Finding All Minimum-Size DFA Consistent with Given Examples: SAT-Based Approach

Ilya Zakirzyanov[1,2(✉)], Anatoly Shalyto[1], and Vladimir Ulyantsev[1]

[1] ITMO University, Saint Petersburg, Russia
{zakirzyanov,ulyantsev}@rain.ifmo.ru, shalyto@mail.ifmo.ru
[2] JetBrains Research, Saint Petersburg, Russia

Abstract. Deterministic finite automaton (DFA) is a fundamental concept in the theory of computation. The NP-hard DFA identification problem can be efficiently solved by translation to the Boolean satisfiability problem (SAT). Previously we developed a technique to reduce the problem search space by enforcing DFA states to be enumerated in breadth-first search (BFS) order. We proposed symmetry breaking predicates, which can be added to Boolean formulae representing various automata identification problems. In this paper we continue the study of SAT-based approaches. First, we propose new predicates based on depth-first search order. Second, we present three methods to identify all non-isomorphic automata of the minimum size instead of just one—the $\#P$-complete problem which has not been solved before. Third, we revisited our implementation of the BFS-based approach and conducted new evaluation experiments. It occurs that BFS-based approach outperforms all other exact algorithms for DFA identification and can be effectively applied for finding all solutions of the problem.

Keywords: Grammatical inference · Automata identification
Symmetry breaking · Boolean satisfiability

1 Introduction

A variety of models exists in automata theory but a *deterministic finite automaton* (DFA) is the basic one and among the most important ones. DFA is a model that recognizes regular languages [1]. The essence of the DFA identification (induction, learning, synthesis) problem is to find a minimum-size DFA (a DFA with the minimum number of states) that is consistent with a given set of labeled examples—positive-labeled strings that must be accepted by the built DFA and negative-labeled strings that must be rejected. A smaller DFA is simpler and, because of well-known Occam's razor principle, it is a model which better explains the observed examples. Thus the DFA learning problem is to find the regular language that most likely was used to generate a set of labeled examples. This problem is among the best-explored ones in grammatical inference [2].

A. Cerone and M. Roveri (Eds.): SEFM 2017 Workshops, LNCS 10729, pp. 117–131, 2018.
https://doi.org/10.1007/978-3-319-74781-1_9

This problem was shown to be NP-hard in [3]. Nevertheless, several efficient DFA learning approaches were developed, see, e.g., [2]. DFA identification using evolutionary computation methods is one of historically the first and effective approaches, see, e.g., [4,5]. Subsequent research resulted in development of a method for evolving DFA using a multi-start random hill climber, see, e.g., [6].

Later approaches are based on heuristic algorithms. The *evidence driven state-merging* algorithm (EDSM) is the most commonly used and the only one which can handle large-sized target DFA [7]. This algorithm is greedy and works in polynomial time. Despite its efficiency in terms of solving time this approach usually finds only a local optimum but not a global one. The performance of EDSM was several times improved by using specialized search procedures, see, e.g., [8,9]. In [6] Lucas and Reynolds compared the EDSM algorithm and the evolutionary algorithm (EA) mentioned above. They found that the EDSM-based approach outperforms the EA in terms of solving time on almost all instances.

The methods mentioned above are not exact—they cannot guarantee that the found DFA is one of the minimum-sized ones. Heule and Verwer proposed so-called *translation-to-SAT* approach which can be applied to DFA identification [10]. This approach, as it can be obtained from the name, is based on the translation the original problem to well-studied *Boolean satisfiability problem* (SAT). The performance of SAT solvers has significantly improved over the last decade. This computational strength can be used in other problems by *translating* these problems into SAT instances, and subsequently running a modern SAT solver on them. This approach was shown to be very competitive for some problems, see, e.g., [11–14]. The authors have shown that translation-to-SAT is effective for solving DFA identification as well. The SAT-based method is exact as opposed to EA and EDSM algorithms, which is important because of the mentioned Occam's razor principle. The authors also proposed a combined approach, which used a few EDSM steps as a preprocessing step, and won the first prize at the *StaMInA* competition [15]. We do not consider of this step in our paper because EDSM is not an exact algorithm.

There are *symmetries* in many combinatorial problems. *Symmetry breaking predicates* can be added as constraints to SAT formula with purpose of elimination some or all symmetries and thus reduce the search space, see, e.g., [16]. When we talk about DFA the most obvious symmetries are groups of isomorphic automata. Heule and Verwer in [10] proposed simple but effective greedy *maximal clique* (max-clique) algorithm. It allows reducing the amount of isomorphic automata in each group from $n!$ to $(n-k)!$, where n is the size of the DFA and k is the size of the found clique. We proposed symmetry breaking predicates which enforce DFA states to be enumerated in the breadth-first search (BFS) order in [17]. These predicates can be added to a Boolean formula before passing it to a SAT solver. This approach allows to reduce the amount of isomorphic automata in each group from $n!$ to only one representative—the BFS-enumerated one. The results for the exact case still were not very good in our previous paper. However, the BFS-based approach is more flexible than max-clique—we demonstrated its flexibility by developing a modification of the noiseless translation-to-SAT technique for the noisy case (some examples are wrong-labeled).

