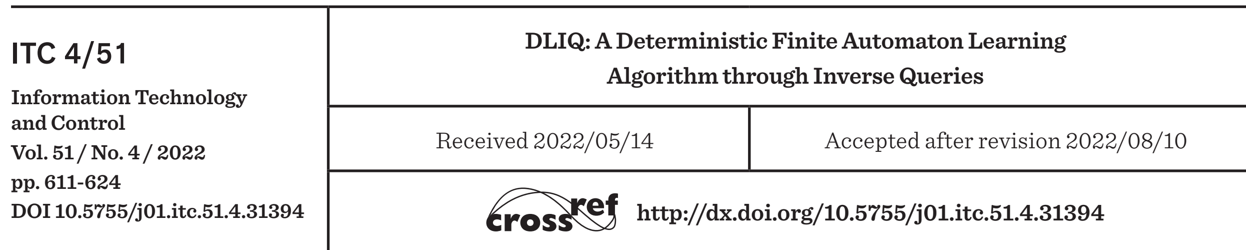

ITC 4/51
Information Technology and Control
Vol. 51 / No. 4 / 2022
pp. 611-624
DOI 10.5755/j01.itc.51.4.31394

DLIQ: A Deterministic Finite Automaton Learning Algorithm through Inverse Queries

Received 2022/05/14 | Accepted after revision 2022/08/10

http://dx.doi.org/10.5755/j01.itc.51.4.31394

**HOW TO CITE:** Haneef, F., Sindhu, M. A. (2022). DLIQ: A Deterministic Finite Automaton Learning Algorithm through Inverse Queries. *Information Technology and Control*, 51(4), 611-624. http://dx.doi.org/10.5755/j01.itc.51.4.31394

# DLIQ: A Deterministic Finite Automaton Learning Algorithm through Inverse Queries

**Farah Haneef, Muddassar A. Sindhu**

Department of Computer Science, Quaid i Azam University, Islamabad. 45320. Pakistan

**Corresponding author:** farah@cs.qau.edu.pk

Automaton learning has attained a renewed interest in many interesting areas of software engineering including formal verification, software testing and model inference. An automaton learning algorithm typically learns the regular language of a Deterministic Finite Automaton (DFA) with the help of queries. These queries are posed by the learner (Learning Algorithm) to a Minimally Adequate Teacher (MAT). The MAT can generally answer two types of queries asked by the learning algorithm; membership queries and equivalence queries. Learning algorithms can be categorized into three broad categories: incremental, sequential and complete learning algorithms. Likewise, these can be designed for 1-bit learning or $k$ -bit learning. Existing automaton learning algorithms have polynomial (at-least cubic) time complexity in the presence of a MAT. Therefore, sometimes these algorithms are unable to learn large complex software systems. In this research work, we have reduced the time complexity of the DFA learning into lower bounds (from cubic to square form). For this, we introduce an efficient complete DFA learning algorithm through Inverse Queries (DLIQ) based on the concept of inverse queries introduced by John Hopcroft for state minimization of a DFA. The DLIQ algorithm takes $O(|P_s||F|+|\Sigma|N)$ complexity in the presence of a MAT which is also equipped to answer inverse queries. We give a theoretical analysis of the proposed algorithm along with providing an empirical analysis of DLIQ and ID (Identification of regular languages) algorithms. For this, we implement an evaluation framework. Results depict that in terms of time complexity our proposed algorithm DLIQ is more efficient than the ID algorithm.

**KEYWORDS:** Automaton learning, complete learning algorithm, delta inverse transitions, inverse query, live completeset, distinguishing string.

## 1. Introduction

Automaton learning or grammatical inference is a domain in which a system is inferred in the form of an automaton by providing a sequence of inputs ($i_1, i_2, . . . , i_n$) and then synthesizing the corresponding output sequence($o_1, o_2, . . . , o_n$) obtained from the *System Under Learning* (SUL) into a finite automaton. Automaton learning makes use of a *learner* (automaton learning algorithm) and a *Minimally Adequate Teacher* (MAT) [2]. The learner learns the regular set from queries and counterexamples depending upon the setup provided by the learning algorithm. Thelearner poses queries to the MAT which responds to those queries about the unknown regular set. It answers two typesof questions: First type is a *membership query*, consisting of the string $t \in \Sigma^*$. The adequate teacher answers as *yes* or *no* depending on whether string $t$ is a member of the unknown regular set or not. The second type of question is a conjecture, consisting of a description of the regular set $S$; the answer is yes if $S$ is behaviorally equivalent to the unknown language and is a string $t$ in the symmetric difference of $S$ and the unknown language otherwise. In the second case, the string $t$ is called a counterexample or a witness because it serves to demonstrate that the conjecturedset $S$ is incorrect.

Automaton learning algorithms are designed in a way to *learn in the limit* to yield a minimal approximation of the target DFA. The concept of learning in the limit was first introduced in 1967 by E. M. Gold [10]. In his paper, he showed that with the help of a grammatical inference or an automaton learning algorithm, a regular language corresponding to some unknown target DFA can be inferred by a finite number of queries / guesses. Three types of automaton learning algorithms have been proposed in the literature which include: complete learning algorithms, in- cremental learning algorithms and sequential learning algorithms. Examples of each category are provided in Section 3.

In recent years, the software engineering research community has used grammatical inference [21, 25, 28, 29] because of its possibility to solve a wide range of practical applications of formal verication [12], model inference [11, 20, 22, 24, 30] and software testing [4]. These applications in general use the concept of inferring an automatonby generating a model of a system under learn (SUL) and analyzing it to check its behavioral correctness with respectto a specification.

The existing automaton learning algorithms have polynomial time complexity (at-least in cubic form) in the presence of MAT. For testing and formal verification of large complex software systems, the existing automaton learning algorithms take a lot of time during inferring the model of System Under Learn (SUL). Sometimes, these algorithms even become fail to learn large complex software systems. In paper [32] authors specifically provide some real world examples of 6 transition systems of CSS processes [5–7] like buffers, schedulers, vending machines and mutual exclusion protocols [23, 34] where they fail o learn them due to inefficient learning algorithms (in form of time) and lack of storage space. In their paper, authors also emphasis on the need of some good automaton learning algorithm that may efficient enough in form of execution time and memory. According to the existing literature [3, 31] although sig- nificant work has been done in the development of DFA learning algorithms but many researchers have an agreement that there is still a need to design more efficient automaton learning algorithms to solve practical learning problems and situations [33].

For this reason, in this paper we introduce a new efficient DFA learning algorithm DLIQ based on the concepts of the ID (Identification of regular languages) algorithm [1] because the concept of minimal adequate teacher (MAT) was first introduced in ID algorithm and without this concept, the learning of a DFA is an NP-hard problem. Secondly, the concept of distinguishing string was also given in the ID algorithm which is used in our proposed algorithm along with inverse queries. The DLIQ algorithm learns the *System Under Learning* (SUL) by splitting it into final and non-final state blocks and then identifying behaviorally equivalent and non-equivalent states by traversing towards the initial state using inverse queries which were initially proposed by John Hopcroft for state minimization of DFAs [13].

The major contributions of the paper are given below:

1. We design and implement a new efficient DFA learning algorithm possessing worst case time complexity in square polynomial form unlike existing DFA learning algorithms.
2. We introduce and define the concept of $\delta^{-1}$ and *Inverse Queries* (IQ) in *DFA learning*.

