An implementation of: exact DFA identification using SAT solvers

Roberto Cipollone Marco Uccelli

Abstract

The algorithm introduced by Heule and Verwer in the paper "Exact DFA identification using SAT solvers" represented a step forward in the study of DFA identification. The authors proposed, as a new approach, a translation of the DFA identification problem to a SAT instance, to take advantage of the more powerful SAT solvers. We present an implementation in the Java programming language of their algorithm and we show and analyze the results obtained.

Contents

1 Introduction **2**

2 DFA identification **2**
2.1 Problem definition . . . 3
2.2 The APTA . . . 4
2.3 Constraints of the identification problem . . . 5
2.4 Encoding in a SAT problem . . . 6
2.5 Extracting the solution . . . 10

3 Implementation **10**
3.1 CNF . . . 11
3.2 Automata . . . 12
3.2.1 Graphs . . . 12
3.2.2 APTA . . . 14
3.2.3 DFA . . . 14
3.2.4 Visualization . . . 15
3.3 Identification . . . 16
3.3.1 ConstraintsGraph . . . 16
3.3.2 Encoding variables . . . 17
3.3.3 Solver . . . 18
3.4 Utils . . . 19

4 How to use the software **20**

5 Results **21**
5.1 Dataset . . . 21
5.2 Outcomes . . . 22

6 Conclusion **23**

1 Introduction

The Deterministic Finite Automaton (DFA) is the most basic and fundamental model of the automata theory, and the problem of identifying a DFA stands among the most studied problems in grammatical inference. The aim of DFA identification is to find the DFA with the minimum number of states that is consistent with a given set of accepted and rejected sequences. The applications of DFA identification are many, for example computational linguistics, speech processing and verification.

It is also a very difficult problem, and it has been shown to be NP-complete. Nonetheless, a lot of DFA identification algorithms have been developed, achieving very good performances. The initial approaches found the firsts efficient solutions using evolutionary computation methods, and they were later replaced by multi-start random hill climbers. The methods were progressively refined till reaching the current state-of-the-art, which is the Evidence-Driven State Merging (EDSM) algorithm [2], an heuristic based algorithm. EDSM is a greedy algorithm that essentially finds a local optimum in polynomial time, and is one of the few ones that can handle large-sized target DFA.

However, two observations can be made when analyzing EDSM and its variants: the first is that those algorithms are not exact, so they can't guarantee that the DFA found has the minimum size; the second is that, however advanced they are, they are still far behind the level of solvers used for other well-studied problems like graph coloring and satisfiability (SAT). To respond to these observations, Heule and Verwer developed, in [1], a new algorithm. Basically they proposed an approach in which the original problem is translated to a SAT instance, so as to exploit the power of modern SAT solvers. The results obtained were very positive and their approach was proved to be very competitive in most problems except the largest instances (such as the Abbandigo One challenge).

In this report we show the work we have done to implement a usable and publicly available version of the algorithm in Java, explaining the whole algorithm in detail and clearing up the functioning of our code.

The document is organized as follows. In Section 2 we look at the theory behind the DFA identification problem and we explain the approach used in the original paper. In Section 3, we thoroughly cover our implementation of the paper, looking at each package and each class of the project. In Section 4, we give a brief explanation on how to use our software, what libraries are needed, how to organize the directories and how to run it. In Section 5, we present our practical results, and we end up with some conclusions in Section 6.

2 DFA identification

This section presents all the concepts we need to understand the identification problem and how it has been solved. All the structures we'll go through, even the most basic models, will find counterparts in our implementation. Therefore, this part will also help in understanding the code. Most of this discussion covers the topics of the first sections in our main reference [1].

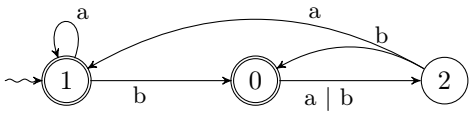


Figure 1: A simple DFA: $\Sigma = \{\text{a,b}\}, Q = \{q_0, q_1, q_2\}, F = \{q_0, q_1\}$, initial state q_1, δ defined by the directed arcs. The vertical bar | is used when more than one symbol is associated to the same transition.

2.1 Problem definition

A **Deterministic Finite Automaton** (DFA) is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ is the set of input symbols (called the alphabet); Q is a finite set of states; $q_0 \in Q$ is the initial state; $\delta : Q \times \Sigma \to Q$ is the (total) transition function; and $F \subseteq Q$ is the set of final states. We say that a $\mathcal{A}$ accepts a string $s = a_1 a_2 \dots a_n \in \Sigma^*$ (Σ^* is the reflexive closure of Σ; any concatenation of symbols) if and only if the repeated application of the transition function from the initial state leads to one of the final states:

$$\begin{array}{ll} g_0 = q_0 & \\ g_i = \delta(g_{i-1}, a_i) \quad \forall i \in 1, \dots, n & \wedge \qquad g_n \in F \end{array}$$

So, a DFA is a simple abstract model that processes strings in input by moving the current state through the set Q and returning a positive response if, after the input has been completely parsed, a final state is reached. The language recognized by $\mathcal{A}$, denoted as $\mathcal{L}(\mathcal{A})$, is the set of strings accepted by $\mathcal{A}$. It is known that DFAs recognize (accept) the class of regular languages.

Let's look at the example in Figure 1. This automaton accepts strings such as: ϵ (the empty string), "b", "a", "bab"; instead, it rejects: "ba", "bb". As always, the language of recognized by this DFA can be expressed with a regular expression.

Sometimes, a graph may describe a transition function which is not total. This happens when there exists a symbol, $l \in \Sigma$, and a state, $q_i \in Q$, for which the outgoing arc from node i, associated to l, is missing. Usually, such a function can be regarded as complete, because traversing a missing transition is equivalent to reaching a dead-end (a rejecting state from which no final states can be reached).

Even if few examples in this document might use the English alphabet for simplicity, the definition allows any discrete set of symbols as input alphabet. For example, each symbol in Σ can represent an event in the environment that we want to distinguish: $\Sigma = \{\texttt{event}_1, \dots, \texttt{event}_L\}$. In this context, a DFA could represent acceptable (or possible) sequences of **events** in the domain we're considering. Sometimes, we will use the term "trace" as a synonym of string, when the input of the DFA is a sequence of events. They are sequences, in general.

Our problem is the opposite of the usual procedure: given a set of sequences and their assignments, we want to find the *smallest* DFA that would produce the same assignments. More precisely, given two sets of sequences S^+ and S^-, **DFA identification** is the problem of finding the smallest DFA, $\mathcal{A}$, that is *consistent* with the input sequences: $s^+ \in \mathcal{L}(\mathcal{A}), s^- \notin \mathcal{L}(\mathcal{A}) \quad \forall s^+ \in S^+, s^- \in S^-$. The *size* of a DFA is the number of states it contains, $|Q|$.

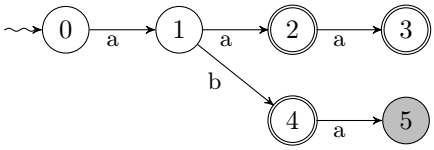


Figure 2: An APTA: $\Sigma = \{\mathrm{a}, \mathrm{b}\}, Q^+ = \{q_2, q_3, q_4\}, Q^- = \{q_5\}, Q^? = \{q_0, q_1\}$.

While parsing an input string with a DFA can be done with time $O(|s|)$, *learning* the structure of a consistent DFA of fixed size is an NP-complete problem.

2.2 The APTA

We solve DFA identification using a merging algorithm. This means that first we build a rough approximation of the solution, then we apply the merging algorithm to find the minimum consistent DFA. Now, we will focus on the first part.

The structure that is used to create the first automaton is the **Augmented prefix tree acceptor** (APTA). A prefix tree acceptor (PTA) is just a DFA without cycles (graphs can represent a DFA, while a PTA is a tree). Instead, an *Augmented* PTA uses a slightly different labelling for the states. In fact, in place of the final states, F, an APTA defines a set of *accepting* states, Q^+, and a set of *rejecting* states, Q^-. Due to this distinction, there exists a third set of states, that we can denote with $Q^?$, which do not accept nor reject. $Q^+, Q^-, Q^?$ form a partition of Q.