In this paper we propose new symmetry breaking predicates based on depth-first search (DFS) order. This is the modification of our previous BFS-based approach. BFS-based predicates were not good enough to compete with the max-clique algorithm in DFA identification in our previous paper. Therefore we revisited our implementation of this technique. It occurs that both BFS-based and DFS-based approaches clearly outperform current state-of-the-art DFASAT from [10]. We also propose a method based on these techniques for solving the *problem of finding all automata* (find-all) with the minimum number of states which are consistent with a given set of examples. This problem has not been solved efficiently before. Moreover, none of the existing approaches for the DFA learning are applicable, even with slight modifications, to solve the find-all problem due to their nature. We use two ways of launching SAT solvers: relaunching a non-incremental solver and using an incremental solver. We also developed the heuristic backtracking method (almost similar to the one presented in the paper [18]) as a baseline for comparing it with SAT-based ones.

2 Preliminaries and Previous Work

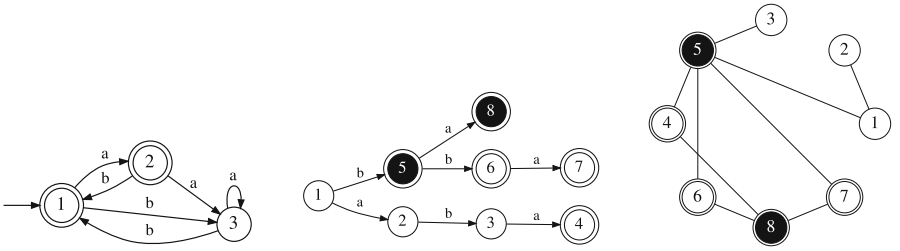
2.1 Encoding DFA Identification into SAT

We assume the reader to be familiar with the theory of languages and automata. The purpose of the DFA identification problem is to find the minimum DFA which is consistent with two given sets of strings: a set of positive examples (S_+) and is a set of negative examples (S_-). In other words, the desired DFA must accept all strings from S_+ and reject all strings from S_-. In this paper it is assumed that DFA states are numbered from 1 to C and the start state has number 1. The example of the minimum DFA for $S_+ = \{aba, bb, bba\}$ and $S_- = \{b, ba\}$ is shown in Fig. 1a.

We briefly describe the current state-of-the-art approach for solving the considered problem. The first step of the technique proposed by Heule and Verwer in [10] is to build an *augmented prefix tree acceptor* (APTA) from the given sets S_+ and S_-. An APTA is a tree-shaped automaton based on a prefix tree for the sets S_+ and S_- but with labeled states. It is called augmented because it may contain states which are not accepting or rejecting. The APTA for S_+ and S_- mentioned above is shown in Fig. 1b.

The second step is to construct the *consistency graph* (CG) for the built APTA. The set of the CG vertices is the same as the APTA vertices set. Two vertices in the CG are adjacent if their merging in the APTA and subsequent determinization process will cause an inconsistency: a situation when an accepting state is merged with a rejecting one. The CG for APTA from Fig. 1b is shown in Fig. 1c.

The third step of the method is to divide the CG vertices set into C disjoint sets. Each set has to contain all vertices equivalent to the corresponding APTA states which will be merged into one state in the resulting DFA. If such a separation can be made, then the automaton with C states consistent with the given sets of strings exists and it can be easily built. C can be iterated from 1 and until



(a) An example of a DFA

(b) An example of an APTA for $S_+ = \{aba, bb, bba\}$ and $S_- = \{b, ba\}$

(c) The consistency graph for the APTA from Fig. 1b

Fig. 1. An example of an APTA and its consistency graph

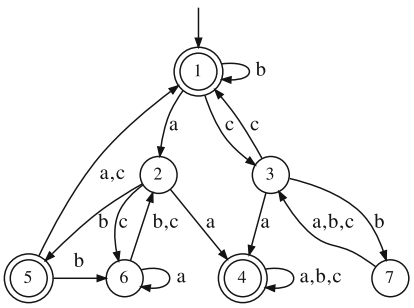
such a partition is found. Thus it is guaranteed that the found C-sized DFA is the minimum DFA consistent with given behavior examples. This can be viewed as a *graph coloring* problem and we need to color CG vertices into the minimum number of colors in such a way that adjacent vertices have different colors.

