The function reorders the automaton's original placement of states in the transition table, to correspond with some ordering of states that is provided by the user as input. Such input could, for example, be based on some *a priori* reasoning of the user, or on an empirical analysis of the history of state visits in prior runs of the FA.

As in the case of the DSA strategy, the reference to this strategy in the mathematical function of subsection 2.4 is in terms of a boolean argument, say P. It indicates whether the SpO strategy is used for implementing the FA or not. Therefore, when the variable evaluates to $true$, a preprocessing operation that reorders the automaton states is to be invoked before acceptance testing. As a result, state i transitions are not necessarily in the ith row of the table. Thus, subsequent accesses to the transition table is done via an auxiliary array, say $p : [0..n)$, whose i^{th} entry is the *new* row number of state i in the transition table. However, if the variable evaluates to $false$, the strategy is not used at all.

The next subsection discusses yet another implementation strategy referred to as allocated virtual caching.

2.3 Allocated Virtual Caching (AVC)

The allocated virtual caching strategy treats a portion of the memory that holds the transition table as a kind of cache area. Typically, this portion of memory is the first V rows of the transition table, where V is some pre-specified value. The phrase *virtual cache* has been coined to differentiate this block of RAM memory from the conventional cache memory in hardware. We algorithmically enforce a type of cache behaviour in utilising this memory. The dedicated portion of the memory is referred to as the *allocated virtual cache*.

During acceptance testing, individual states are transferred into the cache as they are visited, in the hope of enhancing the cache's spatial and temporal locality of reference. The virtual cache will typically be limited in size, and may therefore not contain every single state required. As a result, when reference is made to a state that is not present in the cache, a replacement policy must be used to remove a state from the cache. Removing a state from the cache makes a *cache line* available, so that a new state's information can be placed in the empty cache line.

It is important to realise that the initial cache is regarded as empty, even though the actual cache is occupied by state transition information for the first V table entries. By this we mean that a pointer which keeps track of the portion of cache utilised will initially indicate 0 and progressively climb to V as more and more rows are swopped into cache. Eventually, this pointer will climb to V, whereafter the cache is regarded as full.

The AVC strategy differs from the DSA strategy in the following sense. In the DSA strategy, the state transition information is copied (dynamically) into an free portion of the memory. The AVC strategy, on the other hand, utilises the original

transition table. It swops rows of the initial table, removing some state transition information from the cache into elsewhere in the transition table, and bringing in fresh state transition information into the cache in respective of the most recently encountered state (unless, of course, that most recently encountered state is already in cache, in which case no swopping is required).

Unlike the DSA strategy, the AVC strategy only makes sense if it is bounded. If it were unbounded, then the entire transition table would be regarded as the virtual cache—if a prefix of the string path references all states, then there would never be the need to change the contents of the virtual cache when inspecting the suffix of the string path.

For the present work, for deciding on a state to be removed from cache,we rely on the direct mapping policy that was mentioned in the discussion of the DSA strategy. Of course, we could also chose to have a policy of not swapping out states from the cache once it is full, but instead referencing the transition information from the original state position in the table.

In the mathematical description of the TD algorithm based on an AVC strategy, given in subsection 2.4, V (a natural number) is used as a function argument. If V is 0 then the AVC is not used at all. Alternatively if V has any value in the range $[0, n)$ this means that up to V states may be part of the cache.

The next subsection gives a mathematical function to summarise the above strategies to be built into TD recognizers, relating each implementation strategy to an argument of the function.

2.4 A Unified Formalism of TD algorithms

In general, an acceptor or a string recognizer of a finite automaton is an algorithm that relies on the finite automaton's transition function in order to determine whether a string is part of the language modelled by the FA or not. Therefore, given a input string s and a transition function, δ, the recognizer scans each symbol of the string and returns a boolean. Clearly, this way of describing a recognizer reflects the semantics of the core TD implementation. However, it places no restriction on how the the mapping of s and δ should operate. Since we have introduced three additional strategy arguments for describing the various ways in which a TD recognizer could be implemented, we may now consider a recognizer as a function that requires as arguments a transition function δ, the input string s, and the respective DSA, SpO and AVC strategy arguments, D, P and V. The function then returns a boolean as result. Let us call such a function ρ. It may be characterized as follows:

$$\rho : \mathcal{T} \times \mathbb{N} \times \mathbb{B} \times \mathbb{N} \times \mathcal{V}^* \nrightarrow \mathbb{B}$$

$$\rho(\Delta, D, P, V, s) = \begin{cases} true \text{ if } s \in \mathcal{L}(\mathrm{M}) \\ false \text{ if } s \notin \mathcal{L}(\mathrm{M}) \end{cases}$$

where $0 \leq D \leq |\mathcal{Q}|$, $P \in \mathbb{B}$, and $0 \leq V < |\mathcal{Q}|$.

