ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

INDIAN JOURNAL OF SCIENCE AND TECHNOLOGY
March 2020, Vol 13(10), 1160 – 1171
DOI: 10.17485/ijst/2020/v13i10/149803,

A Novel Negative Selection Algorithm with Optimal Worst-case Training Time Complexity for R-chunk Detectors

Nguyen Van Truong[1,*] and Nguyen Xuan Hoai[2,3]

[1]Department of Mathematics, Thai Nguyen University of Education, Vietnam
[2]AI Academy Vietnam, Vietnam
[3]Ho Chi Minh City University of Technology (HUTECH), Vietnam

Article Type: Article

Article Citation: Nguyen Van Truong, Nguyen Xuan Hoai. A novel negative selection algorithm with optimal worst-case training time complexity for r-chunk detectors. *Indian Journal of Science and Technology*. 2020; 13(10), 1160-1171. DOI: 10.17485/ijst/2020/v013i10/149803

Received date: December 28, 2019

Accepted date: February 4, 2020

***Author for correspondence:**
Nguyen Van Truong
nguyenvantruong@dhsptn.edu.vn
Thai Nguyen University of Education, Vietnam

Abstract

Objectives: To generate complete and non-redundant detector set with optimal worst-case time complexity. **Methods:** In this study, a novel exact matching and string-based Negative Selection Algorithm utilizing r-chunk detectors is proposed. Improved algorithms are tested on some data sets; the experiments' results are compared with recently published ones. Moreover, algorithms' complexities are also proved mathematically. **Findings:** For string-based Artificial Immune Systems, r-chunk detector is the most common detector type and their generation complexity is one of the important factors considered in the literature. We proposed optimal algorithms based on automata to present all detectors. **Novelty/applications:** The algorithm could generate the representation of complete and non-redundant detector set with optimal worst-case time complexity. To the best of our knowledge, the algorithm is the first one to possess such worst-case training time complexity.

Keywords: Artificial Immune Systems, Negative Selection Algorithms, Positive Selection Algorithms, Detector Sets, Self, Non-self.

1. Introduction

The biological immune system is a cooperative system that provides a comprehensive line of defense for human against pathogens. After million years of evolution, it has become a defensive system that is adaptive, inherently distributed, and incredibly robust. It possesses powerful capabilities such as pattern recognition, learning, and memory which helps to combat infections caused by pathogens (such as viruses), even though it needs no central control or coordination.

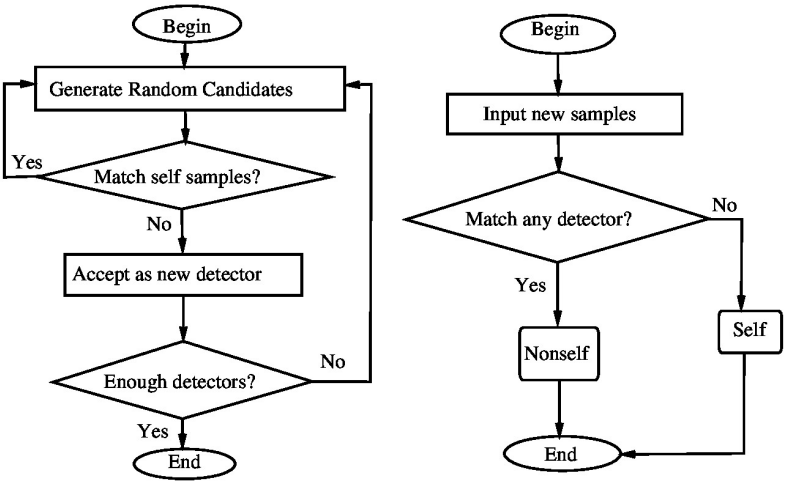
The main player in the biological immune system is the T cells, which could recognize selves and contain an antigen receptor for locating and binding to infected pathogens (non-selves). For detecting non-selves, the biological immune system conducts its learning process in two steps, which does not require any negative example. First, a large number of T cells are randomly generated in the hope to detect large number of pathogens. Then, the selection process acts on the newborn T cells to ensure that they could only recognize non-self not self (to avoid autoimmune reactions). In the case a T cell detects a self (such as a protein), this cell is discarded; otherwise, it is retained [1]. Algorithms that are abstracted and inspired from this selection process are named Negative Selection Algorithms (NSAs).