3 We improve the complete learning process of DFAs by making use of *Inverse Queries* (IQ).

4 We enhance the capabilities of the existing *MAT* to make it capable to answer *Inverse Queries* (IQ).

5 We propose a method to generate live complete set of a randomly generated DFA, based on a procedure namedas *testing tree* which is used for generating covering sets from FSMs.

Rest of the paper is organized as follows: we give preliminaries to understand the proposed algorithm in Section 2, we review the related work of the field in Section 3. The proposed DLIQ algorithm is provided in Section 4, the proofs for correctness and termination of the DLIQ algorithm are given in Section 5, an example describing the working of the algorithm is shown in Section 6. In Section 7, we compare the performance of our proposed DLIQ algorithm with the ID algorithm because DLIQ is based on the concepts introduced in the ID algorithm and both these algorithms have similar characteristics which are unlike other DFA learning algorithm such as: the ID algorithm is also a complete learning algorithm, which employs a table data structure and it does not use the concept of counter- examples as done in the $L^*$ algorithm. At the end, we analyze and present the time complexity of the DLIQ algorithm in Section 8, and finally, we present conclusions along with some new research directions in Section 9.

## 2. Mathematical Preliminaries and Notation

A *Deterministic Finite Automaton* (DFA) $A$ consists of a quintuple $\langle Q, \Sigma, \delta, q_0, F\rangle$ where $Q$ denotes the finite set of states, $\Sigma$ is a finite set of input symbols, $\delta$ is the transition function which gives the next state when we read an input symbol from a specific state $\delta: Q\times\Sigma\rightarrow Q$. The state $q_0\in Q$ is the start state and $F\subseteq Q$ is the set of final states.

As $\Sigma$ is a finite set of input symbols and $\Sigma^*$ is the set of all finite length strings including the empty string $\lambda$. Let $\alpha,\beta,\gamma\in\Sigma^*$, if $\gamma=\alpha.\beta$ then $\alpha$ is called a prefix and $\beta$ is called a suffix of $\gamma$.

A state $q$ is called a live state if there exists strings $\alpha$ and $\beta$ such that $\alpha\beta\in L(A)$ and $q=\delta(q_0,\alpha)$ since $L(A)$ is the language accepted by $A$ [22], whereas a *Live Complete set* $P$ is a set of all strings which may lead to some live state of a DFA. We let $P=\{p_1,p_2,\ldots,p_n\}$ to be the live complete set of $n$ strings.

A state which is not live is called a dead state and we denote it by $d_0$. Mathematically, $d_0\notin F$ and $\delta(d_0,\alpha)=d_0$ for all $\alpha\in\Sigma^*$.

**Definition 2.1.** *Let for a DFA $A$, having transition function $\delta: Q\times\Sigma\rightarrow Q$ which can also be written as $\delta(q_i,\sigma)=q_j$ and the iterated transition function $\delta^*: Q\times\Sigma^*\rightarrow Q$ inductively defined by $\delta(q,\lambda)=q$ where $\lambda$ is an empty string and $\delta^*(q_j,b_1,b_2,\ldots,b_n)=\delta(\delta^*(q,b_1,b_2,\ldots,b_{n-1}),b_n)$*

Likewise, we inductively define $\delta^{-1*}$ using the inverse transition relation $\delta^{-1}$. Where $\delta^{-1}: Q\times\Sigma\subseteq Q$ and can also be written as $\delta^{-1}(q_j,\sigma)\subseteq Q$ where $\Sigma^{-1}$ denotes an inverse transition by reading an element of $\Sigma$ from a state $Q$ to give its predecessor states. The inductive definition of $\delta^{-1*}$ is now simple to follow as $\delta^{-1}(q,\lambda)=q$, if and only if $q$ is a starting state otherwise it returns $\varnothing$ and $\delta^{-1^*}(q_j,b_1,b_2,\ldots,b_n)=\delta^{-1}(\delta^{-1^*}(\{q\},b_2,\ldots,b_n),b_1)$.

In the automaton learning context we define the *Inverse Query* (IQ) as a query which is asked by the learner from the teacher about the predecessor state(s) of a state $q_j$, by reading some string $\alpha\in\Sigma^*$ from it, i.e., $\delta^{-1*}(q_j,\alpha)=?$ The teacher gives response as (Yes/No) based on the answer which can be a set of state(s) or an empty set.

A string $\alpha$ is called an accepting string which when read from the initial state $q_0$ of a DFA$(A)$ leads to some final state $F$ of the DFA$(A)$, i.e., $\delta^*(q_0,\alpha)\in F$, otherwise, when $\delta^*(q_0,\alpha)\notin F$ then $\alpha$ is termed a rejecting string.

A *block* is a set of states denoted by $B(num)$, the block of non-final states is initially specified as $B(1)$ and can be computed as $(Q-F)$ and the block of final states is specified as $B(2)=F$. Let $B(k)$ denote the $k^{th}$ block in the set of blocks. The size of a block $B(k)$ is denoted by $|B(k)|$, and gives the number of states in that block.

## 3. Related Work

In automaton learning [26, 27], there are three basic types of algorithms: incremental learning algorithms, sequential learning algorithms and complete learning algorithms. In incremental learning, the system under learning (SUL)is learnt in a number of increments $i=1,2,\ldots,n$ and the learner makes the hypothesis DFA $M_i$

at the end of eachincrement and ask the equivalence query from the teacher. In case of a negative answer to the equivalence query, the teacher may or may not provide a counterexample. If the teacher provides a counterexample then the learnerextends the learning process on the basis of the received counterexample and incorporates the learning information from previous increment(s) in the new increments. In sequential learning, however, learning is also done in a numberof increments but in each increment the learner starts learning from the scratch and does not use information from the previous increments. In complete learning, the entire system is learnt in a complete fashion to generate a hypothesis.When the whole system (SUL) is learnt by the learner only then it generates a hypothesis DFA, $M$.

According to the existing literature examples of incremental learning algorithms include: IID [22], IDS [18], IKL [19], DKL [17] Kearns [15], TTT [14] and RPNI2 [9] whereas complete learning algorithms are $L^*$ [2], ID [1] and RPNI [8].

The $L^*$ is a complete learning algorithm. It infers a regular language by asking two types of queries; membership and equivalence queries. It poses the membership queries and store the information in the form of a table which is called an *Observation Table* (*OT*). The *OT* should meet two basic properties before asking equivalence queries to make a conjecture. These properties are *closure* and *consistency* [2]. If the observation table is closed but not consistent then columns of the observation table *OT* are extended with a symbol $\sigma$ where $\sigma \in \Sigma$. When the *OT* is *closed* and *consistent*, a conjecture can be constructed. The $L^*$ continues its learning process until *OT* becomes *consistent* and *closed*.

The ID algorithm is a complete learning algorithm [1]. It poses membership queries from the adequate teacher (MAT) to learn the regular set. The concept of MAT was first introduced in this algorithm. It uses the concept of live states, live complete set and dead state $d_0$. The ID algorithm uses the concept of distinguishing strings. To find the blocks of accepting and non-accepting states, the ID algorithm constructs a table. When the first iteration completes, the ID algorithm finds a pair of strings from the live complete set which have the same behavior but for some $\sigma \in \Sigma$ concatenated with both strings results in different behaviors for these strings; as one goes to the accepting block and the other goes to the rejecting block. This gives a potential distinguishing string. If the ID algorithm finds no such pair of strings then it constructs the hypothesis DFA, $\mathcal{H}$ which is isomorphic to the target DFA $\mathcal{A}$.