In fact, ρ may be regarded as the *denotational semantics* of the string recognizer [6]. It specifies the "meaning" of the algorithm in functional terms, but hides details about how the algorithm that performs acceptance testing should actually work. There are, in fact, various ways in which the processing can take place, each corresponding to different instantiations of the strategy arguments. The next section gives algorithms that correspond to various instantiations.

3 The various TD algorithms

The strategy arguments of ρ may be instantiated in $3 \times 2 \times 2 = 12$ different ways: D can either be 0 (no DSA), a bounded value ($D \leq |Q|$), or an unbounded value ($D = |Q|$); P is one of two Boolean values; and V is either 0 (no AVC) or greater than 0. Each of these instantiations may be associated with a different implementation of a table-driven FA-based string recognizer.

Each row of table 1 depicts one of the 12 different algorithms that can be constructed, based on the combination of the values that are assigned to strategy arguments. For easy reference, the last column in the table informs the reader the subsection in the text where the algorithm is discussed. Note that for reasons of space economy, not all algorithms can be fully discussed here. In these cases, the column references a subsection where the algorithm is mentioned. The first column in the table are triplets of the form (D, P, V), indicating instances of the strategy variables that relate to this specific algorithm. The second column informs the reader of the strategies that are implemented in the construction of the algorithm.

The name given to each algorithm starts with the letter t followed by the concatenation of the numbers assigned to each active strategy as follows: the number 1 is assigned to the DSA strategy; the number 2 is assigned to the SpO strategy; and the number 3 is assigned to the AVC strategy. Since there are two variations to the DSA strategy, we chose to prefix its number with: u for the unbounded case, and b for the bounded case. For example, t_{u1} refers to the table-driven algorithm based on the unbounded DSA strategy, t_{b123} refers to the table-driven algorithm based on the bounded DSA strategy combined with the SpO and AVC strategies; and of course, t refers to the core table-driven algorithm.

Combination	Active strategy	Name	Reference
$(0, F, 0)$	None	t	3.1
$(d, F, 0)$	bounded DSA	t_{b1}	3.2
$(n, F, 0)$	unbounded DSA	t_{u1}	3.2*
$(0, T, 0)$	SpO	t_2	3.3
$(0, F, v)$	AVC	t_3	3.4
$(0, T, v)$	SpO and AVC	t_{23}	3.5
$(d, T, 0)$	bounded DSA and SpO	t_{b12}	3.5*
(d, T, v)	bounded DSA, SpO and AVC	t_{b123}	3.7
(d, F, v)	bounded DSA and AVC	t_{b13}	3.6
$(n, T, 0)$	unbounded DSA and SpO	t_{u12}	3.5*
(n, T, v)	unbounded DSA, SpO and AVC	t_{u123}	3.7*
(n, F, v)	unbounded DSA and AVC	t_{u13}	3.6*

Table 1. The Range of TD-based algorithms (those with a * are not discussed in details)

These various algorithms are now discussed in the subsections below.

3.1 The core TD algorithm

This well-known algorithm is rather simple: it takes as input the transition function δ, the string s to be processed, and returns a boolean.

Algorithm 3.1(The core table-driven recognizer)

proc $t(\delta, s)$
  $; q, j := 0, 0$
  do $(j < s.len) \wedge (q \geq 0) \rightarrow$
    $q,\ j := \delta(q, s_j),\ j + 1$
  od
  if $q < 0 \rightarrow$ {return false} $[\!]$ $q \geq 0 \rightarrow$ {return true} **fi**

3.2 The bounded TD-DSA algorithm

The algorithm is based on the DSA strategy and accounts for both bounded and unbounded case according to the nature of the strategy argument D. The unbounded case of the algorithm was discussed in [3] and here, we provide its bounded counterpart, which involves state replacement when the dynamically allocated space is full, and reference is made to a state that has not previously been visited.