NSA is mainly created for leveraging one-class learning tasks such as in the problem of anomaly detection. A NSA comprises of two phases: the detector generation phase that aims at generating a set D of detectors from a given set S of selves and the detection phase for detecting if a given cell (a new data sample) is self or non-self with the help of the generated detector set.

NSA is the most well-known technique of Artificial Immune Systems (AISs), the class of computational methods inspired by the biological immune system. There have been an extensive number of studies on NSAs in the literature resulting various algorithm modifications and applications [2]. Since its introduction, NSAs have been applied in computer virus detection [3–4], intrusion detection [5], anomaly detection [6–8], monitoring UNIX processes, scheduling [9], fault detection and diagnosis [10], email spam detection [11], to name but a few.

Moreover, NSAs have also been applied in immunology, where they are used as models to provide insights into some important principles of immunity and infection [12], and to illustrate the immunological processes such as HIV infection [13–14].

A NSA typically proceeds in two phases: the detector generation (training) and detection phases [15]. Figure 1a gives the flowchart of the first phase, where the candidate



(a) Generation of detector set (b) Detection of new instances

FIGURE 1. Flowcharts of a negative selection algorithm.

detectors are first randomly generated by some processes. They are then censored by being matched against the self-sample data given by a set S, where S might represent the system components. Any candidate detector that matches (at least) one element of S is discarded and the ones that survive are retained and stored in a set called the detector set. The flowchart of the detection phase is given in Figure 1b. It is used to discriminate between selves (system components) and non-selves (anomalies, outliers) in that if a new data instance matches any detector in the detector set, it is regarded as non-self [16].

For string-based NSAs, the two most well-known matching rules for the construction of detector sets are r-contiguous and r-chunk. For both rules, a major problem with existing NSA implementations is that the first phase, detector generation, might have, in the worst case, exponential time complexity. The state-of-the-art algorithm for generating complete and non-redundant detector sets proposed by Elberfeld and Textor [17] possesses time complexities of $O(|S|\ell\ r|\Sigma|)$ and $O(\ell)$, for the detector generation (training) phase and detection phase, respectively. While the worst-time complexity for the detection phase is optimal (linear time), it is still open if the training time of the NSAs in could be improved further. In this article, we will show that at least in the case of r-chunk matching rule, improvement on training time complexity could be made by proposing a fast r-chunk based NSA for generating non-redundant detector sets, which requires only $O(|S|\ell|\Sigma|)$ while still maintains (worst case) time complexity of $O(\ell)$ for the detection phase. It is noted that the reduction of r in the training time complexity is substantial as in some applications such as intrusion detection, r could be approximately 50 [18]. Moreover, it can be easily shown that such (worst-case) training time complexity is optimal (i.e. it could not be further improved).

Table 1 summarizes the (worst-case) time complexity the previously published r-chunk detector-based algorithms and our proposed algorithm. In [17] the table and the rest of the article, it is assumed that binary alphabet is used ($|\Sigma| = 2$). It is noted that the parameter $|D|$ in Table 1 is only relevant for the algorithms that generate detectors in explicit form. Our algorithm and the algorithm in produce the results that obtain maximal number of generated detectors [19–20].

The organization of the rest of the paper is as follows. Some basic terminologies and definitions related to (string) languages, automata, and matching rules (r-chunk, r-contiguous) are given in the next section. Section 3 details our proposed r-chunk based negative selection algorithm. Experiments and discussions are given in Section 4. Finally, the paper is concluded with Section 4, where we will also highlight some possible future works.

TABLE 1. Training and detection (worst-case) time complexities of string-based NSAs

Algorithms	Training	Classification
In [18]	$(2^r+\|S\|)(\ell-r+1)$	$\|D\|\ell$
In [19]	$r^2\|S\|(\ell-r)$	$\|S\|\ell^2 r$
In [16]	$\|S\|\ell r$	ℓ
Present paper	$\|S\|\ell$	ℓ

2. Backgrounds

For being self-contained and consistent with in this section, some basic concepts are defined using similar notations as in Ref. [17].