Let's take, for example, the APTA in Figure 2. While the result of parsing "aa" and "aba" it's clear, the response to ϵ and "a" is not defined (we say it's undefined, or unknown). The most sensible way to complete the transition function of an APTA is to add arcs that reach a dead-end state in $Q^?$. Therefore, the response associated to "b" is undefined. The example also shows why this is called a prefix tree: the computations of two strings, $s_1 = p_0 p_1$ and $s_2 = p_0 p_2$, are the same until the common prefix, p_0, is completely parsed.

To convert an APTA into a DFA it is sufficient to choose whether each node in $Q^?$ is final or not. For example, we could just define $F = Q^+$ (any other state will be rejecting).

Creating an APTA that is consistent with a set of traces is straightforward. Starting from an empty automaton (or one containing just the initial state), we extend the tree with all the transitions we need to completely parse the sequence, and we assign a response to the state reached or the newly created leaf. The example of Figure 2 has been created from the sequences: $S^+ = \{\texttt{aa}, \texttt{aaa}, \texttt{ab}\}$ and $S^- = \{\texttt{aba}\}$. An APTA is also a compact representation of the sequences, because the tree structure emphasize the most frequent prefixes.

Thus, the first step of our algorithm is to create an APTA from the traces in input. As we've seen, this automaton is consistent with the input traces, but it's not minimal. That's why we use the merging algorithm.

2.3 Constraints of the identification problem

The goal of a **state-merging algorithm** is to iteratively select and merge pairs of compatible states until a minimum automaton is reached. Few concepts in this definition need to be defined. First, a DFA $\mathcal{A}$ is *minimum* if the language it recognizes can't be recognized by any smaller DFA: no DFAs $\mathcal{A}'$ exists such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$ and $|Q'| < |Q|$. The rest of this section will explain what compatible states are.

What we should notice is that these definitions are related to the general problem of merging states of an automaton, not the particular merging algorithm. As we will see, our algorithm is not even iterative, but the same considerations apply. Understanding the structure of the problem is essential if we want to exploit it in our solution (as for any Constraint Satisfaction Problem).

A state-merging algoritm can be applied to any Finite State Automaton that uses the same state labelling as the APTA. In fact, the undefined responses are the only that allow the automaton to change. Instead, the accepting and rejecting states limit the feasible merges. Our algorithm will only use APTAs.

Let's denote with $\mathcal{L}^+(\mathcal{A})$ and $\mathcal{L}^-(\mathcal{A})$ the language respectively accepted and rejected by an automaton $\mathcal{A}$. Two states $q_a, q_b \in Q$ are **compatible** if and only if:

$$\mathcal{L}^+(\mathcal{A}_a) \cap \mathcal{L}^-(\mathcal{A}_b) = \emptyset \quad \text{and} \quad \mathcal{L}^-(\mathcal{A}_a) \cap \mathcal{L}^+(\mathcal{A}_b) = \emptyset \tag{1}$$

where $\mathcal{A}_a$ and $\mathcal{A}_b$ are two new automata with q_a and q_b as initial states.

Compatible states can be merged and we say that the merge is **consistent**. A merge of q_a and q_b removes these nodes and creates a new state with the union of all connections of q_a and q_b. While the incoming edges can be safely joined, merging the outgoing arcs creates a non-deterministic automaton. However, since (1) is satisfied, all children connected through the same label can be merged, because they are compatible too.

Languages are usually infinite, so we can't explicitly compute them. Instead, it is much easier to work directly with automata, because they implicitly define languages in a compact way. As a consequence of (1), two states q_a and q_b are *incompatible* if one of these two conditions apply:

1. $q_a \in Q^+, q_b \in Q^-$ or $q_a \in Q^-, q_b \in Q^+$. In fact, $\mathcal{A}_a$ and $\mathcal{A}_b$ would return a different response for the empty string.

2. There exists a label $l \in \Sigma$ such that, if q_a and q_b are merged, the states $\delta(q_a, l)$ and $\delta(q_b, l)$ are incompatible. In this case, there would exist a string, ls (with $s \in \Sigma^*$), for which $\mathcal{A}_a$ and $\mathcal{A}_b$ would return a different response.

In this recursive definition, any missing transition can be ignored, as it always satisfy the requirements. Two states which fail this test are compatible and the merge is consistent.

There are many consistent merges to do in Figure 2. Applying the test, we see that q_5 is incompatible with q_2, q_3, q_4, and q_4 is incompatible with q_2. Instead, (q_1, q_2) is one of the consistent merges. Figure 3 shows what is the result of this merge. Condition 2 says "if q_a and q_b are merged". What this means is that, while we recur to their children, we also need to remember all merges done along the way. This explain why q_1 has been merged with both q_2 and q_3. For the same reason, q_0 can't be merged with q_1 in Figure 4. The recursive part of

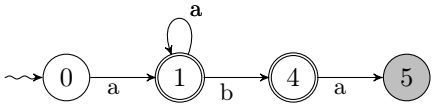


Figure 3: The result of merging q_1 and q_2 from Figure 2.

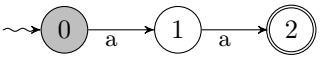


Figure 4: q_0 and q_1 are incompatible.

the merging process is sometimes called **determinization**.

A merge changes the language recognized by an automaton. This happens because when a node, $q_u \in Q^?$, is merged with $q_r \notin Q^?$ it gets the same response as q_r. For example, while the automaton in Figure 2 accepts $\{\texttt{aa}, \texttt{aaa}\}$, Figure 3 accepts any non-empty sequence of "a". However, any consistent merge can only enlarge the languages $\mathcal{L}^+$, $\mathcal{L}^-$, so this operation won't modify the response associated to the strings used to build the original APTA ($S^+ \subseteq \mathcal{L}^+$ and $S^- \subseteq \mathcal{L}^-$, always).

Even for the simplest instances, it is clear that it would be useful to visualize all the constraints at once. A **constraints graph** is the structure that is typically used to visualize *binary relations* in a Constraint Satisfaction Problem (CSP) (as we will see DFA identification is a CSP). In a CSP, we can visualize variables as vertices and binary constraints as edges of the graph. So, we can use the constraints graph to see the incompatible nodes (incompatibility is a binary relation). This graph, in the DFA identification problem, is also called *Consistency Graph* (CG).

Figure 5 shows the CG associated to the APTA in Figure 2. As we can see, q_4 can be merged with q_3, and q_0 is compatible with any other state!

2.4 Encoding in a SAT problem

Up until the publication of [1], the most used methods for solving the DFA identification problem had a common issue: they were not exact, and that means that they were not able to guarantee that the found DFA was one of the minimum-sized ones. In [1], Heule and Verwer introduced a novel technique for addressing the problem, aimed at solving this issue.

The efficiency of SAT solvers has improved exponentially over the last decade, and their power can be exploited to solve problems in other areas that haven't still reached the same level of performance. The method to wield this power is to translate the original problem into a SAT instance, and then to run a SAT solver on this translated problem. This approach has been proved as very effective on

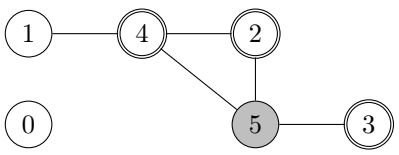


Figure 5: The constraints graph of the APTA in Figure 2.

a number of different problems and, in [1], it was proved to be very effective in dealing with the DFA identification problem too. Moreover, as opposed to EDSM, this method is exact.

SAT is the problem of determining if there exists an interpretation that satisfies a given Boolean formula or, in other words, whether the variables of the Boolean formula can be consistently replaced by true or false values in such a way that it evaluates to true. If that is the case, the formula is called satisfiable. On the other hand, if no such assignment exists, the function expressed by the formula is false for all possible variable assignments and the formula is unsatisfiable. The Boolean formula in input has to be in conjunctive normal form (CNF), meaning that it is a conjunction of clauses, each clause being a disjunction of literals. A literal refers either to a Boolean variable, or to its negation.

