PSC'06

On Implementation and Performance of Table-driven DFA-based String Processors

E. Ketcha Ngassam
Derrick G. Kourie
Bruce W. Watson
FASTAR Research Group
www.fastar.org

Agenda

1. Introduction
2. Implementation Strategies
3. Characterization of TD Recognizers
4. The various Table-driven Algorithms
5. Experimental Results
6. Conclusion and Future Work

Introduction

DFA-based String Recognition: The problem

Given an automaton $M(\mathcal{Q}, \mathcal{V}, \delta, \mathcal{F}, s_0)$*, where:*
$\mathcal{Q}$ is the set of states;
$\mathcal{V}$ the alphabet;
δ the transition function;
$\mathcal{F} \subseteq \mathcal{Q}$ the set of final states;
s_0 the start state;
Such that $\mathcal{L}(M)$ is the language modelled by M with $\mathcal{L}(M) \subseteq \mathcal{V}^*$;
And s, an input string ($s \in \mathcal{V}^*$)
Check whether $s \in \mathcal{L}(M)$

Introduction

Implementation:

Hardcoding:
Transition table embedded in the algorithm

Useful for automata of relatively small size (in the order of hundreds)

Introduction

Implementation:

Table-driven:

Rows represent states and columns alphabet symbols

Performance determined by the pattern of accesses in the table

Efficient if rows are accessed contiguously

High probability of cache misses if random pattern of accesses

Introduction

- **Example of TD DFA-based Recognizer:**

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

The string $s_1 = aaaa$ is processed faster than the string $s_2 = bcdc$
For s_1 rows are access contiguously, hence high probability of cache hits
High probability of cache misses for s_2

- **Problem: explore various strategies aimed at lowering probability of cache misses during acceptance testing**

Implementation Strategies

- **Dynamic State Allocation (DSA)**

A dynamically allocated state is created in memory for acceptance testing

Copy into a block of memory first encountered states

Subsequent reference to such state is made via the new piece of memory

- **Example: accepting the string** $bcdc$

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

- **Will result in:**

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

Implementation Strategies

- **Dynamic State Allocation (DSA)**
  - More details are provided in PSC'05
  - Relies on the use of a natural number D
    - Determines the extent of the dynamic allocation of memory blocks

$D = 0 \equiv$ no DSA strategy
$D = |\mathcal{Q}| \equiv$ the strategy is **unbounded**
$D < |\mathcal{Q}| \equiv$ the strategy is **bounded**
May require replacement policy when threshold is reached
· Direct mapping policy
· LRU policy
· Associative mapping
· No policy

Implementation Strategies

- **State pre-Ordering (SpO)**

Very low percentage of visited states compared to the overall DFA size
Frequently visited states scattered throughout the transition table

- ○ Reorganize δ such that frequently accessed states are grouped together
- ● Reduces the probability of cache misses
- ○ Reorders the DFA's original placement of states in the transition table

Relies on user's input:

- ○ A priori reasoning
- ○ Empirical analysis of the history of states visits in prior runs

Implementation Strategies

- **State pre-Ordering (SpO)**

A boolean argument P is passed to the function such that:
$P = \mathsf{F} \equiv$ SpO strategy not used
$P = \mathsf{T} \equiv$ preprocessing operation required for reordering
- ○ Access to state i is done via an array $p_{[0..n]}$
- ○ The i^{th} entry is the new row of i in the transition table

- Example: $P = \mathsf{T}$, $p_{[0..3]} = \{0, 3, 2, 1\}$

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

- Before acceptance testing, the table is reordered into:

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

Implementation Strategies

- **Allocated Virtual Caching (AVC)**

Treats a portion of the memory that holds the transition table as a ***cache***

The first V rows of the table acts as a virtual cache
- Virtually acts as the hardware cache memory (Replacement policies)

Hope to enhance cache's spatial and temporal locality of reference
- Individual states are transferred into the cache as they are visited
- Virtual cache limited in size
- Reference to a state out of the cache when it is full requires replacement
- The cache initially is regarded as empty
- Use of a pointer to manage cache utilization

Implementation Strategies

- **Allocated Virtual Caching (AVC)**

Example:
$V = 1$ (cache size);
$c = \{0, 1\}$(states initially in cache)
$l = 0$ (pointer);
$i = \{\mathsf{F}, \mathsf{F}, \mathsf{F}, \mathsf{F}\}$ (cache indicator)
$q = 0$ (start state);
$m = \{0, 1, 2, 3\}$ (current states' position)
$s = \mathbf{b}cdc$ (string to be tested)