2.1. Strings, Substrings, Languages

Let Σ be a finite s (non-empty) set of symbols called an alphabet, we define Σ^* as the set of all strings on Σ, that is any string $s \in \Sigma^*$ comprises of a sequence of symbols taken from Σ. For each string s, the number of symbols in s defines its length (denoted as $|s|$). When $|s|=0$, s is called the empty string.

$\forall$ i, j $\in\{1,...,|s|\}$ and i $\leq$ j, s [i] represents the symbol at position i in s and s [i...j] denotes the substring of s with length j − i + 1 defined as the subsequence of symbols starting at position i running to position j in s. When the substring s′ is located at the beginning (end) of s, i = 1 (j = $|s|$), it is call the prefix (suffix) of s. s′ is proper if $|s'| < |s|$. Given $s \in \Sigma^{\ell}$, d $\in \Sigma^{r}$, $1 \leq r \leq \ell$, and i $\in \{1,...,\ell - r + 1\}$, if s [i...i + r − 1] = d, then d is said to occur in s (at position i).

A language S over Σ is defined as a set of strings, i.e. $S \subseteq \Sigma^*$. Given i and j, we define S[i... j] = {s[i... j] | s ∈ S} as the set of all substrings (from position i to position j) in language S.

2.2. Prefix Trees, Prefix Directed Graphs, Automata

A rooted and directed tree T with edge labels from Σ is called a prefix tree over alphabet Σ if for all c ∈ Σ and every node n in T, n has no more than one outgoing edge labeled with c. A tree T contains a string s (s ∈ T) if, there is a path p ∈ T from the root to a leaf of T such that the string concatenated along p equals s.

For a given tree T, the language L(T) = {s|s has a nonempty prefix in T}. For instance, given T as in Figure 2a, we could assert that 10 ∈ T and 0 ∈ T, but 1 ∉ T. Therefore, 0 ∈ L(T) and 01∈ L(T) as 0∈ T, but 11∉ L(T) since T does not contain any prefix of 11.

Similar to prefix trees, a prefix DAG D could be defined as a directed acyclic graph, where its edges have labels as the symbols from an alphabet Σ. A string s ∈ D if there is a path p from a root to a leaf of D such that the string concatenated along p equals s.

For a node n in D, we define the language L(D, n) as the set of all strings s such that s has a (nonempty) prefix equaling the concatenated sequence of labels on the path from n to some leaves in D.

For example, for the DAG D in Figure 2b and its lower left node n, L(D, n) comprises of all strings that start with 11. We also define language $L(D) = \cup_{misarootofD} L(D,m)$.

A finite automaton is defined as a five-tuple $M = (Q, q_0, Q_a, \Sigma, \Delta)$, where Q is a set of states with $q_0 \in Q$ is called the initial state, $Q_a \subseteq Q$ is the set of accepting states, Σ is the alphabet of M, and $\Delta \subseteq Q \times \Sigma \times Q$ is the transition map. The transition map is considered unambiguous in that for any q ∈ Q and c ∈ Σ, there is no more than one q′ ∈ Q with (q, c, q′) ∈ Δ. We could use a graph G = (V, E) to represent the transition relation Q of an automaton M by setting the node set V = Q and E = c-labeled edges, where a c-labeled edge is identified from q to q′ for any q, q′ ∈ Q if (q, c, q′) ∈ Δ.

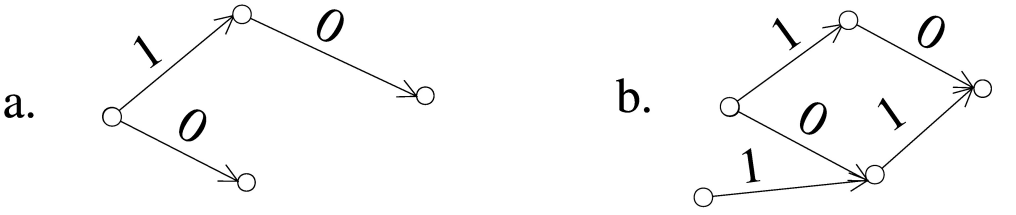


FIGURE 2. Example of a prefix tree and a prefix DAG.

An string s is accepted by a automaton M if in the transition graph of M there is a path from q_0 to some $q \in Q_a$ such that its concatenated sequence of symbols equals s. The set of all strings accepted by an automaton M is its language L(M).