The translation of our problem into a SAT instance is carried out mainly in two steps. The first is to translate the DFA identification to an integer Constraint Satisfaction Problem (CSP), which is a well-known operation. The second is to translate the CSP to SAT. Traditionally, this is achieved through the unary and a binary encodings of the integers. Instead, Heule and Verwer devised a new translation of a coloring problem in a Boolean formula.

In graph theory, the coloring problem is a special case of graph labeling; it is an assignment of labels, traditionally called "colors", to elements of a graph, subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices are of the same color. The convention of using colors originates from coloring the countries of a map, where each face is literally colored.

The main idea of this translation is to use a different color for every state of the identified DFA, and hence the amount of colors should be as small as possible. Every vertex in the constraints graph of the coloring problem represents a distinct state in the APTA. Two vertices v and w in this graph are connected by an edge (cannot be assigned the same color), if merging v and w results in an inconsistency (i.e., an accepting state is merged with a rejecting state). These edges are called *inequality constraints*.

In addition to inequality constraints, another type of constraints are needed, named *equality constraints*. These type of constraints imply that, if the parents $p(v)$ and $p(w)$ of two vertices v and w with the same incoming label are merged, the children v and w must be also merged. The addition of these constraints makes some of the inequality constraints become redundant: only the inconsistent edges between accepting and rejecting states are needed, as the other edges follow logically from combining the inequality and equality constraints. Nonetheless, these redundant constraints are kept in the translation as they may help during the search process. Equality constraints are difficult to represent in the coloring problem. Hence, in our paper, the authors encoded them using satisfiability and they reduced the number of constraints thanks to additional auxiliary variables.

The most well-known and used translation of graph coloring problems into SAT is the one known as the direct encoding. Given a graph $G = (V, E)$, where V is the set of the vertices and E is the set of the edges, and a set of colors C, the direct encoding uses (Boolean) color variables $x_{v,i}$ with $v \in V$ and $i \in C$. If $x_{v,i}$ is assigned to true, it means that vertex v has color i. On these variables are then built the constraints, represented as clauses in a formula in conjunctive normal form.

Table 1: Encoding of DFA identification into SAT. C = set of colors, L = set of labels (alphabet), V =vertices, E =conflict edges.

Variables	**Range**	**Meaning**
$x_{v,i}$	$v \in V; i \in C$	$x_{v,i} \equiv 1$ iff vertex v has color i
$y_{a,i,j}$	$a \in L; i,j \in C$	$y_{a,i,j} \equiv 1$ iff parents of vertices with color j and incoming label a must have color i
z_i	$i \in C$	$z_i \equiv 1$ iff an accepting state has color i
Clauses	**Range**	**Meaning**
$(x_{v,1} \vee x_{v,2} \vee \cdots \vee x_{v,\|C\|})$	$v \in V$	each vertex has at least one color
$(\neg x_{v,i} \vee z_i) \wedge (\neg x_{w,i} \vee \neg z_i)$	$v \in V_+; w \in V_-; i \in C$	accepting vertices cannot have the same color as rejecting vertices
$(y_{l(v),i,j} \vee \neg x_{p(v),i} \vee \neg x_{v,j})$	$v \in V; i,j \in C$	a parent relation is set when a vertex and its parent are colored
$(\neg y_{a,i,h} \vee \neg y_{a,i,j})$	$a \in L; i,h,j \in C; h < j$	each parent relation can target at most one color
Redundant Clauses	**Range**	**Meaning**
$(\neg x_{v,i} \vee \neg x_{v,j})$	$v \in V; i,j \in C; i < j$	each vertex has at most one color
$(y_{a,i,1} \vee y_{a,i,2} \vee \cdots \vee y_{a,i,\|C\|})$	$a \in L; i \in C$	each parent relation must target at least one color
$(\neg y_{l(v),i,j} \vee \neg x_{p(v),i} \vee x_{v,j})$	$v \in V; i,j \in C$	a parent relation forces a vertex once the parent is colored
$(\neg x_{v,i} \vee \neg x_{w,i})$	$i \in C; (v,w) \in E$	all determinization conflicts explicitly added as clauses

It needs to be true that each vertex has a color, so for each vertex the *at-least-one* color clause makes sure that the condition is respected:

$$\bigwedge_{v \in V} (x_{v,1} \vee x_{v,2} \vee \cdots \vee x_{v,|C|})$$

Then, we need clauses that ensure that adjacent vertices don't have the same color. This constraint is translated in CNF as following:

$$\bigwedge_{i \in C} \bigwedge_{(v,w) \in E} (\neg x_{v,i} \vee \neg x_{w,i})$$

Finally, given the set of pairs of vertices, EL, that have the same incoming label in the APTA, if the parents $p(v)$ and $p(w)$ of such a pair $(v,w) \in EL$ have the same color, then v and w must have the same color as well. This condition corresponds to the *equality constraint* previously mentioned. The constraints can be translated in CNF like this:

$$\bigwedge_{i \in C} \bigwedge_{j \in C} \bigwedge_{(v,w) \in EL} \begin{array}{l} (\neg x_{p(v),i} \vee \neg x_{p(w),i} \vee \neg x_{v,j} \vee x_{w,j}) \wedge \\ (\neg x_{p(v),i} \vee \neg x_{p(w),i} \vee x_{v,j} \vee \neg x_{w,j}) \end{array}$$

A problem that's easy to encounter with direct encoding is the size. Most interesting DFA identification problems will be translated in formulae with way too many clauses to be tackled by even the most advanced and performing SAT solvers. The size of a direct encoding is $\mathcal{O}(|C|^2|V|^2)$. That's the reason that pushed Heule and Verwer to propose a new kind of encoding, appropriately named *compact*.

Compact encoding is based on the usage of two sets of new *auxiliary variables*, $y_{a,i,j}$ and z_i, to significantly reduce the number of clauses in the formula.

A variable $y_{a,i,j}$, called *parent relation variable*, if set to true, means that for any vertex with color i, the child reached by label a has color j. Let $l(v)$ denote the incoming label of vertex v, and let $c(v)$ denote the color of vertex v. As soon as both a child v_i and its parent $p(v_i)$ are colored, the corresponding parent relation variable is forced to true by the clause $(y_{l(v_i),c(p(v_i)),c(v_i)} \vee \neg x_{p(v_i),c(v_i)} \vee \neg x_{v_i,c(v_i)})$. This set of auxiliary variables reduces the size of the encoding from $\mathcal{O}(|C|^2|V|^2)$ to $\mathcal{O}(|C|^2|V|)$. Additionally, with this set of variables, there is also the need for *at-most-one* parent relation clauses, that make sure that each parent relation is unique. These clauses can be encoded as $(\neg y_{a,i,h} \vee \neg y_{a,i,j})$, with $a \in L$, $i, h, j \in C$ and $h < j$.

A variable z_i, called *accepting color variable*, if set to true means that the color i is only used for accepting vertices. This set of variable is used for the constraints that require all accepting vertices to be colored differently from the rejecting states. Without auxiliary variables, this can be encoded as $(\neg x_{v,i} \vee \neg x_{w,i})$ for $v \in V_+$, $w \in V_-$, $i \in C$, where V_+ and V_- are the sets of accepting and rejecting vertices respectively, resulting in $|V_+| \cdot |V_-| \cdot |C|$ clauses. Using the auxiliary variables z_i, the same constraints can be encoded as $(\neg x_{v,i} \vee z_i) \wedge (\neg x_{w,i} \vee \neg z_i)$, requiring only $(|V_+| + |V_-|)|C|$ clauses.

Another step taken in by paper is the use of SBP, Symmetry Breaking Predicates, which exploit the structure of the problem to speed up the process. The use of symmetry in this case is straightforward: if a graph cannot be colored with k colors, it will not be colored by any permutation of the same number of colors, so it's useful to avoid the algorithm to nonetheless try to solve the problem once for each permutation. The SBP they added fixed an assignment for a clique. A clique is a subset of vertices of a graph such that each vertex is adjacent to each other. Therefore in any valid coloring of a graph, all vertices in a clique must have a different color. This fact is used to add a preprocessing step in the algorithm, where each vertex in the largest clique that can be found is fixed to a different color. So, let's suppose we have an instance of the coloring problem with C colors and N variables and we're able to find a clique of L nodes. This preprocessing would reduce the unknown variables in our problem to $N - L$.