The next step in the considered algorithm is to translate the graph coloring problem into SAT. Authors proposed so-called *compact encoding* where they use three kinds of Boolean variables to formulate all constraints in CNF: *color variables* $x_{v,i}$ which indicate whether the vertex v in the CG is i-colored; *parent relation variables* $y_{a,i,j}$ which indicate whether there is an a-labeled transition from the i-colored state to the j-colored state in the target DFA; *accepting color variables* z_i which indicate whether the i-colored state in the target DFA is accepting. There are four mandatory and four redundant types of clauses in the proposed compact encoding. The reader can read about them in detail in [10]. The final step of the translation-to-SAT approach is to run an external SAT solver with the built CNF formula. If the formula is satisfiable, then the target DFA can be easily constructed from the found satisfying assignment. Otherwise, the number of colors C is increased.

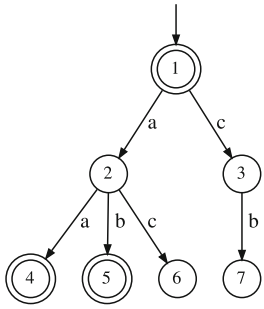
2.2 Symmetry Breakings Predicates

Large Clique Predicates. Heule and Verwer used symmetry breaking predicates in their algorithm [10]. In the case when the CG cannot be colored into C colors the SAT solver tries to solve the same problem $C!$ times—one time for each permutation of colors. In other words the solver considers $C!$ isomorphic automata. The authors suggested to find some large clique in the CG and to fix the colors of its vertices. It helps to reduce the number of unnecessary considerations because in any valid graph coloring all vertices in a clique obviously have different colors. Thus, assuming that the size of the found clique is k, the solver considers only $(C - k)!$ isomorphic automata. Moreover, the process of iterating over C can be started from k instead of 1.

BFS-based Predicates. We proposed the new approach to symmetry breaking in our previous research [17]. Its main idea is to enforce DFA states to be enumerated in the *breadth-first search* (BFS) order. If some order (say lexicographical) on the transition symbols is fixed then only one representative of each equivalence class with respect to the isomorphic relation is BFS-enumerated due to the uniqueness of such BFS traversal. We call a DFA *BFS-enumerated* if its enumeration corresponds to the order of states processing during the BFS traversal. In other words, if we consider a BFS tree, built for some DFA and if we arrange the children of each state from left to right according to the chosen order on the transition symbols then numbers of states should increase from left to right on the same depth (*layer-order*) and from top to bottom (*depth-order*). In [17] we used the definition based on a BFS-queue which is equivalent to the one described above but less apprehensible. An example of a BFS-enumerated DFA is shown in Fig. 2a, and its BFS tree is shown in Fig. 2b.



(a) An example of a BFS-enumerated DFA



(b) A BFS tree of the DFA from Fig. 2a

Fig. 2. A BFS-enumerated DFA and its BFS-tree

If such predicates are used then while a SAT solver searches for a DFA consistent with the given samples, it is restricted to only BFS-enumerated ones. To implement this we proposed three additional kinds of Boolean variables:

1. **p**arent variables $p_{j,i}$ which are true if and only if state i is the parent of state j in the BFS tree;
2. **t**ransition variables $t_{i,j}$ which are true if and only if there is a transition from state i to state j;
3. **m**inimum symbol variables $m_{l,i,j}$ which are true if and only if there is a l-labeled transition from state i to state j and there are no such transitions labeled with a smaller symbol (according to the choosen order on symbols). These variables are used only in the case of a non-binary alphabet.

BFS-enumeration is enforced with the following seven clauses:

1. $\bigwedge_{1 \leqslant i < j \leqslant C} (t_{i,j} \Leftrightarrow y_{l_1,i,j} \vee \ldots \vee y_{l_L,i,j})$—definition of transition variables using variables $y_{l,i,j}$;
2. $\bigwedge_{1 \leqslant i < j \leqslant C} (p_{j,i} \Leftrightarrow t_{i,j} \wedge \neg t_{i-1,j} \wedge \ldots \wedge \neg t_{1,j})$—definition of parent variables using variables $t_{i,j}$;
3. $\bigwedge_{2 \leqslant j \leqslant C} (p_{j,1} \vee p_{j,2} \vee \ldots \vee p_{j,j-1})$—each state except the start one holds a parent with a smaller number (depth-order);
4. $\bigwedge_{1 \leqslant k < i < j < C} (p_{j,i} \Rightarrow \neg p_{j+1,k})$—the ordering of children must be the same as the ordering of parents (layer-order for children of different parents);
5. $\bigwedge_{1 \leqslant i < j < C} (p_{j,i} \wedge p_{j+1,i} \Rightarrow y_{a,i,j})$—in case of a binary alphabet this constraint is sufficient to order two children j and $j+1$ of state i (layer-order for children of one parent);
6. $\bigwedge_{1 \leqslant i < j \leqslant C} \bigwedge_{1 \leqslant n \leqslant L} (m_{l_n,i,j} \Leftrightarrow y_{l_n,i,j} \wedge \neg y_{l_{n-1},i,j} \wedge \ldots \wedge \neg y_{l_1,i,j})$—definition of minimum symbol variables using variables $y_{l,i,j}$ which are used in case of a non-binary alphabet;
7. $\bigwedge_{1 \leqslant i < j < C} \bigwedge_{1 \leqslant k < n \leqslant L} (p_{j,i} \wedge p_{j+1,i} \wedge m_{l_n,i,j} \Rightarrow \neg m_{l_k,i,j+1})$—in case of a non-binary alphabet this constraint forces children of a state to be ordered according to the chosen order on symbols (layer-order for children of one parent).