The RPNI algorithm is a passive learning algorithm [8]. It uses a tree data structure instead of a table for storing information about the hypothesis and does not maintain consistency. It does not use membership queries for learning purpose. It takes two input sets; a set of positive examples and a set of negative examples $S_+$ and $S_-$ respectively. It first writes the elements of $S_+$ and its prefixes in lexicographical order then from the set of positive examples and their prefixes, it constructs the prefix tree $\{PT\}(S_+)$. Then it recursively partitions the branches of the tree into blocks. Initially, each element of $\{PT\}(S_+)$ belongs to its self containing block. The RPNI algorithm recursively applies joint operation on these blocks so that they can be merged into two final blocks. One is the accepting state block and the second is the non-accepting state block.

IID, IDS, RPNI, RPNI2, L*, Kearns, TTT and ID algorithms are 1-bit in nature whereas, IKL, DKL and L* Mealy are k-bit in nature. If we analyze their complexities, we can see that in the presence of an adequate teacher the complexity of these learning algorithms is polynomial (at-least in cubic form) given in Table 3.

The state minimization concept introduced by John Hopcroft, in his algorithm he had used the strategy of making blocks of final and non-final states. With $\delta^-$ transition method, he identified the similar states to generate the minimal target automaton. He claimed that his algorithm takes $nlogn$ complexity for generation of minimal target automaton.

On the basis of the existing literature for automaton learning and state minimization of automata, in this paper we introduce a new efficient DFA learning algorithm DLIQ based on the concepts of the ID algorithm along with inverse transition strategy of John Hopcroft algorithm for state minimization of DFAs. The aim of the current paper is to bring the complexity of DFA learning from polynomial(cubic) to some lower bounds for complete learning of a DFA. A brief summary of existing DFA learning algorithms is given in Table 3 along with the comparison of our proposed algorithm.

**Table 1**

Summary of Existing DFA Learning Algorithms along with Comparison of the Proposed DLIQ Algorithm

| Algorithm | Learning Type | Data Structure | Output Bits | MQ | IQ | Time $O(Queries)$ Complexities |
|---|---|---|---|---|---|---|
| $L^*$ | Complete | Table | 1 | Yes | No | $O(\|\Sigma\|.N^2M)$ |
| ID | Complete | Table | 1 | Yes | No | $O(\|\Sigma\|.\|P\|.N)$ |
| IID | Incremental | Table | 1 | Yes | No | $O(\|\Sigma\|.\|P_l\|.N)$ |
| IDS | Incremental | Table | 1 | Yes | No | $O(\|\Sigma\|.\|P_k\|.N)$ |
| $L^*Mealy$ | Complete | Table | k ≥ 1 | Yes | No | $O(max(N,\Sigma).\|\Sigma\|.NM)$ |
| RPNI | Complete | Tree | 1 | No | No | $O((\|S_p\|+\|S_n\|).\|S_p\|^2)$ |
| RPNII | Incremental | Tree | 1 | No | No | $O((\|S_p\|+\|S_n\|).\|S_p\|^2)$ |
| Kearns | Incremental | Tree | 1 | Yes | No | $O(kN^2+NlogM)$ |
| TTT | Incremental | Tree | 1 | Yes | No | $O(kN^2+NlogM)$ |
| IKL | Incremental | Table | k ≥ 1 | Yes | No | $O(\|\Sigma\|.\|P_k\|.N_l)$ |
| DKL | Incremental | Tree | k ≥ 1 | Yes | No | $O(N.k.\|S_{acc}\|.max(\|S_{acc}\|.\|\Sigma\|,\|i\|))$ |
| DLIQ | Complete | Table | 1 | Yes | Yes | $O(\|\Sigma\|.N+\|P_s\|\|F\|)$ |

The detailed description of the DLIQ algorithm is given in Section 4.

**Table 2**

Description of used Notations

| Notation | Description |
|---|---|
| $P$ | The live complete set |
| $P_s$ | A set consisting of strings in the live complete set and their suffixes |
| $\lambda$ | The empty or null string |
| $F$ | The set of final states |
| $N$ | Number of states in the target |
| $B(num)$ | A block of one or multiple states |
| $k$ | A variable used as a block counter |
| $BlockSet$ | The set of blocks where $BlockSet=\{B(1),B(2),\ldots,B(k)\}$ |
| $B(k)$ | The $k^{th}$ block |
| $num$ | A variable which shows the block number in response to the *BlockQuery* function |
| $B_{num}$ | The List of blocks which is maintained in response to the *BlockQuery* function |
| $B(num')$ | An element of $B_{num}$ containing states which belong to the same block in response to the *BlockQuery* function |

## 4. The Proposed DLIQ Algorithm

The DLIQ is a complete learning algorithm which works on the strategy of $\delta^{-1}$ transitions. It uses a set $P_s$ of *distinguishing strings*. The detailed description of used notations is given in the Table 4.

Initially, the DLIQ algorithm divides the state set into two blocks; the Non-final state block $B(1)$ and the Final state block $B(2)$. The DLIQ algorithm starts learning from the final states on the basis of distinguishing strings and creates / splits blocks into new blocks based on the distinguishing behavior of the predecessor states. In this way, learning is done from final to each live state (till the initial state).

In each iteration, the learner (DLIQ) reads an element of set $P_s$ from a final state set and finds its predecessor state(s). If the predecessor states belong to different blocks then it splits the respective blocks and places the states into new blocks. The learner completes its learning when all elements of set $P_s$ are exhausted.

The proposed DLIQ algorithm is presented below in Algorithms 1 and 2.

**Algorithm 1.** DLIQ Algorithm

1: Input: A Live Complete set $P \subseteq \Sigma^*$ and a DFA $A$ to act as a teacher to answer queries.

2: Output : A DFA $M$ equivalent to the target DFA $A$.

3: **Step 1: if** $P = \varnothing$ **then** $M$ consists of a single non-final state having self transitions. **else**

4: Initially states are divided into two blocks by asking MQ for all $p_i \in P$ as $\delta(p_i, \lambda) = q_i$ where either $q_i \in F$ or not.

// Initiate *BlockSet*

5: B(1) = ($Q$ - $F$) //Non-final states block

6: B(2) = $F$ //Final states block // update *BlockSet*

7: **Step 2**: Make table $\delta^{-1}$ by using final state set $F$ and all their predecessor states, for all input elements $\sigma$ along with empty string $\lambda$, where $\sigma \in \Sigma$

8: **Step 3**: $k = 3$

9: Find suffix set of all $P$ strings as $P_s$

10: **Step 4**:

11: **for all** $p_i \in P_s$ where $p_i \neq \lambda$ **do**

12: **for all** $f_l \in F$ ask inverse query for $p_i$ **do**

13: // finding predecessor state(s) of set $F$ via reading element $p_i$. there are three possibilities against the response of inverse query. "*No*" means no predecessor state, "*Yes*" with one predecessor state or "*Yes*" with multiple predecessor states.

14: **if** $\delta^{-1*}(f_l, p_i) = No = \emptyset$ **then**

15: go to Line 12;

16: **else if** $\delta^{-1*}(f_l, p_i) = Yes = \{q_j\}$ **then**

17: BlockQuery($q_j$, *BlockSet*) = *num* // Block Membership call using Algorithm 2

18: $B(k') = \{q_j\}$

19: **if** $|B(k')| < |B(num)|$ **then**

20: $B(k) = B(k')$

21: $B(num) = B(num) - B(k)$

// update *BlockSet*

22: **end if**

23: **else if** $\delta^{-1*}(f_l, p_i) = Yes = \{q_1, \ldots, q_m\}$ **then**

24: **for** $j = 1$ to $j = m$ **do**

25: BlockQuery($q_j$, *BlockSet*) = *num* // Block Membership call using Algorithm2.