2.3. Detectors

Given an alphabet Σ, a string s ($|s|=\ell$), a self-set $S \subseteq \Sigma^\ell$, and $r \in \{1,...,\ell\}$ (called matching parameter), We could define r-chunk detectors as follows [17].

Definition 1. A r-chunk detector is a tuple (d, i), where $d \in \Sigma^r$ is a string of length r and i is a position ($i \in \{1,...,\ell - r + 1\}$). An r-chunk detector d is said to match a string s if d occurs in s at (at least) one position i.

Given a set of strings S, the set of r-chunk detectors that do not match any string in S, denoted as CHUNK(S, r), is called the detector set for S. A string $m \in \Sigma^\ell$ is called a non-self w.r.t. S and its r-chunk detector set if m matches at least one detector from CHUNK(S, r); Otherwise, m is regarded as self. The set of non-self of S w.r.t its r-chunk detectors, is denoted as We denote CHUNK-NONSELF(S, r).

For every $i \in \{1,...,\ell-r+1\}$, we have CHUNK(S[i...i + r−1],r) $\cup$ S[i...i + r−1] $=\Sigma^r$, where S[i...i + r−1] is called the positive detector set. In the next section, we will use a compression of all positive detectors, S[i...i + r−1], i=1,..,ℓ−r + 1, as a temporary data structure before inverting it to the representation of CHUNK(S, r).

Another popular form of detectors for NSAs is r-contiguous, which is defined as follows [17].

Definition 2. An r-contiguous detector can be any string $d \in \Sigma^\ell$. d is said to match a string $s \in \Sigma^\ell$ if there is a position $i \in \{1,..., \ell - r + 1\}$ such that d[i...i + r − 1] is a substring of s.

Similar to r-chunk detectors, we denote the set of all r-contiguous detectors not matching any string in S as CONT(S, r). A string $m \in \Sigma^\ell$ is non-self if it matches at least a r-contiguous detector in CONT(S, r). Otherwise, it is called self.

Since a r-contiguous detector can be decomposed into $\ell - r + 1$ overlapping r-chunk detectors, r-chunk is considered as a simplification of the r contiguous n matching rule [21]. It has been showed in Ref. [22] that chunk-based detectors could help NSAs work well on problems where contiguous regions in the sequence of input data are not semantically correlated, e.g. when the input sequence are network data packets.

For the sake of comparison, we reuse the example from Ref. [17].

Example 1. Let $\ell = 5$, r = 3 and S = {s_1 = 01111, s_2 = 00111, s_3 = 10000, s_4 = 10001, s_5 = 10010, s_6 = 10110, s_7 = 11111}. We could obtain CHUNK(S, r) = {(000,1), (010,1), (110,1), (010,2), (100,2), (101,2), (110,2), (011,3), (100,3), (101,3)} and CONT(S, r) = {01011, 11011}.

3. Negative Selection Algorithm with Chunk Detectors

Suppose that each self-string of S has an associated index, I = {s_i, i = 1,.., |S|}, We introduce the following two important data structures:

- A two-dimensional array Q, where Q[s][c] is a pointer used for creating new nodes in the tree, $s \in \Sigma^{r-1}$, and $c \in \Sigma$. This data structure is used for gradually expanding the partial DAG.
- An array P, where P[i] is a structure of two fields, a pointer P[i].end and a string P[i].str $\in \Sigma^{r-1}$, i = 1,..,|S|. Each P[i] is joined with s_i, where P[i].end is used to point to the end node in the path of s_i in the prefix DAG and P[i].str is a suffix of the path ended by P[i].end.

Two arrays Q and P are used to expand the prefix DAG step by step. The final automaton to decide the membership of strings in Σ^ℓ is constructed in two stages. The first is to create a DAG G so that $L(G) \cup \Sigma^\ell = \Sigma^\ell \backslash$CHUNK-NONSELF(S,r) by algorithm (Algorithm 1) and this DAG is then turned into an automaton M such that $L(M) \cup \Sigma^\ell$= CHUNK-NONSELF(S,r) in the second stage by algorithm (Algorithm 2). In algorithm 1, the notations NULL and new() are used with the usual semantics as in C programming language.