The encoding completed up to this point, can be also extended further. Another contribution of the paper is the addition of redundant clauses, with the goal of improving the performance of the SAT solver.

The *at-most-one* color clause states that each vertex can be colored by at most one color, and it's encoded as $(\neg x_{v,i} \vee \neg x_{v,j})$, with $v \in V$ and $i, j \in C$. The *at-least-one* parent relation variable states that, for each combination of a color and a label, exactly one parent relation variable must be true, and it's encoded as $(\bigwedge_{j \in C} y_{a,i,j})$ for all $a \in L$ and $i \in C$. The third set of redundant variables state that, when a parent relation is set, and some vertices have the source color, then all child nodes should have the target color. It is expressed as $(\neg y_{l(v),i,j} \vee \neg x_{p(v),i} \vee x_{v,j})$, for $v \in V$ and $i, j \in C$. The last set of clauses consists of adding all edges that are not covered by the *accepting color clauses*, and it's encoded as $(\neg x_{v,i} \vee \neg x_{w,i})$, for $i \in C$ and $(v, w) \in E$, where E is the set of all edges. Although these clauses are redundant, they provide some additional knowledge about the problem to the SAT solver. All constraints are summarized in Table 1.

The general algorithm is comprised of the following steps:

1. Given an APTA, build the related consistency graph.

2. Find the largest clique L (set of vertices) in the consistency graph.
3. Initialize the set of colors C in such a way $|C| = |L|$.
4. Construct a CNF by translating the APTA based on C and SBPs on L.
5. Use a SAT solver on the formula found in the previous step.
6. If the formula is unsatisfiable then add a color to C and return to step 4..
7. Return the DFA found in step 5..

2.5 Extracting the solution

Let's suppose that a CNF formula associated to an identification problem is satisfiable with C colors. This means that any model (any solution) of this formula represents a valid DFA with C states that solves the associated problem. To extract the automaton we can:

1. Create a new set of states, Q, such that $|Q| = C$.
2. Mark each state, q_i, as final if z_i is true in the model.
3. Choose as initial state q_i, where i is the color assigned to the initial state in the original APTA ($i \in \{1, \ldots, C\} : x_{v_0,i} \rightarrow$ true, with v_0 being the index of the initial state of the APTA).
4. For every variable $y_{a,i,j}$, true in the model, add a new arc with label "a" connecting node q_i to q_j.

The CNF associated to the example in Figure 2 is satisfiable with $C = 3$ colors. One possible solution is the following (just the true variables are listed):

$$z_0, z_1$$
$$x_{0,1}, x_{1,1}, x_{2,1}, x_{3,1}, x_{4,0}, x_{5,2}$$
$$y_{a,1,1}, y_{b,1,0}, y_{a,0,2}, y_{b,0,2}, y_{a,2,1}, y_{b,2,0}$$

Figure 6 shows the same APTA and constraints graph, with the solution of the coloring problem applied. This is not the only solution, though, because any of the three colors can be used for q_0, since it is a disconnected node in the constraints graph.

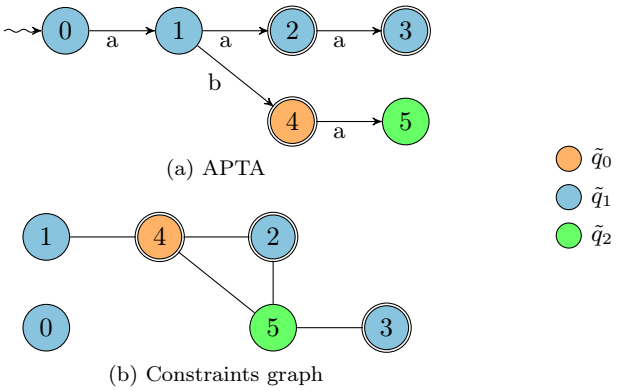
The DFA extracted from this solution is the same that we've seen in Figure 1. We can now check that the automaton is consistent with the input traces: $S^+ = \{\texttt{aa}, \texttt{aaa}, \texttt{ab}\}$ and $S^- = \{\texttt{aba}\}$. So, it is a solution of this simple problem.

3 Implementation

This part describes how the program has been designed and how it works. As we will see, most objects find a direct correspondence with concepts from Section 2. The software is written in Java and it is organized in the following packages:

cnf Classes for handling boolean expressions in conjunctive normal form.

automata DFA, APTA, and graphs in general.



(a) APTA

(b) Constraints graph

Figure 6: A solution to the coloring problem associated to Figure 2. Each color is an identified state: $\tilde{Q} = \{\tilde{q}_0, \tilde{q}_1, \tilde{q}_2\}$.

identification The main logic of the program: the DFA identification problem is solved here.

Every class and method has been documented using JavaDoc comments, with the hope that every part of the program will show a clear iterface. This document, however, is important to understand the general logic of the algorithm, which might not be evident just looking at each component.

3.1 CNF

The `CNF` package deals with the management of the CNF side of the algorithm and with the translation of the final formula to the format that can be accepted and read by the chosen solver.

The `Variable` class implements a generic Boolean variable. On its construction, an `index` and an `assignment` fields are initialized. An index is a unique string identifier, like `x_4,6` or `y_sendfine,1,3`, while the assignment is the value of the variable, so true or false. The class also implements some standard methods useful for handling variables, like `getIndex()`, `isTrue()` or `toString()`.

The `Clause` class implements a clause of a CNF formula, hence a disjunction of variables. The class contains two fields `positiveVariables` and `negatedVariables` that represent the sets of variables in the clause that appear positive or negated respectively.

The `Formula` class implements a CNF formula, which is a conjunction of clauses. Its only field, `clauseList`, represents the list of the clauses in the formula.

The `DimacsSaver` class is used to translate a `Formula` object to a text file in the Dimacs format. The Dimacs format is widely accepted as the standard format for boolean formulas in CNF. The class is centered around the public method `saveToDimacsFile(File file)` that, as the name tells, take the `File` *file* and saves it to the Dimacs format. To do so, it uses other private methods. First is run the method `gatherInformations()`, which sets the internal members of the class. A Dimacs file can start with some line of comments, each starting with a `c` character, then continues reporting the number of variables and clauses on the

line defined by `p cnf`. This initial section of the file is printed by the method `getPreamble()`. Next, are listed all the clauses, with positive numbers, denoting the indices of the corresponding variables, and negative numbers to indicate the negated variables. A `0` denotes the end of the single clause. This section is completed by the method `clauseRepresentation()`, which goes through all the clauses and variables in our formula and translates them to the Dimacs format.

3.2 Automata

Automata are the fundamental abstract models used in this program. We'll need to create, extend, run and visualize both APTAs and DFAs. There are many automata libraries available on the web. However, we didn't use them, because some of them are not general enough to handle APTAs, DFAs, and arbitrary transition labels in a consistent way. Still, the most general ones, have necessarily a more complex structure.

This package is intended to provide just the functionality we need, reliably. The most important classes are `APTA`, `DFA`, `DFABuilder` and `LatexSaver`. Users of these classes can skip section 3.2.1.

3.2.1 Graphs

As we've seen from the first examples, an automaton can be represented as a directed graph with labelled arcs with some specific additions. So, we decided to create a base class that implements the common logic of the child classes. This base class is called `AbstractGraph`. Similarly, any class in this package that implements a node object inherits from `AbstractNode`.

Both `AbstractGraph` and `AbstractNode` are abstract classes. They can't be directly instantiated, because they are meant to be used through inheritance by some concrete classes, even outside this package.

A graph can be represented in different ways. Thanks to Java Garbage Collection there's no need to remove unreachable objects. So, the linked structure becomes the most natural representation for a graph. Each `AbstractNode` instance has two fields: the numeric ID and a Map that associates labels to nodes. These maps contain the arcs leaving each node and, together, they form the whole graph.

In principle, we could store a minimal set of nodes from which any other node can be reached. But, in the automata we're interested in, it is sufficient to keep the initial node to reach any other. The initial node is stored in the field `firstNode`, inside `AbstractGraph`.