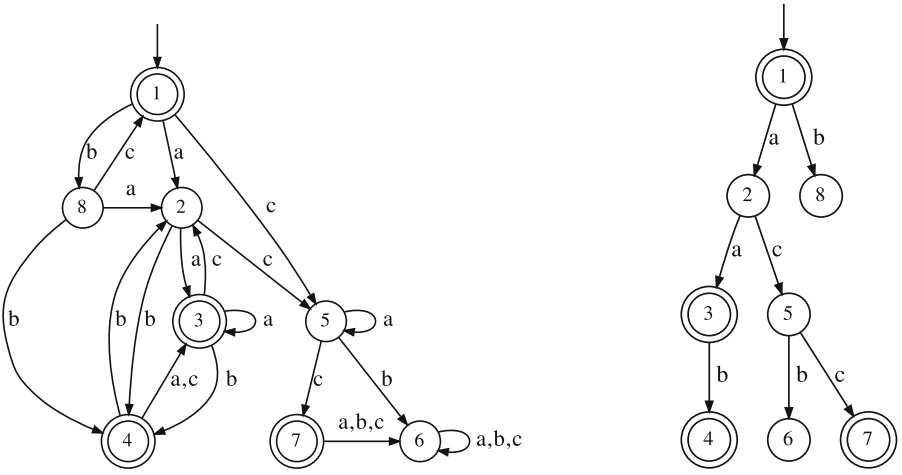
Using variables and clauses described above one can force an automaton to be BFS-enumerated. Unfortunately the implementation of these methods was not perfect when we prepared our previous paper [17] so the results did not show the real improvement caused by the proposed method. We revisited it and performed new evaluation experiments. The results are shown in Sect. 5 and they are quite impressive.

3 DFS-Based Symmetry Breaking Predicates

In this section we propose a new way to fix automata states enumeration to avoid consideration of isomorphic automata during SAT solving. This approach is a modification of our BFS-based predicates. It enforces automata states to be enumerated in the *depth-first search* (DFS) order. We describe the method briefly, paying attention only to the differences between DFS- and BFS-based approaches. Detailed information about BFS-based predicates can be found in Sect. 2.2.

During DFS processing it is necessary to find all adjacent unvisited states for each unvisited state of the DFA. Firstly, the DFS algorithm handles the initial DFA state. Then the algorithm processes the children of this state and recursively executes for each of them. We process child states in some particular (e.g., alphabetical) order of symbols l on transitions $i \xrightarrow{l} j$. Thus again only one representative of each equivalence class with respect to the isomorphic relation will be processed. We call a DFA *DFS-enumerated* if its states are numbered in

the order of handling them by DFS traversal with chosen symbol order. Although there is no traversal, we refer to it for the definition and explanation. The set of developed constraints enforces DFS. An example of a DFS-enumerated DFA is shown in Fig. 3a. A DFS tree for this DFA is shown in Fig. 3b.



(a) An example of a DFS-enumerated DFA

(b) A DFS tree of the DFA from Fig. 3a

Fig. 3. A DFS-enumerated DFA and its DFS tree

All variables which were used for the BFS enumeration are also used for the DFS enumeration, but some the constraints must be changed. In the DFS enumeration $p_{j,i}$ variables ($p_{j,i}$ is true if and only if state i is the parent of j in the DFS tree) are defined differently. Due to the greediness of the DFS algorithm, state i is the parent of state j if it has the maximum number among states that have a transition to j:

$$\bigwedge_{1 \leq i < j \leq C} (p_{j,i} \Leftrightarrow t_{i,j} \wedge \neg t_{i+1,j} \wedge \ldots \wedge \neg t_{j-1,j}),$$

where $t_{i,j} \equiv 1$ if and only if there is a transition between i and j (these variables in their turn are defined by using $y_{l,i,j}$ variables).

Moreover, in the DFS enumeration instead of the children ordering constraint we use the following one. If i is the parent of state j and k is a state between i and j ($i < k < j$) then there is no transition from state k to state q, where q is bigger than j:

$$\bigwedge_{1 \leq i < k < j < q < C} (p_{j,i} \Rightarrow \neg t_{k,q}).$$

Indeed, since $i < k < j$, state k has to be considered by the DFS algorithm before state j. Hence if such a transition would exist then state k must have a lower number than state j.