In addition to the core TD algorithm inputs, TD-DSA also requires as input the number of states, D, that may be dynamically allocated; the block size, Z, of memory required for each newly allocated state; and the starting address, A from which memory is to be allocated. In the algorithm, The variable p serves as a counter of the number of states that have been dynamically allocated to date. If $p \geq 0$ is the qth entry in array $m_{[0,n)}$ (which is initialized to -1) then this means that the qth row of δ has been copied over to the pth row of the table d that has been dynamically evolved in memory. The function $search(m, r)$ returns the index i in the array $m_{[0..n)}$ for which $m_i = r$.

When $m_q = -1$, where q is the current state, then a further test is made on p to find out whether the reserved dynamic portion of memory is full or not. If not full, the block referenced by variable d is expanded by Z bytes (starting from address B which initially is A), the state information is copied into d, and B is incremented to points to the next address where the next state information may be copied if the dynamic memory block is not full.

Algorithm 3.2(The bounded TD-DSA recognizer)

proc $t_{b1}(\delta, D, A, Z, s)$
  $; m_{[0..n)}, B, q, j, p := -1, A, 0, 0, 0$
  ;**do** $(j < s.len \wedge q \geq 0) \rightarrow$
    if $(m_q = -1) \rightarrow$ { state not dynamically allocated}
      if $(p < D) \rightarrow$
        $; m_q, d_p := p, malloc(B, Z)$
        $; d_{p,[0..a)}, p, B := \delta_{q,[0...a)}, p + 1, B + Z$
        $; q := d_{p,s_j}$
      $[\!]$ $(p \geq D) \rightarrow$
        $r := MOD(q, D)$\{ remainder in the division of q by D\}
        $; m_q := r$
        $; i := search(m, r)$
        $; m_i := -1$
        $; d_{r,[0..a)}, q := \delta_{q,[0..a)}, d_{r,s_j}$
      fi

```
        ⫿  m_q ≠ −1 → skip{ state dynamically allocated}
        fi
        ; q, j := d_{m_q, s_j}, j + 1
    od
    if q < 0 → {return false} ⫿ q ≥ 0 → {return true} fi
```

3.3 The TD-SpO algorithm

Again, in addition to the input for the TD algorithm, the TD-SpO algorithm is provided with an auxiliary array $p_{[0..n)}$ such that p_i specifies to which new row of δ the ith row of δ should be moved. At the start, the function $reorder(\delta, p)$ reorders the automaton's states according to p's entries. Then follows proper acceptance testing whereby, access to a state q information is made indirectly via p. Thus, $\delta(p_q, s_j)$ returns the transition triggered by the string's symbol s_j at state q.

Algorithm 3.3(The TD-SpO recognizer)

```
proc t_2(δ, p, s)
    ; reorder(δ, p)
    ; q, j := 0, 0
    do (j < s.len) ∧ (q ≥ 0) →
        q, j := δ(p_q, s_j), j + 1
    od
    if q < 0 → {return false} ⫿ q ≥ 0 → {return true} fi
```

3.4 The TD-AVC algorithm

For this algorithm, the size of the virtual cache, V, has to be provided. Auxiliary arrays $m_{[0..n)}$, $c_{[0..V)}$ and $i_{[0..n)}$ are used. $m_q = r$ means that state q transition information is held in row r of the table δ. The array $c_{[0..V)}$ is used to hold the states currently in the cache, and the array $i_{[0..n)}$ indicates whether a state is currently in the cache ($i_q = 0$) or not ($i_q = -1$). All entries of i are set to -1, indicating that states have not yet been visited although the first V states are initially in the cache by default. The variable l is used as cache line controller and helps establish whether the cache is full or not. The cache is said to be full when the number of different states visited thus far has reached V.

For every iteration of the main loop, a test is made to check whether the current state, q is in the cache or not. If it is ($i_q \neq -1$), then the normal transition code is executed. Otherwise a check is made to see whether the cache is full or not. If the cache is full ($l \geq V$) and q is not in the cache, then further acceptance testing may take place only *after* doing state replacement, as for the DSA strategy. Otherwise, (i.e. the cache is not full and q is not in cache) q's transition information is swapped with that of state l, then l is incremented and acceptance testing takes place. Of course, information of those states that are in the cache by default are not swopped provided that their state matches the state in the cache line (ie. $q = c_l$).