//All predecessor states those belong to the same block are placed in a single block named as $B(num'')$ such as:

$B(num'') = B(num'') \cup \{q_j\}$ where $B(num'')$ belongs to the list $B_{num}$

26: **end for**

27: $B(k') = B(num'')$

28: **if** $|B(k')| < |B(num)|$ **then**

29: $B(k) = B(k')$

30: $B(num) = B(num) - B(k)$

// update *BlockSet*

31: **end if**

32: **end if**

33: **Step 5**

34: **if** $(B(k) \neq \varnothing)$ **then** $k$ ++

35: **if** $(B_{num} \neq \varnothing)$ **then** goto Line 27

36: **end for**

37: **end for**

38: Generate hypothesis automaton $M$ by reading inverse transitions from $\delta^{-1}$ table.

**Algorithm 2.** Block Membership

1: Input: A state $q_j$ where $q_j \in Q$ and a set of existing blocks *BlockSet*.

2: Output: Block number named as *num*, from where $q_j$ belongs to.

3: **Function** BlockQuery($q_j$, *BlockSet*)

4: {

5: **return** *num*

6: }

## 5. Correctness and Termination

The correctness of the algorithm is based on the fact that it correctly produces a learned DFA consistent with the target DFA $A$.

**Conjecture 5.1.** *Splitting of a block $B(num)$ as $B(a)$ and $B(b)$ is done on the basis of mutually exclusive $B(a) \cap B(b) = \varnothing$ and completely exhaustive $B(a)$ $B(b) = B(num)$ property.*

**Lemma 5.2.** *Number of blocks $|BlockSet|$ can not exceed the number of states.*

*Proof.* Since only two blocks are created initially. One consists of only non-final states $B(1)$ and other consists of only final states $B(2)$. When blocks are split then the DLIQ algorithm (line 4 and 4) makes sure that blocks meet mutually exclusive $B(1) \cap B(2) = \varnothing$ and completely exhaustive $B(1) \cup B(2) = Q$ property. Therefore, no state can be in more than one block and every state must reside in some block. Therefore, even in the case when every state resides in a separate block, the number of blocks can not be greater than the number of states.

**Theorem 5.3.** *Let $P_s = \{p_1, p_2, \ldots, p_n\}$ is a set consisting of strings in the live complete set for a target DFA $A$ and their suffixes. Then the blocks created during learning are distinguishable on the elements of $P_s$ read so far and are mutually exclusive in nature.*

*Proof.* The main task is to show that $\delta^{-1}$ transition is a well defined relation. As it takes two arguments; the state and an element $p_i \in P_s$ as a distinguishing string to find the distinguishing behavior of states and return predecessor state(s) via reading $p_i \in P_s$. As $\delta^{-1}(q_j, p_i) = q_{j-1}$ or $q_n$. Since the predecessor states: initially belonging to the same block are split and are placed in newly created blocks during learning. These blocks are distinguishable on the element $p_i$. Therefore, first part of the theorem is proved.

Further, it follows from Conjecture 5.1 that since blocks are always created based on mutually exclusive $B(a) \cap B(b) = \varnothing$ property hence, it suffices to establish that during partition refinement newly created blocks will be mutually exclusive in nature.

**Corollary 5.4.** *Let $P_s = \{p_1, p_2, \ldots, p_n\}$ be a non-empty set consisting of strings in the live complete set and their suffixes and $1 \le i \le n$. The execution of DLIQ on $P_s$ terminates with the program variable $i$ having value $n$.*

*Proof.* It is obvious that the outer for loop of Algorithm 4 terminates when the set $P_s$ having $|P_s| = n$ is exhausted.

**Theorem 5.5 (Theorem).** Let $P_s = \{p_1, p_2, \ldots, p_n\}$ be a set consisting of strings in the live complete set for a target DFA $A$ and their suffixes. The DLIQ algorithm terminates on $P_s$ and have $k$ distinct blocks at the end of the execution.

*Proof.* (a) By Corollary 5.4, DLIQ terminates on $P_s$ with the program variable $i$ having value *n*. (b) The number of distinct blocks created during learning by reading some element $p_i \in P_s$, can be shown to vary between 1 and $m$; $1 \le k \le m$ where $m \le |Q|$ by induction on $P_s$.

**Theorem 5.6 (Theorem).** *The DLIQ algorithm terminates on $P_s$ and the hypothesis automaton $M$ is a canonical representation of $A$.*

*Proof.* As the DLIQ algorithm poses inverse and equivalence queries to get the information about the grammar of the target DFA $A$. It is important to note that DLIQ specifically learns through inverse queries. We use the following two premises to prove this theorem: (a) The Termination Theorem 5.5 establishes that learning terminates when all elements of the set $P_s$ are exhausted. (b) The DLIQ algorithm merges all the states of the target automaton $A$ which are equivalent in behavior into a single state block and the hypothesis automaton $M$ is then constructed using all the blocks created as states of the learned hypothesis. Therefore, the hypothesis automaton $M$ is a unique minimal representation of $A$. Combining (a) and (b) prove the theorem.

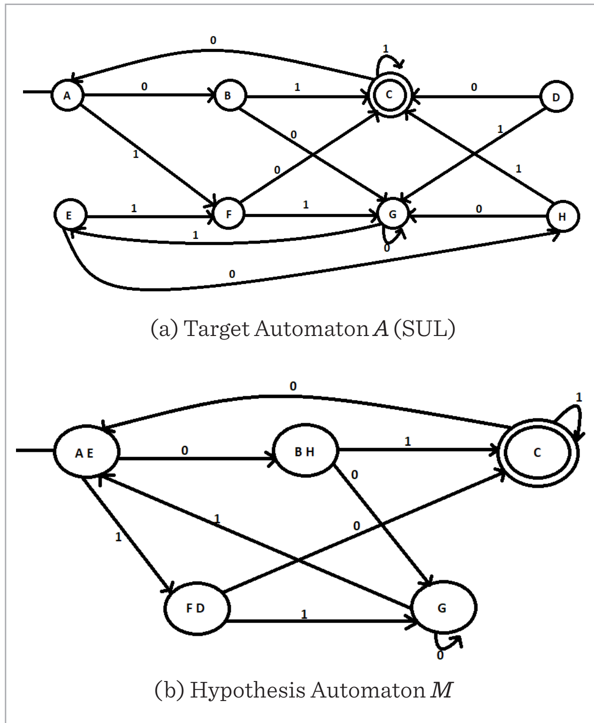
## 6. An Example

Now we illustrate the working of the DLIQ algorithm on an example automaton given in Figure 1a. The in-

puts of the algorithm consist of the Target Automaton ($A$) (given in Figure 1a), the live complete set $P = \{\lambda, 0,1,01,11,111,1110\}$ of the target automaton $A$. Initially states are partitioned into two main blocks by asking the *membership query* (MQ) $\delta(p_i, \lambda)$ =? where $p_i \in P$ of the target automaton $A$. Now states divided into two blocks are: $B(1)$ = {A, B, D, E, F, G, H} and $B(2)$ = {C}.

