A customised automata algorithm and toolkit for language learning and application

Ruoyu Wang and Guoqiang Li*

School of Software,
Shanghai Jiao Tong University,
Shanghai, China
Email: babyfish92@163.com
Email: li.g@sjtu.edu.cn
*Corresponding author

Jianwen Xiang

Wuhan University of Technology,
Hubei, China
Email: xiangjw@gmail.com

Hongming Cai

School of Software,
Shanghai Jiao Tong University,
Shanghai, China
Email: hmcai@sjtu.edu.cn

Abstract: Automata are abstract computing machines. They play a basic role in computability theory and programming language theory. More recently in data analytics, data automata have become a formal way to represent pipelines and workflows. However, in researches involved with automata, there are still situations where redundant work and ununified standards occur. In order to solve that problem, we propose a new toolkit: CAT, which provides a simple and unified framework for automaton construction and customisation. We adopted both structural and behavioural analysis in order to design the body structure. Several calculus algorithms are implemented according to the theoretical accomplishments and designed as overloaded operators. To test the correctness and performance of this toolkit, several bare automata were constructed and compared with ‘GREP’ in Ubuntu Linux. The result showed that CAT has realised most of the design purposes and presents a more illustrative way for writing codes of automata construction and calculation.

Keywords: automata; customise; C++; big data analytics; semantics; toolkit; L*; automata theory; DFA; NFA; PDA; regular language; context free language; Infer.net; framework.


Biographical notes: Ruoyu Wang received his BS degree from Shanghai Jiao Tong University, Shanghai, China in 2015, where he is currently working toward his Master degree in BASICS Lab (Laboratory for Basic Studies in Computing Science). His research interests include big data analytics and machine learning. From October 2015 to April 2016, he studied in NICTA (National ICT Australia) in Australia as an exchange student, working on big data analytics.

Guoqiang Li received his BS, MS, and PhD from Taiyuan University of Technology, Shanghai Jiao Tong University, and Japan Advanced Institute of Science and Technology in 2001, 2005, and 2008, respectively. He worked as a postdoctoral research fellow in the Graduate School of Information Science, Nagoya University, Japan, during 2008–2009, as an Assistant Professor in the School of Software, Shanghai Jiao Tong University, during 2009–2013, and as an academic visitor in the Department of Computer Science, University of Oxford during 2015–2016. He is currently an Associate Professor in School of Software, Shanghai Jiao Tong University. His research interests include formal verification, programming language theory and computational learning theory.

Jianwen Xiang received his PhD from Wuhan University and from Japan Advanced Institute of Science and Technology (JAIST) in 2004 and 2005, respectively. He is currently a Professor of the School of Computer Science and Technology of Wuhan University of Technology, and he was an Assistant Manager at Central Research Laboratories of NEC Corporation. His research interests include dependable computing, formal methods, and software engineering.
1 Introduction

Automata are abstract computing machines. They play a basic role in both theoretical and industrial fields (Hopcroft et al., 2006). After Turing had proposed his ingenious creation, the Turing Machine (Turing, 1936), various types of automata have been studied by computer scientists as substitutions of Turing Machine and put into practical use to build footstones in several popular tools. In computing theory, automata can be applied to study the computing ability of formal models. In industrial practice, automata are widely used to construct token scanners and syntactic analysers in compilers (Aho et al., 1986).

Another important purposes to study automata lay in data analysis and model checking. In order to maintain data semantics, structural approaches are introduced along with finite automata, which is formally named as ‘grammar induction (or grammatical inference)’ (De la Higuera, 2010). Angluin started a new approach, the ‘L*’ algorithm, to ‘learn’ data semantics via queries and comparisons. In this algorithm, data semantics can be expressed in the form of deterministic finite automata (DFA) (Angluin, 1987). In Maler and Mens (2014), symbolic DFA has been applied to condense the expressing methods in L* (Schwoon, 2002) also used this learning algorithm to check certain properties in a given system.

On the other hand, non-structural approaches analyse data classification instead of the transmission features. New types of data automata, such as register automata (Kaminski and Francez, 1994), class memory automata (Cotton-Barratt et al., 2015) and pebble automata (Neven et al., 2004), are created to compare various data sets. They modify and focus more on states, enduing extra ability upon states by establishing innovative components on them.