Here's how these two classes are declared:

```
1  public abstract class AbstractNode
2      <LabelT, NodeT extends AbstractNode<LabelT,NodeT>> {
3    /* ... */
4  }
5
6  public abstract class AbstractGraph
7      <LabelT, NodeT extends AbstractNode<LabelT,NodeT>>
8      implements Iterable<NodeT> {
9    /* ... */
10 }
```

We just need to discuss line 2 and 7, which declares the same generic parameters. `LabelT` is the generic type of the transition labels and `NodeT` is the class of each node in the graph.

Using a generic type for the labels is a clear advantage, because any class can be used: Strings, Characters, custom sets of symbols, complex structures. Since they are used as atomic objects, they just need to provide the `equals()` method, which every object inherits from `Object`, in Java (`hashCode()` is used too, because every Map is implemented with `HashMap`).

`NodeT` has been introduced to solve a problem with inheritance and linked structures. Let's suppose that, from a node object which extends `AbstractNode`, we want to connect or reach another node. What type should the connected node be? Inheritance suggests to use the base class. However, this would imply frequent downcasts to the derived class in order to use the extended functionality. At the same time, every node to connect should be of the right subclass to always ensure safe casts. Generics can solve this issue. In fact, with `NodeT`, we represent the type of all connected nodes. Now, this type can be used as we expect:

```
1 // In AbstractNode
2 public NodeT followArc(LabelT label)           // reach
3 public void addArc(LabelT label, NodeT node)   // connect
```

These functions are always type-safe (thanks to the exact node type), they don't require downcasts and they expose a more precise interface. Despite the unusual class definitions, it's easy to use them, as we will see in section 3.2.2 and 3.2.3.

Subclasses of `AbstractGraph` can create new nodes by calling `newNode()`:

```
1 // In AbstractGraph
2 abstract protected NodeT newNodeObj(int id);
3
4 protected NodeT newNode() {
5   return newNodeObj(nextFreeId++);
6 }
```

They just need to implement `newNodeObj()`, in order to return an object of the correct type. In the graph, every ID is unique but they are not guaranteed to be consecutive numbers.

`DepthPreIterator` is a nested class providing the default iterator for every graph. It follows a depth-first visit, returning the nodes in pre-order. Finally, a method we will frequently use is `followPath()`. It returns the node we reach after an iterative application of the transition function for every symbol in the input sequence.

As we will see, the models defined in this package can be exported and visualized. A first way to do so, is calling `saveDotFile()`. This function saves any `AbstractGraph` in a text file using DOT format. This format is a plain text file that can represent generic graphs. Many automata library accepts DOT files in input. Furthermore, DOT files can be visualized with tools such as GraphViz.

The graph we save in this file is a `digraph`, which stands for "directed graph". The first node has a single incoming edge, coming from an invisible fake node, called `init`. By default, `AbstractNode`s are represented as circles. However, any subclass of `AbstractNode` can personalize its own (DOT file) options file by overriding `dotNodeOptions()`. For example, a DFA node class would return `"[shape=doublecircle]"` for final nodes.

3.2.2 APTA

This class implements the structure that has been introduced in section 2.2. It is defined as:

```
1  // The graph
2  public class APTA<LabelT>
3      extends AbstractGraph<LabelT,APTA.ANode<LabelT>>
4      implements Automaton<LabelT>, LatexPrintableGraph {
5    /* ... */
6
7    // The node
8    public static class ANode<LabelT>
9        extends AbstractNode<LabelT,ANode<LabelT>> {
10     /* ... */
11   }
12 }
```

The main purpose of this portion of code is to show how to inherit from the abstract classes: we just need to specify that an `APTA` is a graph of `ANode`s and that `ANode` can connect to nodes of the same class.

At line 4, we see two interfaces. `LatexPrintableGraph` will be explained in section 3.2.4. The `Automaton` interface defines a function, `parseSequence()`, which accepts a list of symbols and returns the boolean response of whether the automaton accepts the sequence or not.

An `ANode` object has a field of Enum type `Response`, that can assume one of three values, {accept, reject, unknown}, to state which set this node belongs to (Q^+, Q^- or $Q^?$).

An `APTA` is instantiated with the default constructor. Initially, it will contain just the initial state. Then, it can be extended with these two functions:

```
1  public void acceptSequence(List<LabelT> sequence)
2  public void rejectSequence(List<LabelT> sequence)
```

They use the current object to parse the `sequence`. Then, they extend the tree with new transitions, if needed. Finally, they the response of the last node is set to 'accept' or 'reject' respectively. Instead, it's not possible to delete nodes from this graph.

`APTA` implements `Iterable<APTA.ANode<LabelT>>` through its superclass.

3.2.3 DFA

The `DFA` class implements section 2.1. It's declaration strictly follows the code we've seen for the APTA. What changes is that the graph class is `DFA<LabelT>` and the node is `DFA.DNode<LabelT>`.

The basic functionality of a DFA is provided by `AbstractGraph`, already. We just added a Boolean flag in `DNode` to store whether a node is final. Also, `DFA` implements `Automaton<LabelT>` with the function `parseSequence()`.

It's possible to iterate through the nodes of this graph. However, `DFA` doesn't have public methods to easily change the structure of the automaton. To create new DFAs, we should use the `DFABuilder` class. The most important functions in its iterface are:

```
1  public class DFABuilder<LabelT> {
```

(a) Initial state 0 (b) Initial state 2

Figure 7: Two DFAs built from the same graph but with different initial states. Calling `setInitialState(0)` creates 7a; `setInitialState(50)` gives 7b (50 is the identifier of state 2).

```
2
3     public void setFinalState(int state)
4     public void setInitialState(int state)
5     public void newArc(int parent, LabelT label, int child)
6   }
```

Every new `DFABuilder`, internally contains an empty `DFA`. The above functions can be used to modify and extend the graph. When the automaton is complete, we can call `getDFA()` to extract the automaton built.

As we can see from the parameters, states are identified by a numeric ID. This is convenient because there's no need to declare states in advance: every time a new integer is used, a new node is created. For example, if the first instruction is `newArc(2, 'a', 3)`, the 'builder' creates two new nodes connected by a directed arc. `setInitialState(2)` would refer to the first node, as we expect. However, these numbers are relevant just when buiding the graph, the returned DFA would use it's internal IDs (states of a `DFA` use increasing numbers, starting from 0). The order in which those functions are called is not relevant.

`setInitialState()` is important and it should be called for every DFA. For example, let's suppose we execute:

```
1   DFABuilder<Character> builder = new DFABuilder<>();
2   builder.newArc(0, 'a', 1);
3   builder.newArc(1, 'b', 50);
4   builder.newArc(50, 'c', 1);
5   builder.setFinalState(50);
```

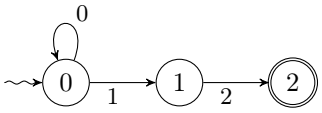
Number 50 would correspond to node 2. What language would this automaton accept? It depends on the initial state. Figure 7 shows two possibilities. As we can see, the automata changes dramatically. Furthermore, in 7b, state 0 becomes unreachable, so it won't be part of the automaton returned by `getDFA()`.

3.2.4 Visualization

This section shows how to visualize the models we've defined. What we do is to use LaTeX for displaying graphs. In fact, almost every figure in this document has been generated this way. It is sufficient to pass our model to `saveLatexFile()`:

```
1   public final class LatexSaver {
2     public static boolean saveLatexFile(
3       LatexPrintableGraph graph, File texFile, double spaceCm)
4   }
```

It will save a text file at the path given by `texFile`. Compiled with LaTeX, this files will create a representation of `graph` (`spaceCm` is just an horizontal spac-



Transitions

0 very

1 complex | many | long

2 arc labels | transitions

Figure 8: A DFA with the legend of the abbreviations.

ing; see the documentation). `DFA`, `APTA` and `ConstraintsGraph` all implement `LatexPrintableGraph`.