The table initially:

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

Implementation Strategies

- **Allocated Virtual Caching (AVC)**

We encounter the symbol b
$l = 0 \leq V$
$q = 0$
$c_l = q = 0$

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

Acceptance testing:

$l = l + 1 = 1$

$c = \{0, 1\}$

$i = \{\mathsf{T}, \mathsf{F}, \mathsf{F}, \mathsf{F}\}$

$m = \{0, 1, 2, 3\}$

$q = \delta(q, b) = 3$

Implementation Strategies

- **Allocated Virtual Caching (AVC)**

We encounter the symbol c
$l = 1 \leq V$
$q = 3$
$c_l = 1 \neq q = 3$

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

Acceptance testing:

$l = l + 1 = 2 > V$ (the cache is full)

$c = \{0, 3\}$

$i = \{\mathsf{T}, \mathsf{F}, \mathsf{F}, \mathsf{T}\}$

$m = \{0, 3, 2, 1\}$

$q = \delta(m_q, c) = 2$

Implementation Strategies

- **Allocated Virtual Caching (AVC)**

We encounter the symbol d
$l = 2 > V$
$q = 2$
$i_q = \mathsf{F}$
The cache is full:
  - Replacement policy to remove a state from the cache;
  - Do acceptance testing without any further replacement

The other iterations are easy to follow

Characterization of TD Recognizers

- **Definition:**

A string recognizer of a DFA is an algorithm that relies on the DFA's transition function to determine whether a string is part of the language modelled by the DFA or not.

Characterization of TD Recognizers

- **Definition:**

Given an input string s*, an automaton* $M(\mathcal{Q}, \mathcal{V}, \delta, \mathcal{F}, s_0)$,
the recognizer scans each symbol of s and returns a boolean $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$

The definition is somewhat general
No restriction on how the recognizer would be implemented
◦ Need to introduce the strategies previously described

We consider a recognizer as a function ρ that accepts as arguments s, δ and the respective strategy variables, and returns a boolean:
◦ ρ is the *denotational semantics* of the recognizer

Characterization of TD Recognizers

- **Formalism:**

Specification of ρ in functional terms
Assume $\mathcal{T} = \mathcal{Q} \times \mathcal{V}$ (transition relation)

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

$$\rho(\delta, D, P, V, s) = \begin{cases} \mathsf{T} & \text{if } s \in \mathcal{L}(\mathrm{M}) \\ \mathsf{F} & \text{if } s \notin \mathcal{L}(\mathrm{M}) \end{cases}$$

where:
$0 \leq D \leq |\mathcal{Q}|$ (DSA strategy);
$P \in \mathbb{B}$ (SpO strategy);
$0 \leq V < |\mathcal{Q}|$ (AVC strategy).

The various TD Algorithms

- **Various instances:**

DSA Strategy: The variable $D \in \{0, d, |Q|\}$; $d < |Q|$
SpO Strategy: The variable $P \in \{\mathsf{T}, \mathsf{F}\}$
AVC Strategy: The variable $V \in \{0, v\}$; $v < |Q|$

Resulting in $3 \times 2 \times 2 = 12$ different algorithms

The algorithms:

Combination	Active strategy	Name
$(0, \mathsf{F}, 0)$	None (core TD)	t
$(d, \mathsf{F}, 0)$	bounded DSA	t_{b1}
$(n, \mathsf{F}, 0)$	unbounded DSA	t_{u1}
$(0, \mathsf{T}, 0)$	SpO	t_2
$(0, \mathsf{F}, v)$	AVC	t_3
$(0, \mathsf{T}, v)$	SpO and AVC	t_{23}
$(d, \mathsf{T}, 0)$	bounded DSA and SpO	t_{b12}
(d, T, v)	bounded DSA, SpO and AVC	t_{b123}
(d, F, v)	bounded DSA and AVC	t_{b13}
$(n, \mathsf{T}, 0)$	unbounded DSA and SpO	t_{u12}
(n, T, v)	unbounded DSA, SpO and AVC	t_{u123}
(n, F, v)	unbounded DSA and AVC	t_{u13}