State information is swopped by the function $swd(\delta[m_q], \delta[m_p])$ that also interchanges the entry m_q of the current state q with the entry m_p of the state p currently in the cache line, and referenced in the algorithm by c_l.

In order to do state replacement when the cache is full, we use the modulo operation to determine the position in the cache to which q should be moved. Then follows implicit interchange of state information as previously described. Of course, whether the cache is full or not, a states's information in i is updated accordingly as depicted in the algorithm.

Algorithm 3.4(Table-driven based on allocated virtual caching)

```
proc  t_3(δ, V, s)
      ; q, j, p, l := 0, 0, 0, 0
      ; m_[0..n), c_[0..V), i_[0..n) := [0..n), [0..V), −1
      do  (j < s.len) ∧ (q ≥ 0) →
          if  (i_q ≠ −1) → skip
          ▯  (i_q = −1) ∧ (l < V) →
              if  q = c_l → skip
              ▯  q ≠ c_l →
                  p := c_l
                  ; swd(δ[m_q], δ[m_p])
                  ; i_p, c_l := −1, q
              fi
              ; i_q, l := 0, l + 1
          ▯  (i_q = −1) ∧ (l ≥ V) →
              p := MOD(m_q, V)
              swd(δ[m_q], δ[m_{c_p}])
              i_q, i_{c_p}, c_p := 0, −1, q
          fi
          ; q, j := δ(m_q, s_j), j + 1
      od
      if  q < 0 → {return false} ▯ q ≥ 0 → {return true}  fi
```

3.5 The TD-SpO-AVC algorithm

The algorithm relies on both SpO and AVC strategies. A naïve approach for its implementation would be to first reorder the automaton's states using the function $reorder(\delta, p)$, and then invoke (for every iteration of the main loop) a function $tdavc(\delta, p, m, c, i, l, V, j, q, s)$ that updates the next state q to be transited to, as well as the next index j of the string s currently being processed. This latter function also takes as parameters the arrays m, c, and i as well as the cache line controller, l, previously described in subsection 3.4. Moreover, access to states' original information is made via entries of the array p. The algorithm below depicts the pseudo-code of algorithm t_{23}.

Algorithm 3.5(The TD-SpO-AVC algorithm)

```
proc  t_23(δ, p, V, s)
      ; reorder(δ, p)
      ; q, j, p, l := 0, 0, 0, 0
      ; m_[0..n), c_[0..V), i_[0..n) := [0..n), [0..V), −1
      do  (q < s.len) ∧ (q ≥ 0) →
```

$tdavc(\delta, p, m, c, i, l, V, j, q, s)$
od
if $q < 0 \rightarrow$ {return false} $[\!]$ $q \geq 0 \rightarrow$ {return true} **fi**

The same principle applies for algorithms t_{u13} and t_{b13}, that also rely on the combination of the SpO strategy with the unbounded and bounded DSA strategies. All that is required is to change the function in the main loop with a function that implements the relevant strategy and updates the next state and the next string index.

3.6 The bounded and unbounded TD-DSA-AVC algorithm

These two algorithms combine the AVC strategy with either the bounded DSA strategy or the unbounded DSA strategy. We only discuss here the unbounded TD-DSA-AVC algorithm. For its implementation, the following simple policy may be adopted for an automaton of n states:

- The first k states of the automaton are cacheables. That is, they are processed within the virtual cache which holds up to V states, and the following holds: $0 < V < k < n$
- The remaining $n - k$ states are processed within the allocated dynamic memory space where each state occupies Z bytes, and the first state to be allocated dynamically is located at address A in the memory.

In the algorithm, for every iteration of the main loop, upon accessing a state q, a test is first made to determine whether the state is cacheable or not. If that is the case, the function $tdavc(\delta, m, c, i, l, V, j, q, s)$ is invoked that updates the next state q, the next symbol s_j to be processed, as well as all the other parameters involved in the function. However, if the state is not cacheable, it ought to be processed within the dynamically allocated space using the function $utddsa(\delta, A, Z, B, q, j, s)$ that updates the variables q and j, as well as the parameter B that holds the next address to be used for space allocation in the case the state currently being processed has not yet been visited. The pseudo-code of the algorithm is depicted below.