The key idea behind our construction is that instead of generating all $\ell-r+1$ prefix trees $T_1,...,T_{\ell-r+1}$ as in Ref. [16], we only create one tree T_1 for S[1..r] explicitly and then enlarge it by adding nodes and edges level-by-level to attain a prefix DAG. After this process, the DAG encodes all positive detectors S[i...i+r−1], i = 1,...,ℓ−r+1. This is then inverted to construct a compression of CHUNK(S, r).

Algorithm 1 To generate positive r-chunk detectors set

1. **Procedure** Positive R-chunk Detector (S, ℓ, r, G)
2. G = ∅
3. For i = 1,...,|S|do
4. insert s_i[1... r] into G and assign P[i].end to the leaf node in path s[1... r]
5. P[i].str = s[2... r]
6. For i = r+1,...,ℓ do
7. For j = 1,...,|S| do
8. If Q[P[j].str][s_j[i]] = NULL then
9. Q[P[j].str][s_j[i]] = new()
10. For j = 1,...,|S| do
11. p = P[j].end
12. For c ∈Σ

13. If(Q[P[j].str][c] ≠ NULL)and(edge starts from p with label c does exist) then
14. create an edge starts from p with label c to Q[P[j].str][c]
15. P[j].end = end node of the edge starts from p with label $s_j[i]$
16. For j = 1,...,|S| do
17. For c ∈Σ do
18. Q[P[j].str][c]=NULL
19. P[j].str = P[j].str[2...r−1]+$s_j[i]$

Algorithm 2 To generate negative r-chunk detectors set
1. **Procedure** Negative R-chunk Detector (S, ℓ, r, G)
2. Positive R-chunk Detector (S, ℓ, r, G)
3. create a special node n′
4. For every non-leaf node n ∈ G do
5. For c ∈ Σ do
6. If no edge with label c starts at n then
7. create new edge (n,n′) labeled with c
8. For every node n ∈ G do
9. If n is not reachable to n′ then
10. delete n

Example 2. Let ℓ = 5, r = 3 and S is self-set from Example 1. The prefix DAG generated by Algorithm 1 is illustrated in Figure 3a. After adjusting by Algorithm 2, this DAG can be turned into an automaton as in Figure 3b.

Based on same self-set from Example 1, the automaton generated by the algorithm in Ref. [17] contains 23 nodes and 25 edges, while the automaton in Figure 3b has 14 nodes and 20 edges only. This supports in part a better memory complexity of our proposed algorithm.

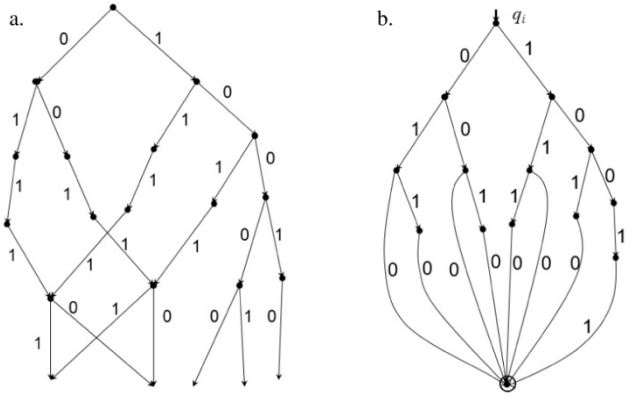


FIGURE 3. A prefix DAG G in a. is generated by Algorithm 1 and an automaton M in b. is turned from the DAG generated by Algorithm 2. $L(G)\cap\{0,1\}^5 = \{0,1\}^5 \setminus$CHUNK-NONSELF(S,3) and $L(M)\cup\{0,1\}^5$ = CHUNK-NONSELF(S,3), where S is self-set from example.

Algorithm 3, CHUNK-NSA, summarizes the overall of the process of our NSA. In the algorithm, the parameters include a self-set S, an integer r ($r \in \{1,...,\ell - r + 1\}$), a self-string s to be detected, and a prefix DAG G. CHUNK-NSA will detect s as self or non-self.