Now, to enforce the DFA to be DFS-enumerated we have to order children according to symbols on transitions (e.g., alphabetically). We consider two cases: alphabet Σ consists of two symbols $\{a, b\}$ and more than two symbols $\{l_1, \ldots, l_L\}$. In the case of two symbols state i can have only two transitions: to state j and to state k (where without loss of generality $j < k$). If the transition from state i to state j is used during the DFS traversal then it must be labeled with a smaller symbol:

$$\bigwedge_{1 \le i < j < k < C} (p_{j,i} \wedge t_{i,k} \Rightarrow y_{a,i,j}),$$

because otherwise state k had to be processed earlier.

In the second case we have to use $m_{l,i,j}$ variables: $m_{l,i,j}$ is true if and only if there is an l-labeled transition from state i to state j and there is no transition from state i to state j with an alphabetically smaller symbol. The idea is similar to the previous case. For state i it remains to arrange its children in the chosen order. For any two transitions from state i to state j and from state i to state k (where without loss of generality $j < k$), if state j is used during the DFS traversal then it must be labeled with a smaller symbol:

$$\bigwedge_{1 \le i < j < k \le C} \bigwedge_{1 \le m < n \le L} (p_{j,i} \wedge t_{i,k} \wedge m_{l_n,i,j} \Rightarrow \neg m_{l_m,i,k}).$$

Thus we proposed the new set of constraints which enforce a DFA to be DFS-enumerated. The predicates (for the case of three or more symbols) translated into $\mathcal{O}(C^4 + C^3L^2)$ (where C is the number of colors and L is the alphabet size) CNF clauses which are listed in Table 1 together with BFS-based predicates, which are translated into $\mathcal{O}(C^3 + C^2L^2)$ clauses.

Table 1. DFS-based and BFS-based symmetry breaking clauses

	Clauses	Range
Both	$t_{i,j} \Rightarrow (y_{l_1,i,j} \vee \ldots \vee y_{l_L,i,j})$	$1 \le i < j \le C$
	$y_{i,j,l} \Rightarrow t_{i,j}$	$1 \le i < j \le C$; $l \in \Sigma$
	$p_{j,i} \Rightarrow t_{i,j}$	$1 \le i < j \le C$
	$p_{j,1} \vee p_{j,2} \vee \ldots \vee p_{j,j-1}$	$2 \le j \le C$
	$m_{l,i,j} \Rightarrow y_{l,i,j}$	$1 \le i < j \le C$; $l \in \Sigma$
	$m_{l_n,i,j} \Rightarrow \neg y_{l_k,i,j}$	$1 \le i < j \le C$; $1 \le k < n \le L$
	$(y_{l_n,i,j} \wedge \neg y_{l_{n-1},i,j} \wedge \ldots \wedge \neg y_{l_1,i,j}) \Rightarrow m_{l_n,i,j}$	$1 \le i < j \le C$; $1 \le n \le L$
DFS	$p_{j,i} \Rightarrow \neg t_{k,j}$	$1 \le i < k < j \le C$
	$(t_{i,j} \wedge \neg t_{i+1,j} \wedge \ldots \wedge \neg t_{j-1,j}) \Rightarrow p_{j,i}$	$1 \le i < j \le C$
	$p_{j,i} \Rightarrow \neg t_{k,q}$	$1 \le i < k < j < q \le C$
	$(p_{j,i} \wedge p_{k,i} \wedge m_{l_n,i,j}) \Rightarrow \neg m_{l_m,i,k}$	$1 \le i < j < k \le C$; $1 \le m < n \le L$
BFS	$p_{j,i} \Rightarrow \neg t_{k,j}$	$1 \le k < i < j \le C$
	$(t_{i,j} \wedge \neg t_{i-1,j} \wedge \ldots \wedge \neg t_{1,j}) \Rightarrow p_{j,i}$	$1 \le i < j \le C$
	$p_{j,i} \Rightarrow \neg p_{j+1,k}$	$1 \le k < i < j < C$
	$(p_{j,i} \wedge p_{j+1,i} \wedge m_{l_n,i,j}) \Rightarrow \neg m_{l_m,i,j+1}$	$1 \le i < j < C$; $1 \le m < n \le L$

4 The Find-All Problem

In this section we consider the problem of finding all non-isomorphic DFA (*find-all problem*) with the minimum number of states which are consistent with a given set of strings. We propose a way to modify the SAT-based method of solving regular DFA identification problem in order to apply it to the find-all problem. We consider two ways of using SAT solvers: restarting a non-incremental solver after finding each automaton and using an incremental solver—if such a solver finds a solution, it retains its state and is ready to accept new clauses. The most common interface and technique for incremental SAT solving was proposed in [19]. We also propose the heuristic backtracking method as a baseline for comparing it with SAT-based ones.