The various TD Algorithms

- **The core TD algorithm**

Takes as input the transition function δ, the string s and returns a boolean

Algorithm 0.1(The core table-driven recognizer)

$$
\begin{array}{l}
\mathbf{proc}\ t(\delta, s) \\
;q, j := 0, 0 \\
\mathbf{do}\ (j < s.len) \wedge (q \geq 0) \rightarrow \\
\qquad q,\ j := \delta(q, s_j),\ j+1 \\
\mathbf{od} \\
\mathbf{if}\ q < 0 \rightarrow \{\text{return } \mathsf{F}\} \ \|\ q \geq 0 \rightarrow \{\text{return } \mathsf{T}\}\ \mathbf{fi}
\end{array}
$$

The various TD Algorithms

- **The bounded TD-DSA Algorithm (t_{b1})**

The strategy argument $D < |Q|$
◦ Unbounded counterpart was discussed in PSC'05
May involve state replacement
when reference is made to a state that has not yet been visited provided the allocated dynamic space is full

The various TD Algorithms

- The bounded TD-DSA Algorithm (t_{b1})

Algorithm 0.2(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
  $[\!]\ m_q \neq -1 \rightarrow$ skip{ state dynamically allocated}
  fi
  $;\, q, j := d_{m_q,s_j}, j+1$
od
if $q < 0 \rightarrow$ {return F} $[\!]$ $q \geq 0 \rightarrow$ {return T} **fi**

UNISA university of south africa

The various TD Algorithms

- **The TD-SpO algorithm**

Additional input: auxiliary array $p_{[0..n]}$
p_i is the new row of δ where i^{th} row of δ should be moved

$reorder(\delta, p)$ recorders the DFA's states according to p's entries
Access to a state q is made indirectly via p:
i.e. $q = \delta(p_p, s_j)$

The various TD Algorithms

- **The TD-SpO algorithm**

Algorithm 0.3(The TD-SpO recognizer)

$$
\begin{array}{l}
\textbf{proc } t_2(\delta, p, s) \\
; reorder(\delta, p) \\
; q, j := 0, 0 \\
\textbf{do } (j < s.len) \wedge (q \geq 0) \rightarrow \\
\qquad q, j := \delta(p_q, s_j), j + 1 \\
\textbf{od} \\
\textbf{if } q < 0 \rightarrow \{\text{return } \mathsf{F}\} \parallel q \geq 0 \rightarrow \{\text{return } \mathsf{T}\} \textbf{ fi}
\end{array}
$$

The various TD Algorithms

- **The TD-AVC algorithm**

V: the size of the virtual cache
$m_{[0..n]}$: holds current state position in δ
$c_{[0..V]}$: holds states currently in the virtual cache
$i_{[0..n]}$: holds information indicting whether a state is in cache or not
l: cache line controller. $l = 0 \Rightarrow$ cache is empty; $l \geq V \Rightarrow$ cache is full
if $l \leq V$
  Acceptance testing if the current state matches the current cache line
  Otherwise, invoke $swd(\delta[m_q], \delta[m_{c_l}])$ before acceptance testing
if $l > V$
  Use replacement policy before acceptance testing

The various TD Algorithms

- **The TD-AVC algorithm**

Algorithm 0.4(Table-driven based on allocated virtual caching)

$$
\begin{array}{l}
\textbf{proc } t_3(\delta, V, s) \\
; q, j, p, l := 0, 0, 0, 0 \\
; m_{[0..n)}, c_{[0..V)}, i_{[0..n)} := [0..n), [0..V), -1 \\
\textbf{do } (j < s.len) \wedge (q \geq 0) \rightarrow \\
\quad \textbf{if } (i_q \neq -1) \rightarrow \text{skip} \\
\quad [\!] \; (i_q = -1) \wedge (l < V) \rightarrow \\
\qquad \textbf{if } q = c_l \rightarrow \text{skip} \\
\qquad [\!] \; q \neq c_l \rightarrow \\
\qquad\quad p := c_l \\
\qquad\quad ; swd(\delta[m_q], \delta[m_p]) \\
\qquad\quad ; i_p, c_l := -1, q \\
\qquad \textbf{fi} \\
\qquad ; i_q, l := 0, l + 1 \\
\quad [\!] \; (i_q = -1) \wedge (l \geq V) \rightarrow \\
\qquad p := MOD(m_q, V) \\
\qquad swd(\delta[m_q], \delta[m_{c_p}]) \\
\qquad i_q, i_{c_p}, c_p := 0, -1, q \\
\quad \textbf{fi} \\
\quad ; q, j := \delta(m_q, s_j), j + 1 \\
\textbf{od} \\
\textbf{if } q < 0 \rightarrow \{\text{return } \mathsf{F}\} \; [\!] \; q \geq 0 \rightarrow \{\text{return } \mathsf{T}\} \textbf{ fi}
\end{array}
$$