Algorithm 3.6(The unbounded TD-DSA-AVC algorithm)

---

proc $t_{u13}(\delta, k, A, Z, s)$
$\quad ; q, j, p, l, B := 0, 0, 0, 0, A$
$\quad ; m[0..n), c[0..V), i[0..n) := [0..n), [0..V), -1$
$\quad$;**do** $(j < s.len \wedge q \geq 0) \rightarrow$
$\quad\quad$ **if** $q < k \rightarrow tdavc(\delta, m, c, i, l, V, j, q, s)$
$\quad\quad$ $[\!]$ $q \geq k \rightarrow utddsa(\delta, A, Z, B, q, j, s)$
$\quad\quad$ **fi**
$\quad$ **od**
$\quad$ **if** $q < 0 \rightarrow$ {return false} $[\!]$ $q \geq 0 \rightarrow$ {return true} **fi**

3.7 The bounded and unbounded TD-DSA-SpO-AVC algorithm

The algorithms consist of the SpO strategy and a combination of the AVC strategy and either of the DSA strategies. For consistency, we briefly discuss the unbounded case. Based on previous discussions, the algorithm would first consists of a preprocessing phase whereby the function $reorder(\delta, p)$ is invoked. Then follows the processing phase similar to the unbounded TD-DSA-AVC algorithm described in the previous subsection. Of course the policy on the number of states to be cacheable and those to be dynamically allocated must be defined. We adopt the same principle as previously described, and the algorithm is depicted below.

Algorithm 3.7(The unbounded TD-DSA-SpO-AVC algorithm)

---

proc $t_{u123}(\delta, p, s, c, k, d, A, Z)$
  $; reorder(\delta, p)$
  $\{\ Initializations\ \}$
  ;**do** $(q < s.len \wedge q \geq 0) \rightarrow$
    if $q < k \rightarrow tdavc(\delta, p, m, c, i, l, V, j, q, s)$
    $[\!]$ $q \geq k \rightarrow utddsa(\delta, p, A, Z, B, q, j, s)$
    fi
  od
  if $q < 0 \rightarrow$ {return false} $[\!]$ $q \geq 0 \rightarrow$ {return true} **fi**

The reader should notice the presence of the auxiliary array p as argument in the functions in the main loop. This simply emphasizes that, unlike algorithm t_{u13}, access to original states' information is made via entries of p.

4 Experimental Results

Various experiments were conducted in order to determine the point at which the derived implementation strategies outperform the core TD implementation. We relied on artificially generated data, in each case contrived to generate an input string that would demonstrate the strength of a relevant new algorithms in relation to the core TD algorithm.

The algorithms were implemented using the Netwide Assembly language (NASM), under the Linux OS, on an Intel Pentium IV machine. For each algorithm under investigation, 120 different automata were generated of size ranging from 100 to 12000 states, with increment of 100. 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 first n symbols where randomly chosen among the automaton states, and those symbols where repeated 4 times.

For each algorithm involving the SpO strategy, the array of state positions $p_{[0..n)}$ was randomly generated.

Each recognizer was then run 50 times, and the minimum time in clock cycles (ccs) was recorded. In order to evaluate the extent to which the algorithms outperformed the core TD algorithm and vice-versa, the data collected for each algorithm was plotted against the data of the core TD algorithm.

For the bounded TD-DSA and the TD-AVC algorithms, we chose a bound of 50% on 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, respectively.

The graphs in figure 1 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 strategy was already discussed in [3]. As shown in the graphs, the TD-algorithm outperforms the bounded TD-DSA algorithm under the conditions discussed above. This suggests that the bounded nature of the algorithm requires frequent state replacement during acceptance testing when the dynamically allocated space is full. Therefore, the following scenarios merit further experimentation in respect 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 guarantee better cache placement and data organization since it may happen that a state is constantly swapped in and out of the cache, and hence poor performance of the algorithm. A policy such as the associative mapping or the LRU policy could perhaps be used to avoid such problem [7]. Alternatively, once the threshold for state allocation has been reached, we could avoid using any replacement policy. Instead, acceptance could be performed through the transition table whenever reference is made to a state out of the dynamic space.
– *Kind of string*: Previous experiments indicated that up to 70% of the automaton's states were accessed during acceptance testing [3], when processing the kind of randomly generated accepting strings that we used for the current experiment. Therefore, dedicating only 50% of those states to the dynamic memory space apparently does not guarantee sufficient improvement on data organization. Instead, for the particular strings generated, excessive overheads are incurred, with consequent poor performance.