No matter what approach to take to maintain data semantics, automata are expected to be used to explore computability confronting infinite data stream (or big data). Demanded by checking, database (XML especially) and graph data analysis, automata on infinite data stream is becoming a new hotspot in computing science (Song and Wu, 2016; Segoufin, 2006). To help study on those fields, several research groups had developed powerful tools to support automata construction and computation, such as Automata (Veanes and Nikolaj, 2012) and Infer.net (Minka et al., 2012) of Microsoft Research. They both have relatively comprehensive support for automata computation and representation. For example, Automata is frequently used to construct automata from given regular expressions. However, when it comes to circumstances that no regular expressions can be provided, Automata is helpless. Moreover, several new approaches in data semantics maintenance (e.g. L*) requires no regular expression but constructing automata state by state. Also, current tools are heavy-weighted, requiring rich background knowledge and experienced skills to take advantage of them. Confronting problems such as ‘Pipeline Jungle’ (Sculley et al., 2014), there will be a large proportion of redundant cost to modify and manage systems built by those tools.

Motivated by all problems above, we designed and implemented a light-weighted tool named customised automata toolkit (CAT), which provides a simple and unified framework for automata construction and customisation. CAT integrates many operations among automata, such as intersection, union, subtraction, complement, minimisation, sampling and transformation. Due to its simplicity and flexibility, CAT can reduce not only the learning cost, but also communication cost, which made it a powerful tool in both researches and industry to customise transition systems in a more unified way. In this paper, we put it in detail the architecture of CAT and its functionality. We performed a correctness test and a performance case study to show the great practicability in application of CAT.

Section 2 gives related research works in the past decades. Section 3 introduces several concepts to help get a better understanding of CAT. Detailed arhitecture and design principles are introduced in Section 4. Section 5 shows some basic correctness tests. A new case study had been performed in Section 6. And in Section 7, both advantages and disadvantages are discussed.

2 Related work

After concept of artificial intelligence had been proposed, learning on regular systems had been firstly studied by Gold (1967). And Biermann had given an off-line learning algorithm (Biermann and Feldman, 1972). After that, online learning methods have been continuously studied by numerous researchers till now. Traktenbrot and
Barzdin (1973) proposed a first online learning algorithm, whereas it required a relatively strict assumption which is prefix-closeness. A modified algorithm, $L^*$, was proposed by Angluin (1987). $L^*$ was a huge success due to its fine properties:

1. It requires no strict assumption.
2. It generates a minimum DFA.

$L^*$ had been applied to neural networks, geometry, data mining, verification and many other fields (Alur et al., 2005; Cobleigh et al., 2003; Klarlund, 1994).

On the basis of $L^*$, Lang (1992) and Oncina and Garcia (1992) proposed a more efficient algorithm, though they have a trade-off that the result may not necessarily be minimal. Berg et al. (2005) and Hungar et al. (2003) also proposed some domain specific optimisations. Later on, $L^*$ had been extended to timed system (Grinchtein et al., 2005, 2004) and $\omega$-regular languages. However, how to extend $L^*$ to $\omega$-regular languages remained open for many years (Leucker, 2007). Besides Angluin, De la Higuera and Janodet (2004), Jayasrirani et al. (2012) and Saoudi and Yokomori (1993) tried their own solutions. And Klarlund (1994), Maler and Pnueli (1991) and Maler and Staiger (1997) provided footstones for learning on regular $\omega$-languages. With the help from previous theoretical achievements (Maler and Staiger, 1997; McNaughton, 1966), Angluin and Fisman (2014) finally proposed an extended algorithm, although it is not yet very efficient. Another extension on infinite data semantics is to extend $L^*$ on infinite alphabet (Maler and Mens, 2014). It applied concept of Symbolic automata, compressing expression of DFAs.

3 Preliminaries

Automata are abstract computing machines. They consume input and transfer among states. They can be categorised into various types according to their structure and behaviour. Moreover, what automata represent is a set of strings they accept, thus the calculus on automata is actually operation on set of strings. Several concepts on genres and operations of automaton are introduced here to help get a better understanding of what CAT is designed for.

Definition 3.1: Non-deterministic finite automata (NFA) are automata composed of the following five parts:

- $Q$: set of states.
- $\Sigma$: set of input characters.
- $\delta$: a transition function: $Q \times \Sigma \rightarrow 2^Q$. It takes a state and an input character as parameters and returns a set of states that it will transfer to.
- $q_0$: initial state.
- $F$: set of final states.