`LatexSaver` uses a `tikzpicture` to represent the graph (TikZ is a powerful graphics library for TEX). It also defines a common preamble for all graphs, and some keywords to be used as styles. Examples of styles are 'accept' and 'reject' for nodes, and 'backward' and 'self loop' for arcs. A `LatexPrintableGraph` has a function, `getLatexGraphRepresentation()`, which retuns TikZ code that represents the graph (actually it is interpreted as body of the `\graph` command). For example, this code:

```
1 [accept] ->
0 [accept,>"b"] ->
2 [>"a | b"]
2 -> [clear >, "b", backward] 0,
2 -> [clear >, "a", backward] 1,
1 -> [clear >, "a", self loop] 1
```

produces Figure 1.

In many real cases, the identified DFA might be complex, because of a large number of transitions. To partially solve this issue, we can call `dfa.useLegend()` with a true argument. So, when passed to `LatexSaver`, `dfa` will produce the graph and a legend. The legend lists all the abbreviations (integers) of the transitions, with their meaning in terms of the real labels. See Figure 8, for an example.

Still, this tool is not perfect, because arcs could overlap. Another possibility is to export `DFA`s and `APTA`s with `saveDotFile()`, then visualize them with GraphViz.

3.3 Identification

This package contains two classes, `ProblemEncoding` and `Solver`, that realize the global logic of the algorithm that we presented in section 2.4. The other objects that we find here abstract some concepts, that can simplify the process.

3.3.1 ConstraintsGraph

As we know, a constraints graph of the *DFA Identification* problem is an undirected graph whose arcs connects uncompatible states. The `ConstrantsGraph` class follows the structures of the other graphs in the automata package. However, it's use is restricted to the identification problem.

To create a new graph that represents the constraints associated to an APTA, we just need to pass the domain as argument of the constructor:

`new ConstraintsGraph(apta)`. The constructor starts an initialization phase in which the graph is created. Consequently, every future query on the graph can be answered more efficiently. This class only supports `APTA<String>`. Still, this is not a limitation, since every structure can be converted into a meaningful string representation that preserves the equality test.

The public methods of a `ConstraintsGraph` return quantities that, together, define the particular instance of the identification problem. Let's see the interface more in detail. First, we certainly need the set of all constraints. The method

```
public Set<Pair<CNode,CNode>> constraints()
```

iterates over the graph and returns the set of all arcs. Edges are represented as pairs of nodes and `CNode` is just the node object for this graph. A `CNode` is similar to an `APTA.ANode`. Even though the response field is not used by this model, it helps to visualize the nodes more clearly.

The graph can be iterated, because it `implements Iterable<CNode>`, and it can be visualized, because it is a `LatexPrintableGraph`. Two public functions, `getAcceptingNodes()` and `getRejectingNodes()`, return the collections of all nodes with the two responses (in constant time). `allLabels()` returns the set of all strings that appears in the input domain. This will be the input alphabet, Σ, of the identified DFA. A function, `getClique()`, returns a set of nodes that belong to a large clique.

We know the meaning of the edges in this graph, but how is the graph built? First, every positive node is connected with every negative one. These arcs connects the directly incompatible nodes, because they have a different response. These edges, together with the requirements of a deterministic DFA, form a minimal set of constraints (remember: the determinization constraints can't be represented in this class, because they involve more than two variables).

Then, we need to add all arcs created in the recursive step of the definition of compatibility (section 2.3). Let's denote with q_i a state of the APTA and with n_i the related node in the constraints graph. To compute the missing edges, we check whether `isAConsistentMerge`(q_i, q_j) fails:

1. If n_i is connected to n_j, the test fails.
2. Given the set of the common labels, $L = L_1 \cap L_2$ (where L_i is the set of all transitions leaving state q_i), if any of these fail

   $$\texttt{isAConsistentMerge}(\delta(q_i, l), \delta(q_j, l)) \qquad \forall l \in L$$

   the test fails.
3. If 1 and 2 succeeded, the test succeed if and only if n_i is not connected to any of the nodes merged with n_j during step 2, and the same holds for n_j.

If `isAConsistentMerge`(q_i, q_j) fails, the arc (n_i, n_j) is added to the constraints graph.

3.3.2 Encoding variables

Section 2.5 showed that every variable has a meaning, in terms of the extracted DFA. Each variable (mostly the true ones) describes a portion of the automaton.

So, it seems reasonable to add the meaning to the variables themselves and use subclasses to represent their contribution to the final DFA.

Therefore, instead of simple Boolean variable `cnf.Variable`, we use subclasses of `EncodingVariable`:

```
public abstract class EncodingVariable extends Variable {

  protected abstract void actionOnDFA(DFABuilder<String> d);

  public void extendDFA(DFABuilder<String> dfa) {
    if (isTrue()) {
      actionOnDFA(dfa);
    }
  }
}
```

All variables of the identification problem extend `EncodingVariable`. Each of them can simply override `actionOnDFA` to personalize its behaviour.

A `FinalVariable` extends `EncodingVariable` and it is used to represent any z_i. Its action of the DFA is to mark state i as final (see `DFABuilder` in section 3.2.3). A `ColorVariable`, instead, does nothing, because it simply associates states of the APTA to states of the identified DFA. It can represent any $x_{v,i}$. However, the color of the initial state q_{v_0} of the APTA is important. `InitialColorVariable` represent each of these. Its effect is to mark state v_0 as initial. Last group of variables is the parent relation, $y_{a,i,j}$. A `ParentVariable` adds a new arc, labelled with a, connecting the states i and j, to the final DFA.

3.3.3 Solver

The `ProblemEncoding` class deals with the translation from the graph coloring problem to the SAT problem, presenting us the final CNF that will be passed to the SAT solver. Various parameters are needed when initializing an instance of `ProblemEncoding`:

```
  public ProblemEncoding(APTA<String> apta,
                         ConstraintsGraph cg,
                         Set<CNode> clique,
                         int colors)
```

the APTA, built from a number of accepted and rejected traces; the related constraints graph, as defined in the previously presented class; the biggest clique in that same graph, used, as already seen, to speed up the whole process by skipping some trivially unsatisfiable combinations of colors; and the number of colors to test. They are all passed as parameters to the constructor of this class. In the constructor, once these inputs are set, we also initialize the sets of Boolean variables for the problem. The three sets of variables $x_{v,i}$, $y_{a,i,j}$ and z_i are all created as instances of the three different classes `ColorVariable`, `ParentVariable` and `FinalVariable`, extending the class `EncodingVariable`, described in the previous paragraph.

The methods of the class can then be used to fill our final formula with all the clauses specified in Table 1. The methods which generate each clause are all `private`, but can be accessed with the two public methods `generateClauses()` and `generateRedundantClauses()`:

```
1   public void generateClauses() {
2     initCliqueVar();
3     atLeastOneColor();
4     accRejNotSameColor();
5     parentRelationWhenColor();
6     parentAtMostOneColor();
7     parentAtLeastOneColor();
8   }
9
10  public void generateRedundantClauses() {
11    atMostOneColor();
12    parentForceVertex();
13    determinConflicts();
14  }
```

In theory we wanted the `generateClauses()` method to deal with the generation of the first four clauses from Table 1, and the `generateRedundantClauses()` method to deal with the last four (redundant) clauses. In practice, during testing, we made small adjustments to fit our needs.

The `generateClauses()` method incorporates the four methods that generate the necessary clauses but not only. The method `initCliqueVar()` is used to create both the clauses that assign a different color to each vertex in the clique, and the clauses that specify if each of these colors correspond to a final state or not. The method `parentAtLeastOneColor()` is used to generate a redundant clause (that specify that each parent relation must target at least one color), but it's nonetheless included in the `generateClauses()` method because we always want to create a complete DFA. The `generateRedundantClauses()` method then ends up incorporating only the remaining three methods. Once all the wanted clauses have been generated, we can then use the `getEncoding()` method to obtain the final encoding, i.e. the wanted CNF formula.