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 cache was full. This approach resulted in TD-AVC competing with it core TD counterpart as shown in figure 1-IV. Again, since up to 70% of the automaton's state are visited, we merely have 20% of the states that are processed out of cache whiles the remaining are processed in cache. This observation clearly shows that the strategy is competitive under appropriate circumstances. However, the range of data that will provide better cache utilization requires further investigation. Furthermore, even better performance of TD-AVC over the TD is expected if the cache size increases at about 70% of the total number of states, since most of the states would be contiguously organized. The graphs in figure 1-III also reveal that the TD-SpO strategy outperforms its TD counterpart. It should be noted that we did not take into account the time taken to reorder the states, since this is regarded as a once-off pre-processing activity. The results confirm that pre-ordering of the state rows can indeed bring about performance improvements. This will be at an optimal when pre-ordering maximizes spacial and temporal locality of reference.

Figure 2 depicts the graphs of the various algorithms obtained by combining the implementation strategies. As may be observed, the combination of the SpO strategy with the unbounded TD-DSA yields better performance over the core TD algorithm. This also applies to algorithms t_{23} (this version used direct mapping for replacement). Again, the pre-ordering of states apparently results in better data organization, and hence better cache utilization. However, for algorithm t_{u123}, the AVC strategy re-

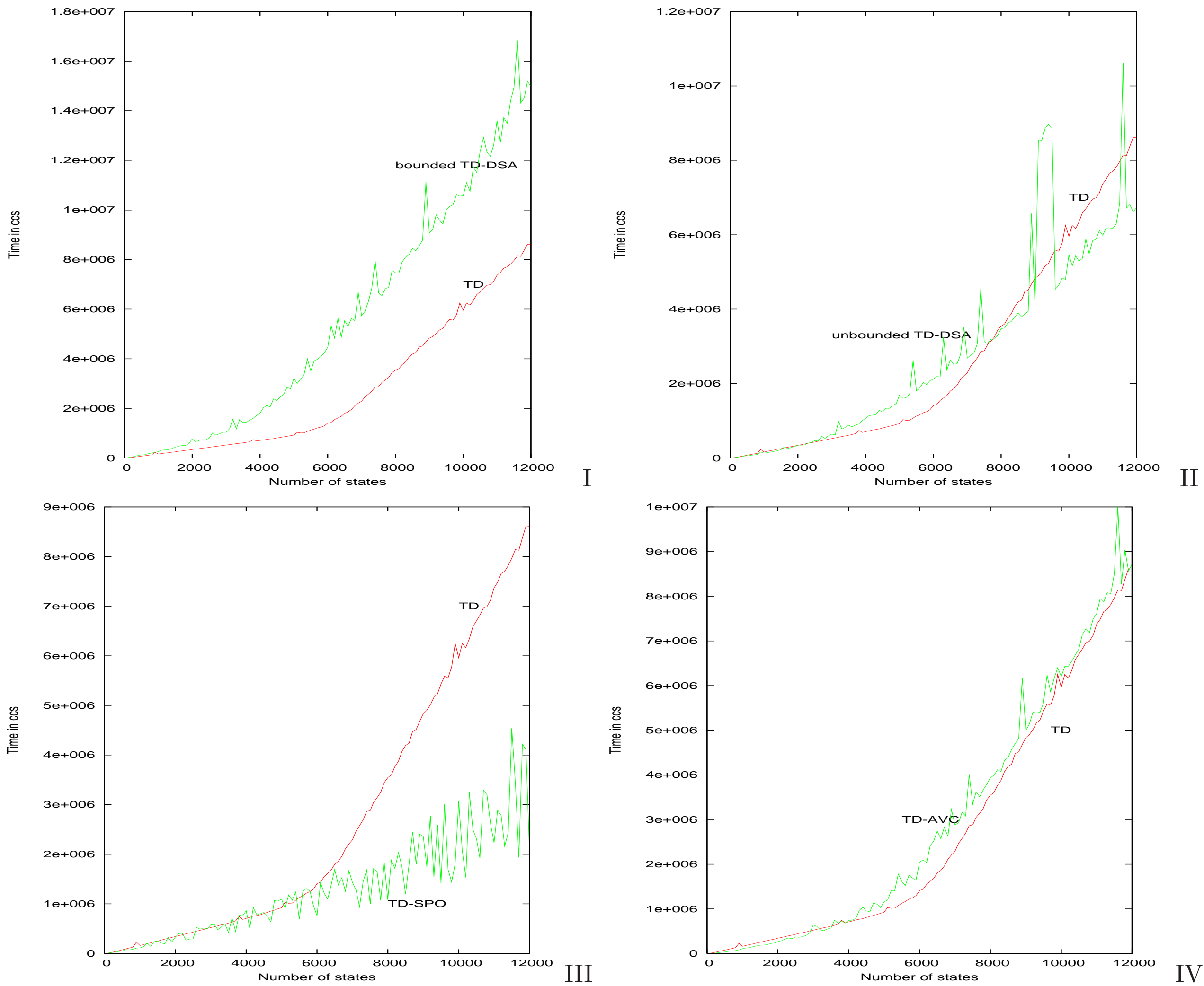

Figure 1. Performances of TD vs DSA (I & II), AVC (III), and SPO (IV)

lied on direct mapping for replacement; this resulted in overheads and therefore poor cache utilization, and hence poor performance. It is expected that by choosing suitable strings, a better performance would be observed. For algorithm t_{b123}, the bounded nature of both DSA and AVC strategies required replacement, resulting in various overheads; this explains its poor performance over the TD algorithm. Again, suitable data set would produce better result. 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 conclusion and further direction to this contribution are depicted in the next section.

5 Conclusion and Future Work

In this paper, we have investigated various ways of improving performances of the conventional TD algorithm using various implementation strategies to which were associated parameter arguments. A 6-argument function provided the denotational semantics of various TD FA-based string recognizers. Alternative instantiations of these arguments represented new TD-based algorithms. The algorithms were then