**Figure 1**

Target Automaton Before and After Learning




(a) Target Automaton $A$ (SUL)

(b) Hypothesis Automaton $M$

According to the step 2, the algorithm finds the predecessor(s) of all the final states and their predecessor states (Table 3) by asking $\delta^{-1}$ queries from the teacher for all elements of $\Sigma$ along with empty string $\lambda$ (according to line 7 of the Algorithm 1).

Initially $k = 3$ and $P_s = \{\lambda, 0, 1, 01, 11, 10, 111, 110, 1110\}$. According to the step 4, $p_i$ = 0. Predecessor state(s) of $b_l = \{C\}$ via reading 0 are F and D those belong to block $B(1)$. Therefore, $B(1'')$ = { F, D } and now $B(k') = B(3')$ = { F, D }. As $|B(3')| < |B(1)|$ therefore, $B(3)$ = { F, D } and $B(1)$ = { A, B, E, G, H }. Now according to step 5, we find $B(3) \neq \varnothing$, therefore, $k$ = 4 and $B_{num} = \varnothing$. Since, $B(2)$ has no more elements therefore, inner for loop ends. Now, $p_i$ = 1. The predecessor states of C via reading 1 are found to be B, C, H. Both B and H belong to block $B(1)$ and C belongs to the block $B(2)$ therefore, $B(1'')$ = { B, H } and $B(2'')$ = {C}. We get, $B(k') = B(4')$ = {B, H} as $|B(4')| < |B(1)|$, therefore, $B(4)$ = {B, H} and $B(1)$ = {A, E, G}. Now we go to step 5 and find $B(4) \neq \varnothing$, therefore $k$ = 5 and $B_{num} \neq \varnothing$ therefore, goto the line 27 of Algorithm 1. We get, $B(k') = B(5')$ = {C} as $|B(5')| = |B(2)|$, therefore, if condition is false. Now control again go to the step 5. As $B(5) = \varnothing$ therefore, value of $k$ remains the same as $k$ = 5. Since, $B(2)$ has no more elements therefore, inner for loop ends. The next $p_i$ = 01. The predecessor states of C via reading 01 are A, F, D, E. A and E belong to $B(1)$ and F, D belong to $B(3)$ therefore $B(1'')$ = { A, E } and $B(3'')$ = {F, D}. According to line 27, $B(k') = B(5')$ = {A, E} as $|B(5')| < |B(1)|$ so, $B(5)$ = {A, E} and $B(1)$ = {G}.

Again go to step 5. As $B(5) \neq \varnothing$, therefore, $k$ = 6. As $B_{num} \neq \varnothing$ therefore, goto the line 27. We get, $B(k') = B(6')$ = {F, D} as $|B(6')| = |B(3)|$, therefore, if condition is false. Now control again go to the step 5. As $B(6) = \varnothing$ therefore, value of $k$ remains the same as $k$ = 6. Similarly, algorithm will be executed for all $p_i$ elements one by one and split the blocks. When the set $P_s$ is completely exhausted, the algorithm terminates and complete its learning. After termination, the final blocks will be: {{C}, {B, H}, {A, E}, {D, F} and {G}}.

***Automaton Construction:*** The final number of blocks show the number of states. Each block represents a single state. By reading the $\delta^{-1}$ table input transitions, algorithm will construct the hypothesis automaton $M$ (described in Figure 1(b)).

**Table 3**

$\delta^{-1}$ Table

| States/Inputs | $\lambda$ | 0 | 1 |
|---|---|---|---|
| C | ∅ | F, D | B, C, H |
| F | ∅ | ∅ | A, E |
| D | ∅ | ∅ | ∅ |
| B | ∅ | A | ∅ |
| H | ∅ | E | ∅ |
| A | A | C | ∅ |
| E | ∅ | ∅ | G |
| G | ∅ | B, H, G | F, D |

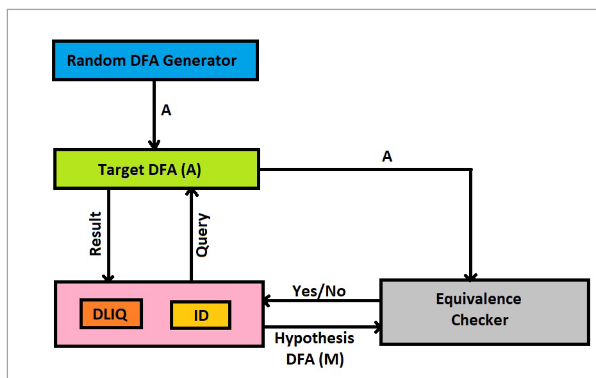
## 7. Comparison of DLIQ and ID Algorithm

For comparison of performance of DLIQ with other related algorithms we considered the algorithms given in Table 1 which summarizes different types of automaton learning algorithms available in the literature based on different parameters. Like the DLIQ algorithm proposed in this paper only three others are complete learning algorithms which include: the L*, the ID,the RPNI.The L*, the ID and the DLIQ are based on table data structure while the RPNI is based on tree data structure therefore it cannot be used for comparison with DLIQ. Therefore, for comparison of DLIQ only the L* and/or the ID can be used. However, one significant difference in the design of the L* is its requirement to make use of a counterexample whenever the hypothesis built by it is not equivalent to the target automaton. Since this characteristic is lacked by the ID and the DLIQ algorithm plus it makes the learning more directed as compared to ID and DLIQ therefore, we were left with only the ID algorithm to perform comparison with the DLIQ algorithm as both share almost all parameters of comparison except the inverse query.

For the comparison of DLIQ and ID algorithms we have setup an evaluation framework (given in Figure 2). Thisframework consists of following modules:

1. A target DFA (A)
2. A random DFA generator
3. DLIQ and ID algorithms
4. An automaton equivalence checker

**Figure 2**
Evaluation Framework




### 7.1. Experimental Setup

We have implemented the DLIQ and ID algorithms in Java. For execution of experiments we have used a PC having Windows 8.1 pro, 16GB RAM and Intel core i5-3470 processor. We have performed multiple experiments on both these algorithms. The experiments varied due to two main parameters of the target DFA $A$. These parameters include:

1. The state size $|Q|$
2. The input alphabet size $|\Sigma|$

For performance analysis of both these algorithms, we setup the experiments to be executed with state size $|Q|$ varying between $10, 20, 30, \ldots, 100$ and alphabet size varying between $2, 4, \ldots, 10$. The target DFAs were randomly generated with all possible combinations of both the parameters such as for ($|Q|$=10, $|\Sigma|$=2), ($|Q|$=10, $|\Sigma|$=4), ($|Q|$=10, $|\Sigma|$ =6), ($|Q|$=10, $|\Sigma|$=8), ($|Q|$=10, $|\Sigma|$=10). We also generated their respective live complete sets. We adopted the strategy of *testing tree generation* for FSMs as described in [16] for generating the live complete sets of randomly generated DFAs. For this, we start from the initial state and make it the root node of the tree and store it in a Hash Set. Then we find the next state for all the elements of set $\Sigma$ and if a new state is not stored in the Hash Set is found then we create a branch of the tree from the current node to that state. This process continues until no new state is found. Then we create strings that take us to the final states. This is the whole procedure that we followed to generate respective live complete set of a randomly generated DFA. Furthermore, for a particular parameter configuration we repeated the experiments 10 times and then compiled the results to compute the mean of 10 experiments.