4.1 SAT-Based Methods

The main idea of SAT-based methods of solving the find-all problem is to ban satisfying interpretations (variable values) which have already been found. It is obvious that if the proposed symmetry breaking predicates are not used then this approach finds many isomorphic automata—exactly $C!$ for each equivalence class where C is the DFA size. Since max-clique predicates fix k colors only (where k is the clique size), the algorithm of Heule and Verwer finds $(C - k)!$ isomorphic automata which is still bad. The BFS-based and DFS-based symmetry breaking predicates allow us to ban isomorphic DFA from one equality class by banning an accordingly enumerated representative. It must be noted that although the idea to discard satisfying interpretations is classic for such methods, it cannot be used in practice without effective symmetry breaking techniques. There were no known techniques to deal with factorial number of isomorphic automata earlier, and thus the considered problem could not be solved effectively. Proposed symmetry breaking predicates change the situation and bring the solution. It is easy to implement this by adding a *blocking* clause into the Boolean formula. Since we know that $y_{l,i,j}$ variables define the structure of the target DFA entirely, it is enough to forbid only values of these variables from the found interpretation:

$$\neg y_1 \vee \neg y_2 \vee \ldots \vee \neg y_{n|\Sigma|},$$

where y_k is some $y_{l,i,j}$ from the found interpretation for $1 < k < n|\Sigma|$.

There are two different ways of using SAT solvers as it was stated above. First, we can restart a non-incremental SAT solver with the new Boolean formula with the blocking clause after finding each automaton. The second approach is based on incremental SAT solvers: after each found automaton we add the blocking clause to the solver and continue its execution.

It is necessary to mention the case when some transitions of the found DFA are not covered by the APTA. It means that there are some *free* transitions which are not used during processing any given word and each such transition can end in any state, since this does not influence the consistency of the DFA with a given set of strings. But in the case of the find-all problem basically we

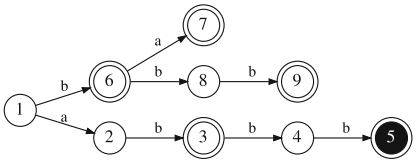
do not wish to find all these automata distinguished only by such transitions. Thus we propose a way to force all free transitions to be self-loops—end in the same state as they start. To achieve that we add auxiliary '**u**sed' variables: $u_{l,i}$ is true if and only if there is an l-labeled APTA edge from the i-colored state:

$$\bigwedge_{l \in \Sigma} \bigwedge_{1 \leq i \leq C} u_{l,i} \Leftrightarrow x_{1,i} \vee \ldots \vee x_{|V_l|,i},$$

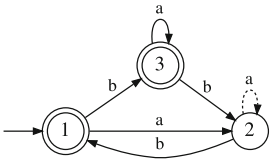
where V_l is the set of all the APTA states which have an outcoming edge labeled with l. To force unused transitions to be self-loop we add the following constraints:

$$\bigwedge_{l \in \Sigma} \bigwedge_{1 \leq i \leq C} \neg u_{l,i} \Rightarrow y_{l,i,i}.$$

These additional constraints are translated into $\mathcal{O}(C|L|)$ clauses. See Fig. 4 for an example of an APTA for $S_+ = \{ab, b, ba, bbb\}$ and $S_- = \{abbb\}$ and its consistent DFA with an unused transition. If we add the proposed constraints, then this transition will be forced to be a loop as shown by a dashed line in Fig. 4b.



(a) An example of an APTA for $S_+ = \{ab, b, ba, bbb\}$ and $S_- = \{abbb\}$



(b) The DFA is built from the APTA from Fig. 4a with unused a-labeled transition from state 2

Fig. 4. An example of an APTA and its consistent DFA

4.2 Backtracking Algorithm

The solution based on backtracking does not use any external tools like SAT solvers. This algorithm works as follows. Initially there is an empty DFA with n states. Also there is a *frontier*—the set of edges from the APTA which are not yet represented in the DFA. Initially the frontier contains all outcoming edges of the APTA root. The recursive function `Backtracking` maintains the frontier in the proper state. If the frontier is not empty, then the function tries to augment the DFA with one of its edges. Each found DFA is checked to be consistent with the APTA and if the DFA complies with it then an updated frontier is found. If the frontier is empty then the DFA is checked for completeness (a DFA is complete if there are transitions from each state labeled with all alphabet symbols). If it is not complete and there are nodes which have the number of

outcoming edges less than the alphabet size then we add missing edges as self-loops with function `MakeComplete`. Algorithm 1 illustrates the solution. The function `FindNewFrontier` returns the new frontier for the augmented DFA or null if the DFA is inconsistent with the APTA. This algorithm is an exact search algorithm based on the one from [18].

```
Data: augmented prefix tree acceptor APTA, current DFA (initially empty),
      frontier (initially contains all APTA root outcoming edges)
DFAset ← new Set<DFA>
edge ← any edge from frontier
foreach destination ∈ 1..|S| do
    source ← the state of DFA from which edge should be added
    DFA′ ← DFA ∪ transition(source, destination, edge.label)
    frontier′ ← FindNewFrontier(APTA, DFA′, frontier)
    if frontier′ ≠ null then
        if frontier′ = ∅ then
            DFAset.add(MakeComplete(DFA′))
        else
            DFAset.add(Backtracking(APTA, DFA′, frontier′))
        end
    end
end
return DFAset
```