The various TD Algorithms

- **The TD-SpO-AVC algorithm**

Relies on both SpO and AVC

A naïve approach:

- Use $reorder(\delta, p)$ to reorder the automaton's states (SpO)
- Perform acceptance testing using AVC approach

The various TD Algorithms

- **The TD-SpO-AVC algorithm**

Algorithm 0.5(The TD-SpO-AVC algorithm)

$$
\begin{array}{l}
\textbf{proc } t_{23}(\delta, p, V, s) \\
;reorder(\delta, p) \\
;q, j, p, l := 0, 0, 0, 0 \\
;m_{[0..n)}, c_{[0..V)}, i_{[0..n)} := [0..n), [0..V), -1 \\
\textbf{do } (q < s.len) \wedge (q \geq 0) \rightarrow \\
\qquad tdavc(\delta, p, m, c, i, l, V, j, q, s) \\
\textbf{od} \\
\textbf{if } q < 0 \rightarrow \{\text{return } \mathsf{F}\} \parallel q \geq 0 \rightarrow \{\text{return } \mathsf{T}\} \textbf{ fi}
\end{array}
$$

The various TD Algorithms

- **The bounded and unbounded TD-DSA-AVC algorithm**

Combination of the AVC with either the bounded/unbounded DSA

A simple policy:

- The first k states of the DFA are cacheable with $0 < V < k < n$
- The remaining $n - k$ states are processed using bounded DSA
- Test whether the state is cacheable or not.

The various TD Algorithms

- The bounded and unbounded TD-DSA-AVC algorithm

Algorithm 0.6(The unbounded TD-DSA-AVC algorithm)

$$
\begin{array}{l}
\textbf{proc } t_{u13}(\delta, k, A, Z, s) \\
;q, j, p, l, B := 0, 0, 0, 0, A \\
;m[0..n), c[0..V), i[0..n) := [0..n), [0..V), -1 \\
;\textbf{do } (j < s.len \wedge q \geq 0) \rightarrow \\
\qquad \textbf{if } q < k \rightarrow tdavc(\delta, m, c, i, l, V, j, q, s) \\
\qquad \| \quad q \geq k \rightarrow utddsa(\delta, A, Z, B, q, j, s) \\
\qquad \textbf{fi} \\
\textbf{od} \\
\textbf{if } q < 0 \rightarrow \{\text{return } \mathsf{F}\} \, \| \, q \geq 0 \rightarrow \{\text{return } \mathsf{T}\} \textbf{ fi}
\end{array}
$$

The various TD Algorithms

- The bounded and unbounded TD-SpO-DSA-AVC algorithm
  - It is a combination of:

    SpO strategy;

    Bounden or Unbounded DSA strategy and

    AVC strategy.
  - A simple policy for the unbounded TD-SpO-DSA-AVC implementation:
    * reorder the states in δ based on entries of an auxiliary array $p_{[0..n]}$
    * Apply the unbounded TD-DSA-AVC procedure discussed in the previous slide
    * Access to states information is done indirectly via p

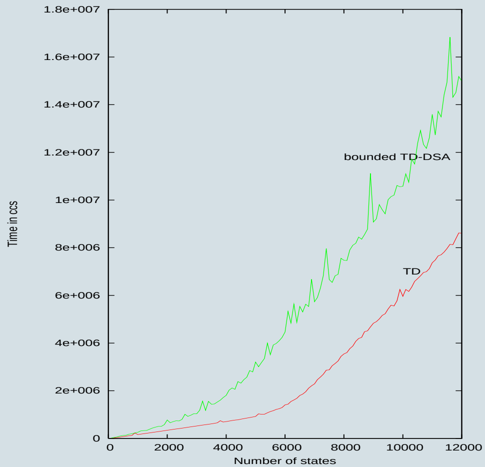
The various TD Algorithms

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

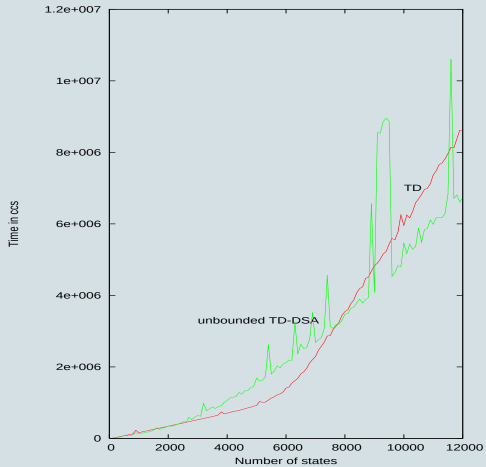
Algorithm 0.7(The unbounded TD-SpO-DSA-AVC algorithm)