The `Solver` class is used to call the SAT solver and to extract the solution, starting from the formula obtained with the `ProblemEncoding` class. As our SAT solver of choice we used Sat4j ([3]), a java library for solving boolean satisfaction and optimization problems. The method `extractSolution()` represents the core of the entire class. It takes as argument the `Formula` obtained from the `ProblemEncoding` class and returns a `List` of `EncodingVariable`s with their boolean values as found by the solver, if the formula was satisfiable. Otherwise, it returns `null`. In order to pass the formula to Sat4j, it is first translated in the Dimacs format through of our `DimacsSaver` class. A map

```
Map<Integer,Variable> mapToVar = saver.idToVarsMap()
```

will be later used to make the backwards step from the solver format to a list of our `EncodingVariable`s. From the solution we got, we can then also create a DFA with the help of the previously mentioned `DFABuilder` class, using the method `extractNewDFA`. This method takes the solution found and just creates a new DFA and extends it with the true variables in the solution.

3.4 Utils

The `Util` package is a small support package containing only two classes.

`Pair` is a very small class with a really straightforward use as a container of two elements. It gets used in various places of the project, mainly to represent

edges in graphs.

The `TraceManager` class is used to build the APTA. It is used to load files in the XES format and to extract from those the necessary traces. The XES standard (eXtensible Event Stream) defines a grammar for a tag-based language whose aim is to provide designers of information systems with a unified and extensible methodology for capturing systems behaviors by means of event logs and event streams defined in the XES standard. To handle this particular format we used the OpenXES library ([4]), which is the reference implementation of the first XES standard. Each XES file is composed by one or more *logs*, which in turn are composed by many traces, which are lists of labels, i.e. strings. Each XES file contains good or bad traces, accepted or not, as described by the file name (if a file name ends with "OK" or not). The method

```
public static APTA<String> parseTracesFiles(File dir)
```

is used to access all the *.xes* files in a specified directory to get all the traces in them and build the APTA.

4 How to use the software

Our project is publicly available for download on the platform GitHub at `https://github.com/cipollone/trace-dfa`. The first step is to clone the repository and place it in a folder of choice:

```
git clone https://github.com/cipollone/trace-dfa
cd trace-dfa/
git checkout release
```

Then, some dependencies are needed. Ant, OpenXES, Guava and Sat4j must be downloaded and put (they come as Jars) in a directory named `lib` in the top level directory of the project (the one containing `build.xml`). The command `ant` is used to run the software, and, to obtain the documentation, use the command `ant doc`, which will generate it in a directory named `doc`:

```
ant             # runs the software
ant doc         # generates the doc
```

The input of the program is composed of two sets of sequences: one for learning the DFA and one for testing. All sequences must be in XES format. By default, the program will use the directory `traces/train` for learning and `traces/test` for testing. However one may specify arbitrary paths when running the program, using options:

```
ant -Dtrain=any_train_dir -Dtest=any_test_dir
```

Each log file is a sequence of traces in XES format composed of events. Only the parameter *concept:name* of each event is used to identify the transitions. Files containing traces to be accepted by the DFA need to have "OK" added in the filename of the XES file. Sequences to reject need no modification. Test traces can be used to check that the DFA respects the constraints (in this case one can use the same traces both for learning and for testing), or to see how the solution generalizes to unseen sequences.

If the program is distributed as a single Jar file, it will contain all dependencies in it. One may run it with:

```
java -jar TraceDFA.jar traces/train/ traces/test/
```

In this case, the input traces need to be always passed as arguments.

As result, the program will create a directory `output` some files:

dfa.tex is the main output of the program. This is the LaTeX representation of the extracted Finite State Automaton which is consistent with the given traces.

apta.tex is the APTA that the algorithm uses internally to represent the input traces (to learn). In the case of many traces, this file cannot be compiled, due to space limitation on the page.

constraints.tex is the graph of constraints that the algorithm internally uses to represent the constraints in the coloring problem. This also could exceed TeX limitations.

dfa.dot is the same DFA extracted by the program, but saved in the well-known *dot* graph description language.

5 Results

5.1 Dataset

We tested our implementation on a set of five XES logs, one containing accepted traces and four containing rejected traces. Log files differ for the number of traces in them and for the number of errors in each trace to reject, for a total of 49 accepted traces and 1152 rejected traces. The logs were extracted from executions of a real process: a local Italian management system of road-traffic violations.

The road-traffic violations are handled by the Italian police using an information system that was specifically built for this purpose by an external company. In Figure 9 we can see a model, a Petri net, which shows how the Italian police ideally handles road-traffic violations. A new process instance is created when an offender is fined, so when the transition *CreateFine* is fired. From this point, the offender can pay the fine (*Payment*) in different moments of the process: right when the fine is created (*CreateFine*), after the fine notification is sent to the offender (*SendFine*), when the offender receives the notification (*Notification*). This fact explains the presence of *Inv1*, *Inv2* and *Inv3* transitions: "inv" stands for invisible, and from any point the invisible transitions bring the process to its end. If the whole amount of the fine is paid, an invisible transition is fired and the fine management is closed. Once the notification is received, if, after 180 days, the fine results still not payed, a penalty is added to the amount (*AddPenalty*). The offender can also try to appeal against the fine through a judge (*AppealtoJudge*) and/or the prefecture (*AppealtoPrefecture*) and, if successful, transitions *Inv4* or *Inv6* are fired and the process ends. If unsuccessful, transitions *Inv5* or *ReceiveResults* are fired and the process continues. If the offender perseveres in not paying, eventually the process ends as the case is handed over to credit collection (*SendforCreditCollection*).

Correct traces (traces that should be accepted by the DFA) are those which follow the correct procedure and terminate in the 'End' place of the Petri Net.

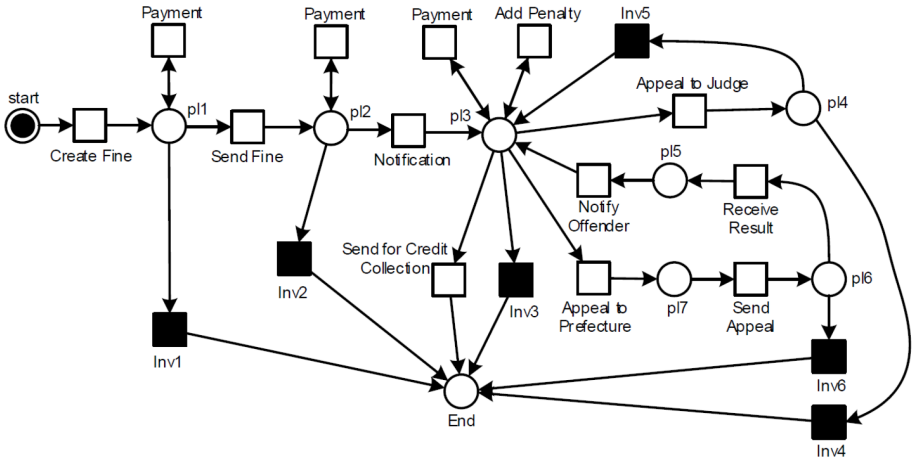


Figure 9: The Petri net describing the road-traffic violation process model

Wrong traces (sequences that should be rejected by the DFA) present transitions in the wrong places or missing entirely. With this dataset, the DFA we will identify should be able to tell whether a given sequence terminates correctly or not.

5.2 Outcomes

On our dataset, the algorithm we implemented showed a very good performance on different machines. We tested it both on a high-end machine with an Intel Core i7, 2.80 GHz with 16 Gb of memory and on an older model with an Intel Core i5, 2.67 GHz with 8 Gb of memory, and both completed each test in just a few seconds. The algorithm was first ran without the redundant clauses in the SAT translation and then with them. Without the blocked clauses, it takes just 4 seconds for the newer machine to find a solution, and the older one takes only 10 seconds. The solution of our traces is a DFA with four states. Hence the time it takes to find it comprises: the time to find the largest clique in the constraint graph; the time spent for each of the failed attempts because of the insufficient number of colors (states); the time needed for the last successful attempt. In our case, the algorithm first finds a clique of two vertex in the graph, then it starts trying to solve the SAT instance, failing with 2 and 3 colors, and finally succeeding with 4.

Figure 10 shows one of the outcomes of our algorithm. As we mentioned, the solution is a DFA with 4 states. This solution has 3 final states. The arcs show all the transitions between states and each of them is numbered. Below the graph, a legend displays the meaning of each number as a set of different labels, used to enhance the readability of the DFA. The DFA obtained with our algorithm is always complete.