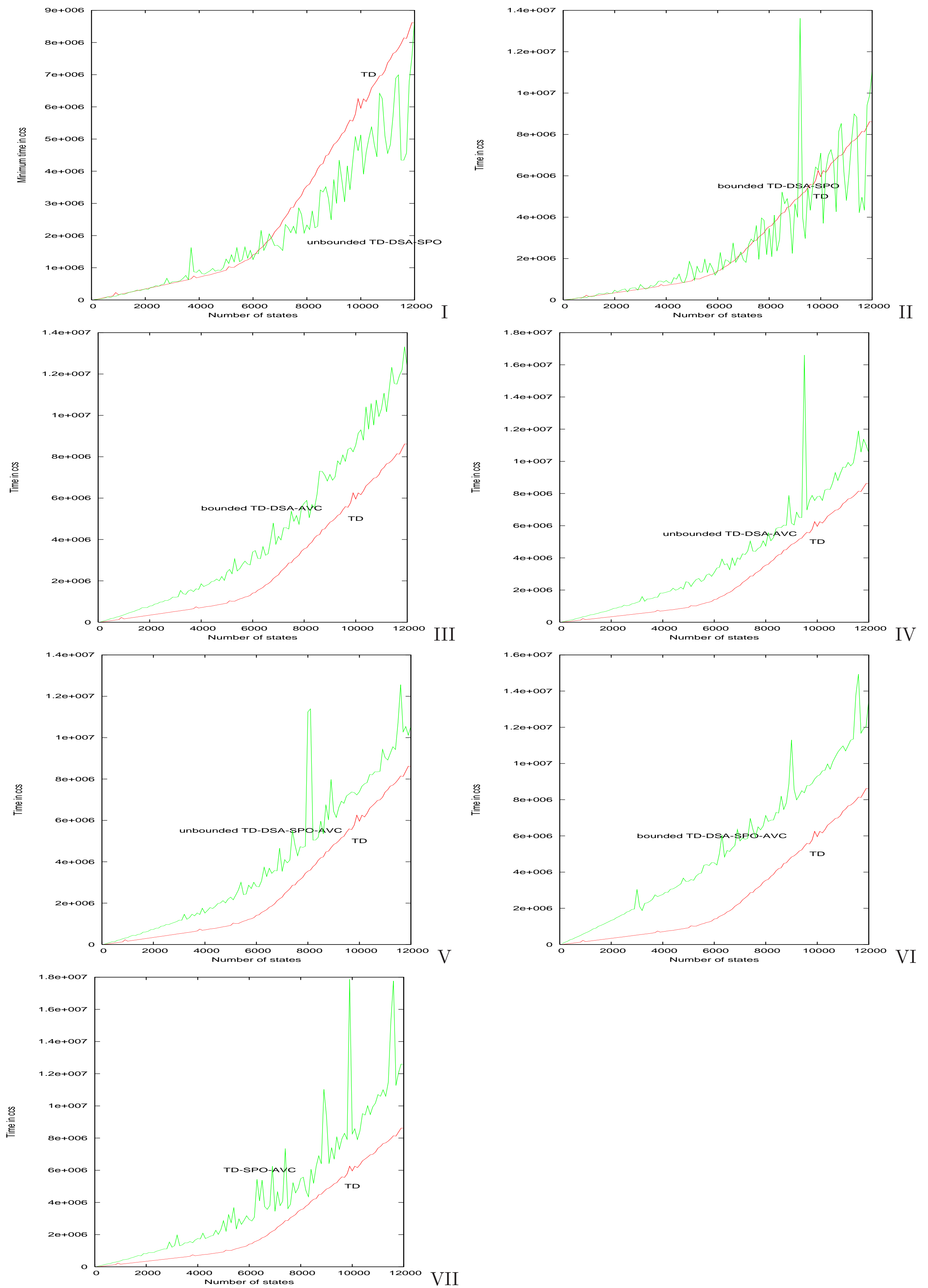

Figure 2. Performances of TD vs DSA-SPO (I & II), DSA-AVC (III & IV), DSA-SPO-AVC (V & VI), and SPO-AVC (VII)

implemented and performance recorded. It was shown that, based on the strings made of long repeated sequences, some of the algorithms outperformed the traditional TD algorithm, but the others were not of interest due to the appropriateness of the kind of string considered and the policy used for replacement.

The algorithms were tested by relying on artificially generated data (strings and automata). Thus as a matter of future work, various experiments will be conducted on real life data such as genetic sequences, micro-satellites for tandem repeat detection, network intrusion detection, and the like. We also wish to further explore the string characteristics and appropriate sizes for dynamically allocated space (in the case of DSA) and virtual cache (in the case of AVC) respectively.

The algorithms presented in this work are part of a toolkit that is under construction for FA-based string processing, targeted at applications that use FA-based string recognition as part of their model solution.

References