$$
\begin{array}{l}
\textbf{proc } t_{u123}(\delta, p, s, c, k, d, A, Z) \\
;reorder(\delta, p) \\
\{ \ Initializations \ \} \\
;\textbf{do } (q < s.len \wedge q \geq 0) \rightarrow \\
\qquad \textbf{if } q < k \rightarrow tdavc(\delta, p, m, c, i, l, V, j, q, s) \\
\qquad \| \quad q \geq k \rightarrow utddsa(\delta, p, A, Z, B, q, j, s) \\
\qquad \textbf{fi} \\
\textbf{od} \\
\textbf{if } q < 0 \rightarrow \{\text{return } \mathsf{F}\} \ \| \ q \geq 0 \rightarrow \{\text{return } \mathsf{T}\} \ \textbf{fi}
\end{array}
$$

Experimental Results

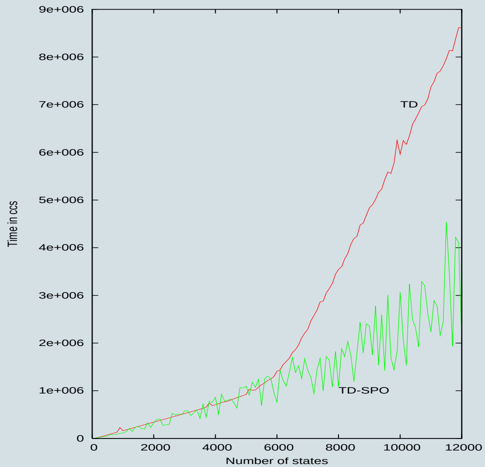


Bounded TD-DSA vs. TD

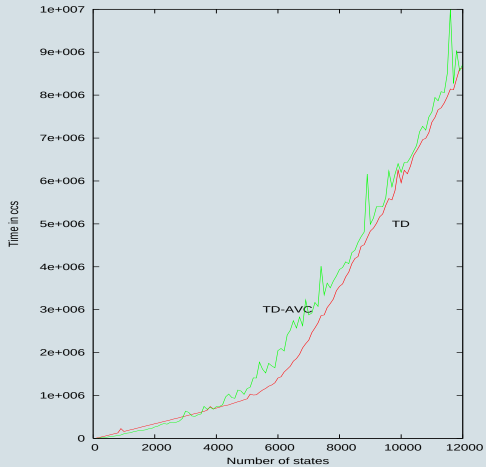


Unbounded TD-DSA vs. TD

Experimental Results

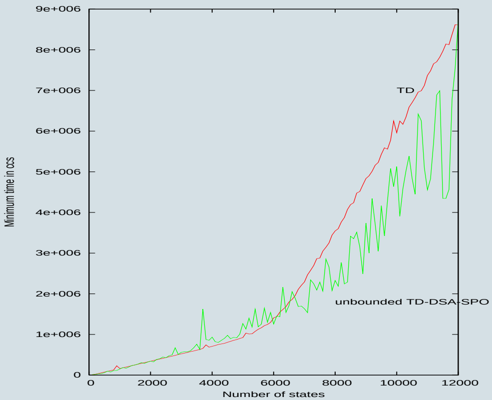


TD-SpO vs. TD

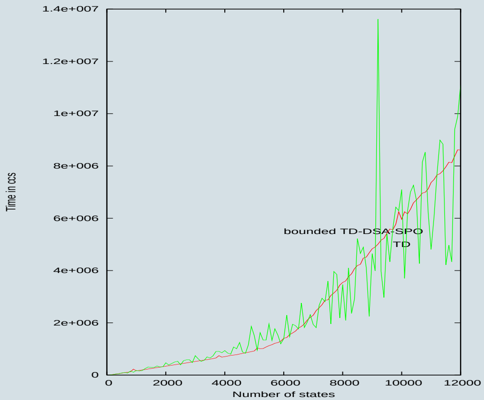


TD-AVC vs. TD

Experimental Results

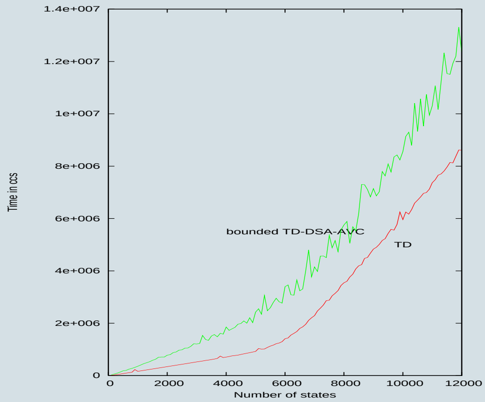


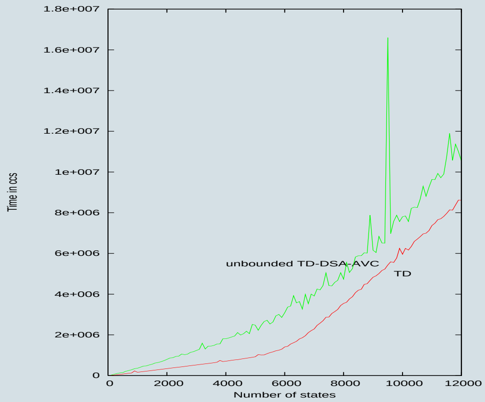
Unbounded TD-DSA-SpO vs. TD



bounded TD-DSA-SpO vs. TD