Algorithm 1. Backtracking solution

5 Experiments

All experiments were performed using a machine with an AMD Opteron 6378 2.4 GHz processor running Ubuntu 14.04. All algorithms were implemented in Java, the *lingeling* SAT solver was used [20]. As far as we know all common benchmarks are too hard for solving by exact algorithms without some heuristic non-exact steps. Thus our own algorithm was used for generating problem instances. This algorithm builds a set of strings with the following parameters: size N of DFA to be generated, alphabet size A, the number S of strings to be generated. The algorithm is arranged as follows. First of all N states are generated and uniquely numerated from 1 to N. Each state is equiprobably set to be accepting or rejecting. Next on step i the algorithm picks state i, evenly chooses another state from $[i+1; N]$ and adds a random-labeled transition from the first state to the second. After $N-1$ such steps we have partially built an automaton where all states are reachable from the initial one (1-numbered). In the end the algorithm picks each state one by one and add all missing (in terms of automaton completeness) transitions with destination randomly chosen among all states. Finally S strings are generated by processing the automaton.

The distribution of the words' length is shifted to longer words. These strings with the accepting or rejecting labels form the instance of the DFA identification problem.

For DFA identification we used the following parameters: $N \in [10; 30]$ with step 2, $A = 2$, $S = 50N$. We compared the SAT-based approach with three types of symmetry breaking predicates: the max-clique algorithm from [10] (the current state-of-the-art) and the proposed DFS-based and BFS-based methods. Each experiment was repeated 100 times. The time limit was set to 3600 s. The results are listed in Table 2. It can be seen from the table that both DFS-based and BFS-based strategies clearly outperform the max-clique approach. BFS-based strategy in its turn notably outperforms DFS-based one when target automaton size is larger than 14. These results for the BFS-based approach were not obtained in our previous research due to weaker technical implementation.

Table 2. Median execution times of exact solving DFA identification in seconds

N	DFS	BFS	max-clique
10	20.9	20.5	23.3
12	40.4	37.6	240.3
14	82.2	62.4	TL
16	205.1	114.1	TL
18	601.7	181.9	TL
20	2501.6	293.7	TL
22	TL	453.3	TL
24	TL	625.1	TL
26	TL	925.8	TL
28	TL	1314.4	TL
30	TL	1635.5	TL

The second experiment concerned the find-all problem. A random dataset was also used here. We used the following parameters: $N \in [5; 15]$, $A = 2$, $S \in \{5N, 10N, 25N\}$. We compared the BFS-SAT-based method with the restarting strategy (REST column in the table), the BFS-SAT-based method with the incremental strategy (INC) and the backtracking method (BTR). Each experiment was repeated 100 times as well. The time limit was set to 3600 s. The results are given in Table 3. The first column in each subtable contains the number of instances which have more than one DFA in the solution (> 1). If less than 50 instances were solved then TL is shown instead of a value. It can be seen from the table that SAT-based methods work significantly faster than the backtracking one when the size of the automaton is greater than 8. It happens because the SAT-based methods with BFS-based predicates consider only one DFA for each equivalence class with respect to the isomorphic relation instead

Table 3. Median execution times in seconds of SAT-based restart method, SAT-based incremental method and backtracking method

$\|N\|$	$S = 5\|N\|$				$S = 10\|N\|$				$S = 25\|N\|$			
	>1	REST	INC	BTR	>1	REST	INC	BTR	>1	REST	INC	BTR
5	53	2.3	2.0	0.8	40	3.6	3.3	1.3	17	4.1	3.4	1.5
6	56	2.8	2.4	2.1	31	4.7	3.9	1.7	27	5.4	4.3	1.7
7	87	3.9	2.5	4.1	27	3.7	3.0	3.1	13	7.4	6.7	2.5
8	80	4.6	3.7	87.2	34	7.0	6.5	41.7	16	10.1	8.9	11.6
9	91	7.6	3.9	475.1	50	7.7	6.4	121.6	10	13.8	13.0	61.4
10	89	15.7	5.3	2756.2	47	8.6	7.0	974.7	11	18.8	16.1	276.8
11	94	19.9	7.3	TL	63	18.5	13.8	3108.0	9	24.5	21.9	1158.4
12	90	28.0	9.9	TL	49	22.3	16.7	TL	8	33.5	27.2	3289.1
13	92	185.5	18.1	TL	57	36.9	22.6	TL	12	62.0	51.4	TL
14	87	408.5	49.0	TL	71	85.1	41.8	TL	4	67.0	56.2	TL
15	95	571.1	174.1	TL	69	193.3	95.7	TL	6	29.2	26.2	TL

of $N!$. As we see, the incremental strategy in its turn clearly outperforms the restart strategy. It can be explained as incremental SAT solver saves its state but non-incremental solver does the same actions on each execution.