Unlike what is written in the paper, adding the redundant clauses didn't improve the performance of the algorithm but, on the contrary, it made them worse. Running the algorithm with all the clauses took 6 seconds for the newer machine and 25 for the older one. Our hypothesis is that the decrease in performance could be the consequence of the peculiar heuristic of the solver we

employed, Sat4j, which is different from the one used in the paper, which is named *picosat*.

There is another important property of our algorithm, that we noticed during testing. In Figure 10, the DFA has 3 final states: q_2, q_3 and q_0 and the initial state is q_2. When we ran again the algorithm, the outcome was the one in Figure 11. In this case, the solution presents some differences. The number of states is the same, it is the minimal number of states for the given traces. However, most arcs are changed and even the number of final states is different, as, in this solution, there are 2 of them, q_2 and q_1. For every run of the algorithm, the solution presents a slightly different structure, always with constant number of states.

To understand this behaviour, we should notice that our algorithm generates a CNF formula as a unordered set of clauses (which in turn are unordered sets of variables), hence the formula is always different. Many parts of our algorithm use HashSets data structures, in fact. This is, with a good probability, the cause of the behaviour, but we can't also rule out the participation of the solver and its methods when handling the formula. A different input could drive the solver toward a different assignment, because, if the number of color is correct, there are many solutions to our problem.

What is important is that any output is a DFA, consistent with the input. In fact, as a final test, we always run the DFA to check that it always recognizes each of the correct traces and rejects all the wrong ones. Still, this variability is an useful hint of some lack of constraints in our problem and it can be used at our advantage.

To better get an intuition of these changes, we computed the language recognized by one DFA but not the other: $\mathcal{L}_\setminus = \mathcal{L}(\mathcal{A}_a) \setminus \mathcal{L}(\mathcal{A}_b)$. What we found is that the automaton representing $\mathcal{L}_\setminus$ is huge, so it's not useful to report it here. Probably, this happens because two solutions assign different responses to a *sparse* set of strings. Furthermore, any sequence in $\mathcal{L}_\setminus$ is unconstrained. So, it could be useful to add (short) sequences from this set, to the input dataset, in order to improve or guide the solution further.

These consideration are important, because they inform us about the goodness of our dataset with respect to our goal. For example, the interpretation of the extracted DFA from our input traces, is an automaton that is able to detect when a fine is completely paid. As we see from Figure 10, both `{createfine}` and `{createfine, addpenalty}` lead to final states (as we can see from the Petri net, this is wrong). So, we certainly would need to add these sequences to the input dataset.

6 Conclusion

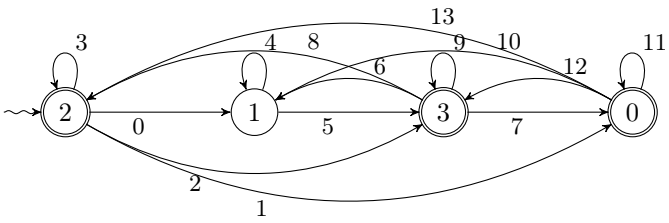
The problem we were able to solve is *learning* the structure of a Deterministic Finite Automaton that recognizes or rejects a given set of sequences in input. This problem is known to be NP-complete. Nonetheless, what we compute is the *minimum* DFA that is able to correctly parse the examples we provide.

Our work has been to implement, in a Java software, the algorithm described in paper [1]. The SAT solver was able to solve the DFA identification problem, just because this paper proposed a compact way to translate the constraints in a CNF formula. In fact, the Boolean variables that express the parent relation help

in keeping the formula small enough, to be able to solve interesting problems. The same algorithm can also be applied to partially identified solutions of larger instances.

The software we realized implements the complete algorithm that the paper describes. Also, our purpose was to keep the program as simple as possible, both for users and programmers. Some classes make few assumptions on the specific domain. So, it might be useful to import or extend them. We also provide a JavaDoc documentation, which more precisely describes the API.

As we've seen, the language recognized by a DFA can be converted to a regular expression. So, learning a DFA is like finding a "small" grammar that includes the examples provided. Therefore, what we produce is a generalization of the visible samples. However, this tool is not limited to language recognizers. For example, with the appropriate class, labels could represent input-output relations. Still, we're investigating how this tool can be generalized to other applications that falls in the same class as the DFA identification problem.



Transitions

0 sendfine | insertdateappealtoprefecture | appealtojudge | inv1) | payment | inv3) | inv2) | addpenalty | inv5) | inv4) | sendforcreditcollection | inv6)

1 createfine | notifyresultappealtooffender

2 sendappealtoprefecture | receiveresultappealfromprefecture

3 insertfinenotification

4 sendfine | sendappealtoprefecture | insertdateappealtoprefecture | receiveresultappealfromprefecture | createfine | appealtojudge | inv1) | inv3) | payment | addpenalty | inv2) | sendforcreditcollection | inv4) | insertfinenotification | notifyresultappealtooffender | inv6)

5 inv5)

6 sendfine | sendappealtoprefecture | insertdateappealtoprefecture | appealtojudge | inv1) | inv3) | addpenalty | sendforcreditcollection

7 createfine | inv5) | inv4) | insertfinenotification

8 receiveresultappealfromprefecture | notifyresultappealtooffender

9 payment | inv2) | inv6)

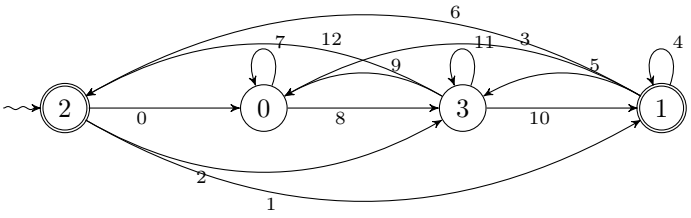
10 sendappealtoprefecture | inv2) | insertfinenotification | inv6)

11 createfine | inv1) | payment | inv3) | addpenalty | inv5)

12 sendfine | receiveresultappealfromprefecture | appealtojudge | inv4)

13 insertdateappealtoprefecture | sendforcreditcollection | notifyresultappealtooffender

Figure 10: Example of DFA found by the algorithm



Transitions

0 sendfine | insertdateappealtoprefecture | payment | inv2) | sendforcreditcollection | notifyresultappealtooffender

1 sendappealtoprefecture | createfine | inv5) | inv4) | inv6)

2 receiveresultappealfromprefecture | appealtojudge | inv1) | inv3) | addpenalty | insertfinenotification

3 sendappealtoprefecture | receiveresultappealfromprefecture | createfine | inv6)

4 inv1) | inv3) | payment | addpenalty | inv5) | inv4)

5 sendfine | insertdateappealtoprefecture | inv2) | notifyresultappealtooffender

6 appealtojudge | sendforcreditcollection | insertfinenotification

7 sendfine | sendappealtoprefecture | insertdateappealtoprefecture | receiveresultappealfromprefecture | createfine | appealtojudge | inv1) | payment | inv3) | addpenalty | inv5) | sendforcreditcollection | inv4) | insertfinenotification | notifyresultappealtooffender | inv6)

8 inv2)

9 insertdateappealtoprefecture | receiveresultappealfromprefecture | appealtojudge | inv1) | inv3) | addpenalty | sendforcreditcollection | inv6)

10 inv2) | insertfinenotification | notifyresultappealtooffender

11 sendfine | payment | inv4)

12 sendappealtoprefecture | createfine | inv5)

Figure 11: Another example of a different DFA found with the same input

References

[1] Marijn J. H. Heule and Sicco Verwer. "Exact DFA Identification Using SAT Solvers". In: *Grammatical Inference: Theoretical Results and Applications*. Ed. by José M. Sempere and Pedro García. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 66–79. ISBN: 978-3-642-15488-1.

[2] Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. "Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm". In: *International Colloquium on Grammatical Inference*. Springer. 1998, pp. 1–12.

[3] Daniel Le Berre and Anne Parrain. "The SAT4J library, Release 2.2, System Description". In: *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), pp. 59–64. URL: `https://hal.archives-ouvertes.fr/hal-00868136`.

[4] Eric Verbeek. *OpenXES: Developer Guide*. 2014. URL: `http://www.xes-standard.org/openxes`.