Algorithm 3 A fast r-chunk detector-based NSA
1. **Procedure** Chunk-NSA(s, S, ℓ, r, G, M)
2. Negative R-chunk Detector(S, ℓ, r, G)
3. turn G into an automaton M
4. If s ∈ L(M) then
5. output s is nonself
6. Else
7. output s is self

Theorem 1. Given any $S \subset \Sigma^{\ell}$ and $r \in \{1,...,\ell\}$, algorithm CHUNK-NSA constructs an automaton M such that $L(M) \cap \Sigma^{\ell}$ = CHUNK-NONSELF(S,r) in time $O(|S|\ell|\Sigma|)$ and checks if $s \in L(M)$ in time $O(\ell)$.

Proof: We first construct a prefix DAG G as follows: starting with an empty prefix DAG, and also a prefix tree in this case, the algorithm inserts every $s \in S[1...r]$ into it. This was done by the first for loop in Algorithm 1 in time $|S|.r.|\Sigma|$.

Then for each s[i], s∈ S, and $i = r + 1,..,\ell$, we add nodes and corresponding edges to G by using linking pointers of P and Q as in lines 6–19 in algorithm 1. The first inner for loop (line 7) is to create new pointer to expand G to next level. Note that, P[i] is joined with s_i as mentioned at the beginning of this section. The second inner for loop (line 10) is to add new nodes and corresponding edges to G. The for loop in line 16 to free the pointers in Q and to update string in P.

It needs only (ℓ−r) iterative steps to generate G that presents all positive r-chunk detectors. There are three inner for loops (lines 7, 10, 16), each with |S| steps. Therefore, the time complexity of Algorithm 1 is $|S|+3|S|(\ell-r+1)|\Sigma|$.

Algorithm 2 inverts the DAG G as follows: Firstly, a special node n′ is created. Then for every node n that is not a leaf and every symbol $c \in \Sigma$ for which there is no edge starting at n and labeled with c, we create a new edge (n,n′) with c as its label. Finally, we delete every node n which is not reachable to n′. There are no more than $|S|\ell|\Sigma|$ nodes in the DAG G, therefore, the time complexity of algorithm 1 is $|S|\ell|\Sigma|$, or $O(|S|\ell)$.

The DAG G generated by Algorithm 1 is turned into a finite automaton M by making the leaf node n′ as an accepting state with self-loops for all $c \in \Sigma$ and the root of G is set as the initial state of M. Now, we obtain automaton M that has the properties claimed by the theorem. Moreover, it is easy to see that it take $O(\ell)$ to check whether $s \in L(M)$ or not. So we obtain the claimed runtimes. □

Obviously, any algorithm that generate a complete CHUNK(S, r) set if and only if it reads all training data at least one time. Consequently, we obtain the following corollary.

Corollary 1. The algorithm CHUNK-NSA has optimal worst-time complexity w.r.t. generating a representation of complete CHUNK(S,r) set.

4. Experiments and Results

All experiments run on Windows 8 Pro 64-bit, Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz (4 CPUs), 4GB RAM.

In our experiments, we use a popular flow-based datasets NetFlow and a random dataset. The flow-based NetFlow is generated from packet-based DARPA dataset [23] is used for experiment 1. It was encoded as a set of 105,033 binary strings with their length of 57. A randomly created dataset containing 50,000 strings of length 100 is used for experiment 2. The parameter values and running times (in milliseconds) of the experiments are showed in Table 2.

In Table 2, the runtime of NSA in Ref. [16] for both experiments are in shown in columns a and c, respectively, while the runtime of our proposed NSA are given in columns b and d. The results in the table show that there is a positive correlation between threshold r and the ratio of the runtime of NSA in to the runtime of CHUNK-NSA (as in columns a/b and c/d).

Figure 4 shows that, for random strings in experiment 2, the ratio is closer to r when r increases (i.e. our proposed NSA almost r times faster than the NSA proposed in Ref. [24]. The ratio increases slower as r increases in experiment 2 (for real data) approaching r/3. Overall, the experimental results are consistent with our theoretical proof (Theorem 1).