Our implementation of proposed predicates and algorithms is available on our laboratory github repository[1].

6 Conclusions

We have proposed DFS-based symmetry breaking predicates. They can be added to the Boolean formula before passing it to a SAT solver while solving various DFA identification problems with SAT-based algorithms. Using these predicates allows reducing the problem search space by enforcing DFA states to be enumerated in the depth-first search order.

We have revisited our implementation of the proposed symmetry breaking predicates and compared the translation-to-SAT method from [10] to the same one with proposed symmetry breaking predicates instead of original max-clique predicates. The proposed approach clearly improved the translation-to-SAT technique which was demonstrated with the experiments on randomly generated input data. The BFS-based approach has shown better results than the DFS-based one if the target DFA size is large.

Then, we have proposed a solution for the find-all DFA problem. The proposed approach can efficiently solve the problem that the previously developed methods cannot be applied for. We performed the experiments which have shown

[1] https://github.com/ctlab/DFA-Inductor.

that our approach with the incremental SAT solver clearly outperfoms the Backtracking algorithm.

Acknowledgements. The authors would like to thank Igor Buzhinsky, Daniil Chivilikhin, Maxim Buzdalov for useful comments. This work was financially supported by the Government of Russian Federation, Grant 074-U01.

References

1. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Boston (2006)
2. De La Higuera, C.: A bibliographical study of grammatical inference. Pattern Recogn. **38**(9), 1332–1348 (2005)
3. Gold, E.M.: Complexity of automaton identification from given data. Inf. Control **37**(3), 302–320 (1978)
4. Dupont, P.: Regular grammatical inference from positive and negative samples by genetic search: the GIG method. In: Carrasco, R.C., Oncina, J. (eds.) ICGI 1994. LNCS, vol. 862, pp. 236–245. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58473-0_152
5. Luke, S., Hamahashi, S., Kitano, H.: Genetic programming. In: Proceedings of the genetic and evolutionary computation conference, vol. 2, pp. 1098–1105 (1999)
6. Lucas, S.M., Reynolds, T.J.: Learning DFA: evolution versus evidence driven state merging. In: The 2003 Congress on Evolutionary Computation, 2003. CEC 2003, vol. 1, pp. 351–358. IEEE (2003)
7. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Honavar, V., Slutzki, G. (eds.) ICGI 1998. LNCS, vol. 1433, pp. 1–12. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054059
8. Lang, K.J.: Faster algorithms for finding minimal consistent DFAs. Technical report (1999)
9. Bugalho, M., Oliveira, A.L.: Inference of regular languages using state merging algorithms with search. Pattern Recogn. **38**(9), 1457–1467 (2005)
10. Heule, M.J.H., Verwer, S.: Exact DFA identification using SAT solvers. In: Sempere, J.M., García, P. (eds.) ICGI 2010. LNCS (LNAI), vol. 6339, pp. 66–79. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15488-1_7
11. Lohfert, R., Lu, J.J., Zhao, D.: Solving SQL constraints by incremental translation to SAT. In: Nguyen, N.T., Borzemski, L., Grzech, A., Ali, M. (eds.) IEA/AIE 2008. LNCS (LNAI), vol. 5027, pp. 669–676. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69052-8_70
12. Galeotti, J.P., Rosner, N., Pombo, C.G.L., Frias, M.F.: TACO: efficient SAT-based bounded verification using symmetry breaking and tight bounds. IEEE Trans. Softw. Eng. **39**(9), 1283–1307 (2013)
13. Ulyantsev, V., Tsarev, F.: Extended finite-state machine induction using SAT-solver. In: Proceedings of ICMLA 2011, vol. 2, pp. 346–349. IEEE (2011)
14. Zbrzezny, A.: A new translation from ECTL* to SAT. Fundamenta Informaticae **120**(3–4), 375–395 (2012)
15. Walkinshaw, N., Lambeau, B., Damas, C., Bogdanov, K., Dupont, P.: STAMINA: a competition to encourage the development and assessment of software model inference techniques. Empirical Software Engineering **18**(4), 791–824 (2013)

16. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. KR **96**, 148–159 (1996)
17. Ulyantsev, V., Zakirzyanov, I., Shalyto, A.: BFS-based symmetry breaking predicates for DFA identification. In: Dediu, A.-H., Formenti, E., Martín-Vide, C., Truthe, B. (eds.) LATA 2015. LNCS, vol. 8977, pp. 611–622. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15579-1_48
18. Ulyantsev, V., Buzhinsky, I., Shalyto, A.: Exact finite-state machine identification from scenarios and temporal properties. Int. J. Softw. Tools Technol. Transf. 1–21 (2016)
19. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
20. Biere, A.: Splatz, lingeling, plingeling, treengeling, YalSAT entering the SAT competition 2016. In: Proceedings of SAT Competition, pp. 44–45 (2016)