Experimental Results

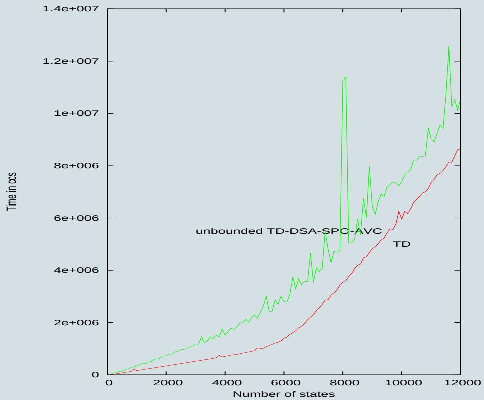




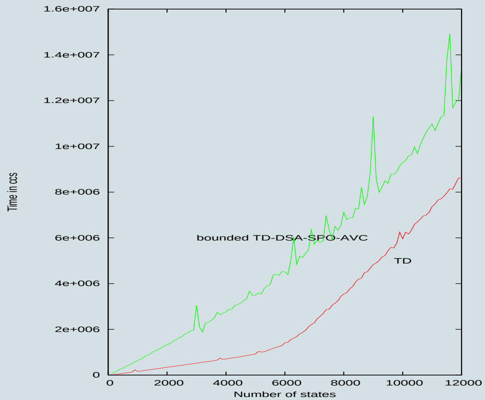
Bounded TD-DSA-AVC vs. TD   unbounded TD-DSA-AVC vs. TD

UNISA university of south africa

Experimental Results

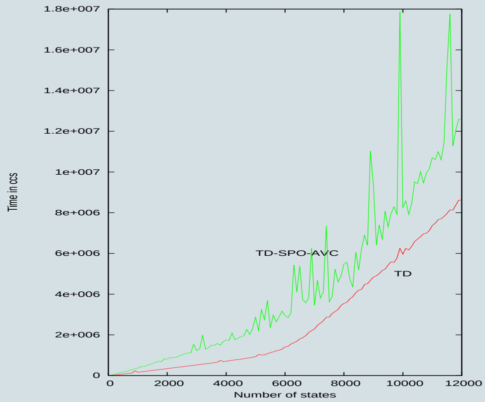


Unbounded TD-DSA-SpO-AVC vs. TD



Bounded TD-DSA-SpO-AVC vs. TD

Experimental Results



TD-AVC vs. TD

Conclusion and Future Work

- We have investigated various ways of improving the performance of DFA-based string processors
  - A 6-arguments function provided the denotational semantics of a string recognizer
  - Instantiations of the arguments resulted in 12 different table-driven DFA-based string recognizers
  - Some algorithms outperformed the core TD algorithm for strings mae of long sequences
  - Some algorithm were not of interest due to the replacement policy used during acceptance testing

As a matter of future work:

- Need to explore in dept the replacement policy in used
- need to test the algorithm on real life data