TABLE 2. Comparison of proposed chunk-NSA with NSA

r	Experiment 1			Experiment 2		
	a	b	a/b	c	d	c/d
10	1330	454	2.9	1490	482	3.1
11	1395	439	3.2	1633	472	3.5
12	1564	454	3.4	637	360	4.5
13	1767	435	4.1	2134	453	4.7
14	1771	418	4.2	2276	451	5.0
15	2092	486	4.3	2793	450	6.2
16	1985	437	4.5	3086	365	8.5
17	2071	391	5.3	4079	427	9.6
18	2249	410	5.5	4509	422	10.7
19	2345	375	6.3	5312	470	11.3
20	2859	359	7.0	6796	437	15.6

5. Conclusions

In this study, we have introduced a new NSA to generate complete and non-redundant r-chunk detector sets with optimal worst-time complexity. Our theoretical proof and empirical experiments show that the proposed r-chunk detector algorithm, CHUNK-NSA, trains much faster the state-of-the-art one.

A limitation of CHUNK-NSA is that it is more memory consuming than the NSA as it utilizes two extra arrays Q and P with the memory complexities of $|\Sigma|^r$ and $|S|r$, respectively. In our opinion, this drawback is not serious as the modern computing systems could support huge internal memory storage, especially when $|\Sigma|$ is small (e.g.

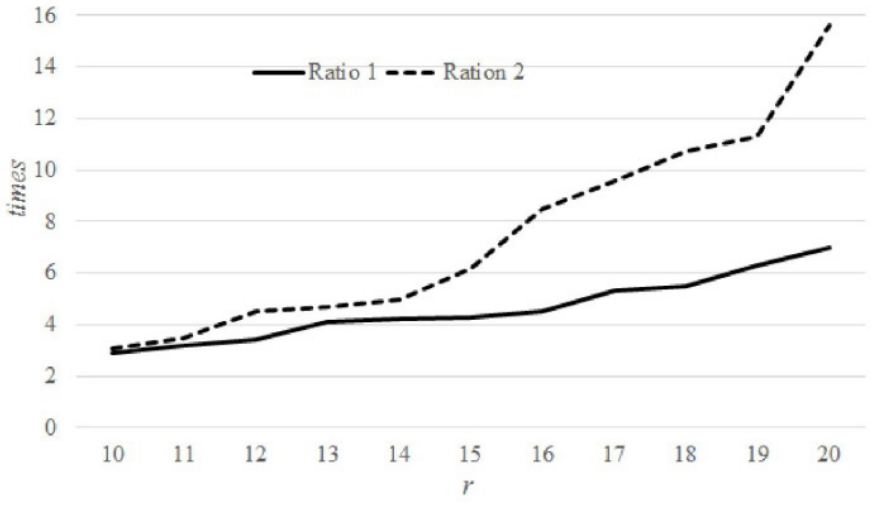


FIGURE 4. Comparison of ratios of runtime of NSA into runtime of CHUNK-NSA.

binary) and r is not big (e.g. 50–60 in applications for network security).We anticipate that the approach can be further developed for r-contiguous detector-based NSAs. This will be our immediate future research direction.

Acknowledgment

This research is supported in part by Thai Nguyen University via a university research grant (grant number DH2017-TN01-03).

References

1. Dasgupta D. artificial immune systems and their applications. Springer-Verlag: Berlin Heidelberg. 1998; 1–16. https://link.springer.com/content/pdf/bfm%3A978-3-642-59901-9%2F1.pdf
2. Zhou J, Dipankar D. Revisiting negative selection algorithms. *Evolutionary Computation*. 2007, 14, 223–251. DOI: 10.1162/evco.2007.15.2.223.
3. Forrest S, Javornik B, Smith RE, Perelson AS. Using genetic algorithms to explore pattern recognition in the immune system. *Evolutionary Computation*. 1993, 1, 191–211. https://www.cs.unm.edu/~forrest/publications/immune-92.pdf
4. Suha A, Raed AZ, Alaa AH. Virus detection using clonal selection algorithm with Genetic Algorithm. *Applied Soft Computing*. 2013, 13(1), 239–246. https://doi.org/10.1016/j.asoc.2012.08.034
5. Textor J. A comparative study of negative selection based anomaly detection in sequence data. In: International conference on artificial immune systems. 2012; 28–41. https://link.springer.com/chapter/10.1007/978-3-642-33757-4_3