### 7.2. Empirical Evaluation

For comparison of both these algorithms (DLIQ and ID), we have considered the following two parameters:

1. Number of queries (these included the queries posed by the learner to the MAT);
2. The learning time (ms).

#### 7.2.1. Number of Queries

In the case of the DLIQ algorithm, the learner(DLIQ) poses Inverse Queries (IQs) to learn a minimal target

DFA whereas, for the ID algorithm, the learner (ID) poses Membership Queries (MQs) for learning purpose.

### 7.2.2. The Learning Time

In order to analyze the performance of both these algorithms with respect to time complexity, we have considered only the learning time which a learner takes to learn the target DFA. We are not interested to compare the time taken by the equivalence checker to check the behavioral equivalence of hypothesis automaton with target automaton.

## 7.3. Results and Analysis

The computed results are given in Table 4 and Table 5. All the results in the tables and the graphs depict that when the number of states or input alphabet size increases, there is a significant rise in the number of queries and learning time of the ID algorithm whereas with the increase in states or input alphabet size, the graphs of DLIQ algorithm grow relatively slowly. Therefore, it is obvious that the DLIQ algorithm is more efficient than the ID algorithm in terms of time and number of queries posed to the MAT.

**Table 4**

Query wise Comparison of ID and DLIQ Algorithm

| No. of States | ID | | | | | DLIQ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | \|Σ\|=2 | \|Σ\|=4 | \|Σ\|=6 | \|Σ\|=8 | \|Σ\|=10 | \|Σ\|=2 | \|Σ\|=4 | \|Σ\|=6 | \|Σ\|=8 | \|Σ\|=10 |
| 10 | 120.3 | 280.1 | 661.7 | 720.6 | 800.0 | 28.6 | 76.4 | 89.9 | 96.8 | 132.7 |
| 20 | 160.4 | 720.7 | 840.3 | 1440.3 | 2200.2 | 58.5 | 161.2 | 160.8 | 196.0 | 288.3 |
| 30 | 660.8 | 960.4 | 2340.8 | 2400.8 | 3900.5 | 158.9 | 136.4 | 285.6 | 310.4 | 391.0 |
| 40 | 1120.0 | 3040.1 | 2880.3 | 4160.3 | 7200.1 | 206.4 | 252.9 | 338.1 | 463.2 | 544.1 |
| 50 | 1900.3 | 7000.0 | 9300.1 | 13600.2 | 19500.1 | 480.2 | 725.0 | 610.1 | 706.3 | 773.0 |
| 60 | 3480.2 | 7440.2 | 10080.0 | 16320.2 | 22800.1 | 336.0 | 457.9 | 472.3 | 865.1 | 866.6 |
| 70 | 5460.2 | 11480.2 | 16389.3 | 23520.0 | 30100.2 | 569.2 | 616.0 | 732.9 | 870.2 | 1150.4 |
| 80 | 6880.2 | 13120.6 | 21221.4 | 24960.1 | 32800.1 | 547.2 | 656.3 | 700.3 | 901.2 | 1292.3 |
| 90 | 7920.3 | 16920.4 | 24840.3 | 30960.7 | 42300.1 | 840.1 | 971.0 | 1184.2 | 1451.3 | 1276.3 |
| 100 | 9800.2 | 21200.3 | 30600.2 | 43200.1 | 55000.3 | 1033.0 | 1407.3 | 1467.2 | 1772.3 | 1770.5 |

**Figure 3**

Comparison of DLIQ and ID Algorithm with respect to Queries

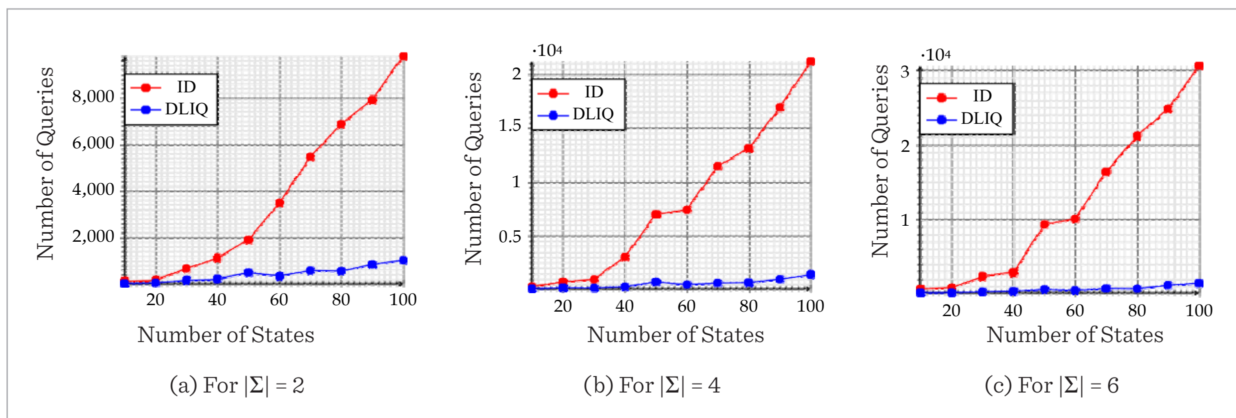


**Table 5**
Time (ms) wise Comparison of ID and DLIQ Algorithm

| No. of States | ID | | | | | DLIQ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | \|Σ\|=2 | \|Σ\|=4 | \|Σ\|=6 | \|Σ\|=8 | \|Σ\|=10 | \|Σ\|=2 | \|Σ\|=4 | \|Σ\|=6 | \|Σ\|=8 | \|Σ\|=10 |
| 10 | 8.3 | 10.1 | 19.1 | 22.3 | 24.8 | 6.3 | 7.2 | 12.4 | 13.8 | 13.9 |
| 20 | 11.0 | 22.3 | 23.3 | 27.6 | 72.1 | 7.2 | 10.3 | 12.7 | 15.6 | 28.7 |
| 30 | 58.1 | 83.0 | 113.0 | 120.0 | 138.2 | 20.0 | 17.8 | 38.8 | 38.9 | 42.3 |
| 40 | 67.2 | 173.0 | 169.7 | 189.2 | 481.5 | 29.1 | 31.9 | 41.0 | 44.3 | 49.7 |
| 50 | 81.6 | 341.7 | 411.8 | 512.3 | 778.4 | 34.3 | 51.0 | 47.9 | 50.2 | 58.0 |
| 60 | 93.2 | 398.2 | 439.1 | 823.4 | 998.3 | 36.2 | 51.2 | 51.3 | 55.1 | 61.3 |
| 70 | 121.4 | 437.9 | 632.4 | 1001.0 | 1102.5 | 39.8 | 57.3 | 63.2 | 65.2 | 69.8 |
| 80 | 156.2 | 510.2 | 691.2 | 1035.6 | 1214.3 | 42.7 | 61.9 | 71.7 | 74.3 | 79.2 |
| 90 | 181.4 | 578.0 | 783.0 | 1105.5 | 1538.4 | 47.9 | 64.0 | 78.1 | 83.1 | 87.9 |
| 100 | 210.3 | 631.2 | 987.4 | 1550.0 | 2053.6 | 55.3 | 67.3 | 81.2 | 92.4 | 113.4 |