[1] J. P. Hayes: *Computer Architecture and Organization*, McGraw-Hill, third ed., 1998.

[2] E. N. Ketcha: *Hardcoding finite automata*, Master's thesis, University of Pretoria, Department of computer Science, Pretoria 0002, South Africa, November 2003.

[3] E. N. Ketcha, D. G. Kourie, and B. W. Watson: *Reordering finite automatata states for fast string recognition*, in In Proceeding of the Prague Stringology Conference, Prague, Czech Republic, August 2005, Czech Technical University.

[4] E. N. Ketcha, D. G. Kourie, and B. W. Watson: *Dynamic allocation of finite automatata states for fast string recognition*. International Journal of Foundation of Computer science, 2006.

[5] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings*. SIAM Journal on Computing, 6(3) 1977, pp. 368–387.

[6] B. Meyer: *Introduction to the Theory of Programming Languages*, Prentice Hall, c.a.r hoare series ed., 1990.

[7] D. A. Patterson and J. L. Hennessy: *Computer Organization and Design*, Morgan Kaufmann, third ed., 2005.

[8] K. Thompson: *Regular expression search algorithm*. Communications of the ACM, 11(6) 1968, pp. 323–350.

[9] A. C. Yao: *The complexity of pattern matching for a random string*. SIAM Journal on Computing, 8(3) 1979, pp. 368–387.