6. Dipankar D, Stephanie F. Novelty detection in time series data using ideas from immunology. In: International conference on intelligent systems. 1995; 1–8. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.9949&rep=rep1&type=pdf
7. Dong L, Shulin L, Hongli Z. A negative selection algorithm with online adaptive learning under small samples for anomaly detection. *Neurocomputing*. 2015; 149, 515–525. https://doi.org/10.1016/j.neucom.2014.08.022
8. Fan Z, Tao CWaL, Xiaochun C, Haipeng P. An antigen space triangulation coverage based real-value negative selection algorithm. *IEEE Access*. 2019, 7, 51886–51898. **DOI:** 10.1109/ACCESS.2019.2911660.
9. Murugesan R, Kumar VN. A fast algorithm for solving JSSP. *European Journal of Scientific Research*. 2011, 64, 579–586. https://www.researchgate.net/publication/287164942_A_fast_algorithm_for_solving_JSSP
10. Guilherme CS, Reinaldo MP, Walmir MC. Immune inspired fault detection and diagnosis: a fuzzy-based approach of the negative selection algorithm and participatory clustering. *Expert Systems with Applications*. 2012, 39, 12474–12486. https://doi.org/10.1016/j.eswa.2012.04.066
11. Chikh, R., Chikhi, S. Clustered negative selection algorithm and fruit fly optimization for email spam detection. *Journal of Ambient Intelligence and Humanized Computing*. 2019, 10, 143–152. http://doi-org-443.webvpn.fjmu.edu.cn/10.1007/s12652-017-0621-2
12. Butler TC, Kardar M, Chakraborty AK. Quorum sensing allows T cells to discriminate between self and nonself. *Proceedings of the National Academy of Sciences*. 2013, 110(29), 11833–11838. https://doi.org/10.1073/pnas.1222467110
13. Košmrlj A, Read E, Qi Y, Allen T, Altfeld M, Deeks S. Effects of thymic selection of the T-cell repertoire on HLA class I-associated control of HIV infection. *Nature*. 2010, 465, 350–354. DOI: 10.1038/nature08997.
14. Košmrlj A, Jha AK, Huseby ES, Kardar M, Chakraborty AK. How the thymus designs antigen-specific and self-tolerant T cell receptor sequences. *Proceedings of the National Academy of Sciences*. 2008, 105(43), 16671–16676. https://doi.org/10.1073/pnas.0808081105
15. Ji Z. Negative selection algorithms: from the Thymus to V-detector. The University of Memphis. 2006.
16. Forrest S, Perelson AS, Allen L, Cherukuri R. Self-nonself discrimination in a computer. In: IEEE symposium on security and privacy. 1994; 202–212. DOI: 10.1109/RISP.1994.296580.
17. Michael E, Johannes T. Negative selection algorithms on strings with efficient training and linear-time classification. *Theoretical Computer Science*. 2011, 412(6), 534–542. https://doi.org/10.1016/j.tcs.2010.09.022
18. Dipankar D, Gonzalez F. An immunity-based technique to characterize intrusions in computer networks. *IEEE Transactions on Evolutionary Computation*. 2002, 6(3), 281–291. https://doi.org/10.1109/TEVC.2002.1011541
19. Stibor T, Bayarou KM, Eckert C. An investigation of R-chunk detector generation on higher alphabets. In: Genetic and evolutionary computation conference. 2004; 299–307. https://link.springer.com/chapter/10.1007/978-3-540-24854-5_31
20. Michael E, Johannes T. Efficient algorithms for string-based negative selection. In: International conference on artificial immune systems. 2009; 109–121. DOI: 10.1007/978-3-642-03246-2_14.
21. Fernando E, Stephanie F, Paul H. The crossover closure and partial match detection. In: International conference on artificial immune systems. 2003; 249–260. https://link.springer.com/chapter/10.1007/978-3-540-45192-1_24
22. Justin B, Fernando E, Stephanie F, Matthew G. Coverage and generalization in an artificial immune system. In: Genetic and evolutionary computation conference. 2002; 3–10. https://dl.acm.org/doi/10.5555/2955491.2955493

23. Quang AT, Frank J, JIankun H. A real-time NetFlow-based intrusion detection system with improved BBNN and high-frequency field programmable gate arrays. In: IEEE international conference on trust, security and privacy in computing and communications. 2012; 201–208. **DOI:** 10.1109/TrustCom.2012.51.
24. DARPA dataset. https://www.ll.mit.edu/r-d/datasets. Date accessed: 09/2016.