**Figure 4**
Comparison of DLIQ and ID Algorithm with respect to Queries

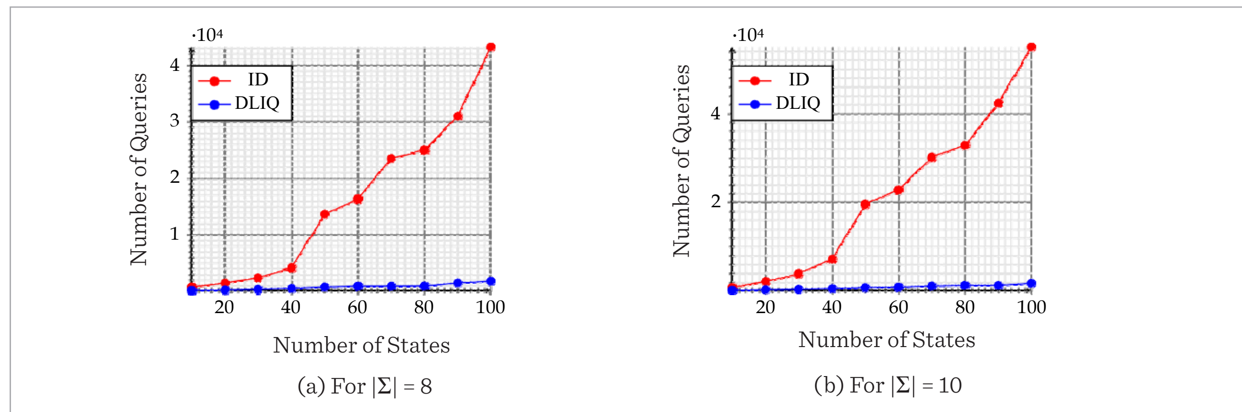



**Figure 5**
Comparison of DLIQ and ID Algorithm with respect to Time Efficiency

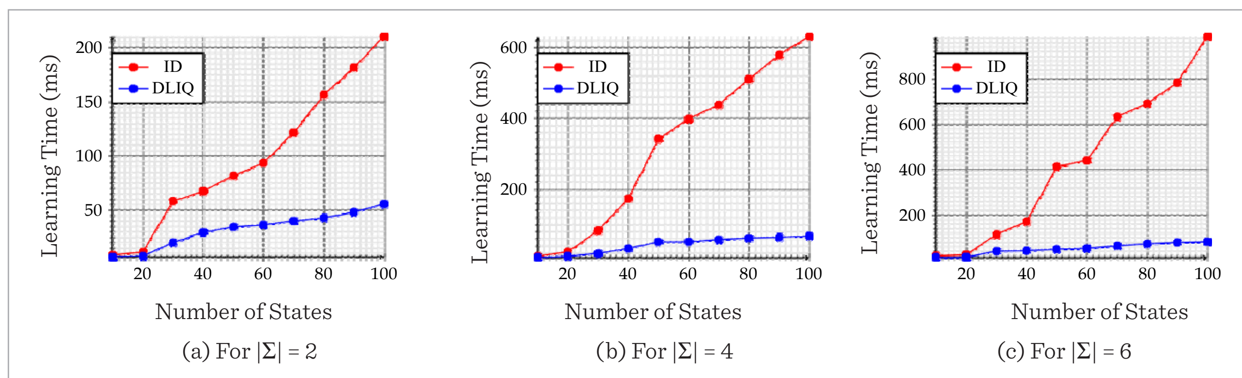
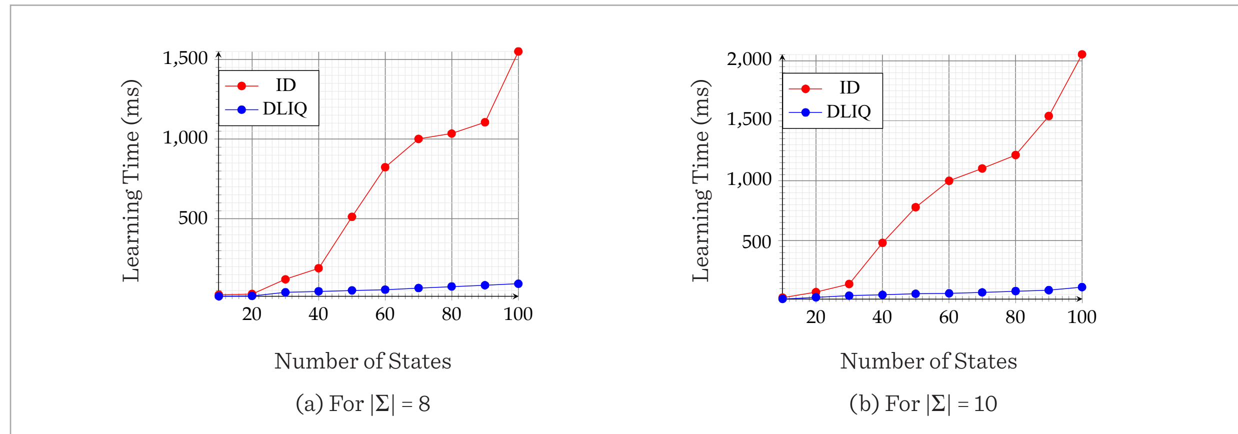


**Figure 6**

Comparison of DLIQ and ID Algorithm with respect to Time Efficiency




(a) For |Σ| = 8 (b) For |Σ| = 10

## 8. Complexity Analysis

The minimally adequate teacher (MAT) has polynomial time complexity but its complexity is assumed as constant for the query complexity analysis [2]. The MAT possesses $O(1)$ time complexity for a membership and $\delta^{-1}$ query whereas it has polynomial time complexity for an equivalence query.

Table construction takes $O(|\Sigma|N+1)$ time complexity. In each iteration, the learner asks $\delta^{-1}$ query against an element of set $P_s$ (used as distinguishing string in the proposed algorithm) therefore, its time complexity is $O(|P_s|)$ whereas, the number of states visited at each step are maximum of $|F|$ (which is in most of the cases $<<N$) therefore, the nested loop takes $O(|P_s||F|)$ time complexity. The block splitting process takes $O(1)$ time complexity. Therefore, the query based time complexity of the proposed algorithm is $O(|P_s||F|+|\Sigma|N)$. Automaton reconstruction is done by reading the $\delta^{-1}$ table therefore, it also takes $O(|\Sigma|N)$ time complexity. In view of above asymptotic analysis, total worst case time complexity of the algorithm is $O(|P_s||F|+|\Sigma|N)$.

If we carefully analyze this complexity, we can see that the dominant factor involved in this complexity is $|P_s||F|$ where $|P_s|\approx N$ and in most of the cases $F<<N$ (almost constant). The other factor that is $|\Sigma|.N$; in most of the time $|\Sigma|$ also act as nearly constant therefore, the average case complexity of the DLIQ algorithm is nearly linear. For this, we can claim that in this paper, we have reduced the time complexity of the DFA learning.

## 9. Conclusions and Future Directions

We have reduced the complexity of the deterministic finite automaton (DFA) learning (from cubic to square form in worst case) and presented a novel complete learning algorithm DLIQ for learning of a DFA. We have also shown experimentally that the DLIQ is more efficient than the ID algorithm in terms of time complexity. This algorithm improves the complete learning process of DFAs by making use of $\delta^{-1}$ queries. In the future, we plan to extend the concept used for 1-bit learning to $k$-bit learning for reducing the complexity of $k$-bit learning algorithms also. Another direction of work can be to design incremental automaton learning algorithms both for the 1-bit and $k$-bit scenarios.

### Acknowledgement

We gratefully acknowledge financial support for this research from Higher Education Commission of Pakistan (HEC) under grant no: 9223/Federal/NRPU/RD/HEC/2017.

## References


1. Angluin, D. A Note on the Number of Queries Needed to Identify Regular Languages. Information and Control, 1981, 51, 76-87. https://doi.org/10.1016/S0019-9958(81)90090-5
2. Angluin, D. Learning Regular Sets from Queries and Counterexamples. Information and Computation, 1987, 75, 87-106. https://doi.org/10.1016/0890-5401(87)90052-6
3. Angluin, D., Chen, D. Learning a Random DFA from Uniform Strings and State Information. In: Chaudhuri, K., GENTILE, C., Zilles, S. (eds) Algorithmic Learning Theory. Lecture Notes in Computer Science, 2015, 9355. Springer, Cham. https://doi.org/10.1007/978-3-319-24486-0_8
4. Boehm, B. W. Software Engineering: R&D Trends and Defense Needs. Research Directions in Software Technology, 1978.
5. Cleaveland R., J. P., Steffen, B. A Semantics-Based Verification Tool for Finite State Systems in Proceedings of the Ninth IFIP Symposium on Protocol Specification, Testing and Verification, North Holland, 1989.
6. Cleaveland R., J. P., Steffen, B. The Concurrency Workbench in Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems, LNCS, 1989. https://doi.org/10.1007/3-540-52148-8_3
7. Cleaveland R., J. P., Steffen, B. The Concurrency Workbench: Operating Instructions. University of Edinburgh, Laboratory for Foundations of Computer Science, Technical Note 10, 1988.
8. Departarnento, J. N., Garcia, P. Identifying Regular Languages In Polynomial in Advances in Structural and Syntactic Pattern Recognition. Series in Machine Perception and Artificial Intelligence, 1992, 5, 99-108. https://doi.org/10.1142/9789812797919_0007
9. Dupont, P. Incremental Regular Inference. Proceedings of the Third ICGI-96, 1996. https://doi.org/10.1007/BFb0033357
10. Gold, E. M. Language Identification in the Limit. Information and Control, 1967, 10, 447-474. https://doi.org/10.1016/S0019-9958(67)91165-5
11. Groce, A., Peled, D., Yannakakis, M. Adaptive Model Checking. Logic Journal of the IGPL, 2006, 14, 729-744. https://doi.org/10.1093/jigpal/jzl007
12. Hessel, A. et al. Testing Real-Time Systems Using UPPAAL. Formal Methods and Testing, 2008, 77-117. https://doi.org/10.1007/978-3-540-78917-8_3
13. Hopcroft, J. An n Log n Algorithm for Minimizing States in a Finite Automaton. Theory of Machines and Computations, 1971, 189-196. https://doi.org/10.1016/B978-0-12-417750-5.50022-1
14. Isberner, M., Howar, F., Steffen, B. The TTT Algorithm: A Redundancy-free Approach to Active Automata Learning in International Conference on Runtime Verification, 2014, 307-322. https://doi.org/10.1007/978-3-319-11164-3_26
15. Kearns, M. J., Vazirani, U. V., Vazirani, U. An Introduction to Computational Learning Theory, MIT Press, Cambridge, Massachusetts, 1994. https://doi.org/10.7551/mitpress/3897.001.0001
16. Mathur, A. P. Foundations of Software Testing 2E. Dorling Kindersley (India) Pvt. Ltd, 2013.
17. Mazhar, R., Sindhu, M. A. DKL: An Efficient Algorithm for Learning Deterministic Kripke Structures. Acta Informatica, 2020. https://doi.org/10.1007/s00236-020-00387-2
18. Meinke, K., Sindhu, M. A. Correctness and Performance of an Incremental Learning Algorithm for Kripke Structures. Technical Report, School of Computer Science and Communication, Royal Institute of Technology, Stockholm, 2010.
19. Meinke, K., Sindhu, M. A. Incremental Learning-Based Testing for Reactive Systems. Tests and Proofs (eds Gogolla, M. & Wolff, B.), 2011, 6706, 134-151. ISBN: 978-3-642-21767-8. https://doi.org/10.1007/978-3-642-21768-5_11
20. Michaliszyn, J. O. J. Learning Deterministic Automata on Infinite Words. ECAI 2020, 2020.
21. Michaliszyn, J. O. J. Learning Infinite-Word Automata with Loop Index Queries. Artificial Intelligence, 2022, 1(307), 103710. https://doi.org/10.1016/j.artint.2022.103710
22. Parekh, R., Nichitiu, C., Honavar, V. A Polynomial Time Incremental Algorithm for Regular Grammar Inference. Proceedings of Fourth International Colloquium on Grammatical Inference (ICGI 98) (Springer, 1998). https://doi.org/10.1007/BFb0054062
23. Parrow, J. Verifying a CSMA/CD Protocol with CCS in Proceedings of the Seventh IFIP Symposium on Protocol Specification, Testing and Verification (1987).
24. Peled, D., Vardi, M. Y., Yannakakis, M. Black Box Checking in FORTE, 1999, 225-240. https://doi.org/10.1007/978-0-387-35578-8_13

25. Pellegrino G., H. C.L. Q., Verwer, S. Learning Deterministic Finite Automata from Infinite Alphabets. International Conference on Grammatical Inference, 2017, 120-131.
26. Petrenko, A., Avellaneda, F., Groz, R., Oriat, C. FSM Inference and Checking Sequence Construction are Two Sides of the Same Coin. Software Quality Journal, 2018, 1-24. https://doi.org/10.1007/s11219-018-9429-3
27. Seijas, P. L., Thompson, S., Francisco, M. Á. Model Extraction and Test Generation from JUnit Test Suites. Software Quality Journal, 2018, 26, 1519-1552. https://doi.org/10.1007/s11219-017-9399-x
28. Sheinvald, S. Learning Deterministic Variable Automata over Infinite Alphabets. International Symposium on Formal Methods, 2019, 633- 650. https://doi.org/10.1007/978-3-030-30942-8_37
29. Smetsers, R., Volpato, M., Vaandrager, F., Verwer, S. Bigger is Not Always Better: On the Quality of Hypotheses in Active Automata Learning. International Conference on Grammatical Inference, 2014, 167-181.
30. Smetsers, R., Fiterau-Brostean, P., Vaandrager, F.W. Model Learning as a Satisfiability Modulo Theories Problem. In S.T. Klein et al. (eds.) Proceedings 12th International Conference on Language and Automata Theory and Applications (LATA), Bar-Ilan, Israel, April 9-11, 2018, LNCS 10792, 182-194, 2018. https://doi.org/10.1007/978-3-319-77313-1_14
31. Sofia, C., Howar, F., Jonsson, B., Steffen, B. Active learning for extended finite state machines. Formal Aspects of Computing, 2016, 28(2), 233-263. https://doi.org/10.1007/s00165-016-0355-5
32. Therese, B., Jonsson, B., Leucker, M., Saksena, M. Insights to Angluin's learning. Electronic Notes in Theoretical Computer Science, 2005, 118, 3-18. https://doi.org/10.1016/j.entcs.2004.12.015
33. Vaandrager, F. Model learning. Communications of the ACM, 2017, 60, 86-95. https://doi.org/10.1145/2967606
34. Walker, D. Analysing Mutual Exclusion Algorithms Using CCS. University of Edinburgh Technical Report ECS-LFCS-88-45, 1988.

This article is an Open Access article distributed under the terms and conditions of the Creative Commons Attribution 4.0 (CC BY 4.0) License (http://creativecommons.org/licenses/by/4.0/).