One NFA starts at $q_0$ to read input string character by character and transfers among all states in $Q$. If it eats up all input and is able to stop on some final state, then this input string is accepted by the NFA.

Definition 3.2: DFA are specialised NFAs. They are also composed by a quintet of $(Q, \Sigma, \delta, q_0, F)$, except that $\delta$ has a slightly different definition. In DFA, transition function is defined as a mapping of $Q \times \Sigma \rightarrow Q$, in which the result is deterministic.

Definition 3.3: Pushdown automata (PDA) are automata composed of the following seven parts:

- $Q$: set of states.
- $\Sigma$: set of input characters.
- $\Gamma$: set of stack symbols.
- $\delta$: a transition function: $Q \times \Sigma \times \Gamma \rightarrow 2^Q \times \Gamma^*$. It takes a state, an input character and a stack symbol as parameter and returns a set of states it will transfer to along with a string of stack symbols respectively.
- $q_0$: initial state.
- $Z_0$: initial stack symbol.
- $F$: set of final states.

Also, PDAs maintain a stack within them. One PDA starts at $q_0$ to read input string character by character. Each time it reads a input character, it will refer to the transition function to modify its current state together with the current state and top most symbol in the stack. If it eats up all input and is able to stop on some final state, then this input string is accepted by the PDA.

For PDAs, there is a slight different between deterministic and non-deterministic ones, as it is between DFAs and NFAs. Also, it has another accepting approach which is proved equivalent to that mentioned above: it also accepts one input string if it stops with an empty stack.

Definition 3.4: Symbolic automata is not a new type of automata. It is a concept of modifying the way one automaton reads input string. Conventionally, one automaton reads an input character and feeds it to $\delta$ as what it was. However, when the concept of Symbolism is adopted, one automaton will map the raw input character to a new one, i.e. a symbolic character. The mapping function is usually written as $\psi: \Sigma \rightarrow \Sigma$. And the bold font is used to denote symbolic components.

For instance, a symbolic DFA that accepts all valid variable names in C/C++ can be defined in Figure 1.

\[
\psi_{q_0}(c) = \begin{cases} 
\alpha & \text{if } c \in \{c | \text{ c is an ASCII character}\}, \\
\beta & \text{o.w.}
\end{cases}
\]

where $\Sigma = \{c | \text{ c is an ASCII character}\}$, $\Sigma = \{\alpha, \beta, \gamma, \delta, \theta\}$, $\psi = \{\psi_{q_0}, \psi_{q_1}, \psi_{q_2}\}$. The mapping function family are defined below:

\[
\psi_{q_0}(c) = \begin{cases} 
\alpha & \text{if } c \in \{A - Za - z\} \\
\beta & \text{o.w.}
\end{cases}
\]
Definition 3.5: Boolean operations include Intersection, Union, Subtraction and Complement, which are the same as they are on ordinary sets.

Definition 3.6: Concatenation is an operation that concatenates two strings, for example \( a \) and \( b \), end to end, written as \( a \cdot b \). And concatenation between two automata is defined as below:

Suppose for automata \( A \) and \( B \), their languages (sets of accepted strings) are \( L_A \) and \( L_B \) respectively, then their concatenation is set \( \{a \cdot b | a \in L_A, b \in L_B \} \).

Definition 3.7: Reverse on strings is an operation that reorder a string from end to beginning. For example, a string \( s = a_0 \cdot a_1 \cdot \ldots \cdot a_n \), its reverse is \( s^R = a_n \cdot a_{n-1} \cdot \ldots \cdot a_0 \). Reverse on an automaton is to apply reverse on all strings accepted by it.

Definition 3.8: Closure (aka. Kleene Closure) is an operation that gives a union of all self-concatenation of an automaton. I.e. for an automaton \( A \), its Kleene closure is \( A^* = \bigcup_{n=0}^{+\infty} A^n \), where \( A^0 = \{\epsilon\}, A^n = A^{n-1} \cdot A \).

Definition 3.9: Minimisation of an automaton \( A \) is to construct a new automaton of the same type which is equivalent to \( A \) and has a minimum number of states.

Definition 3.10: Transformation is to construct an equivalent automaton of a different type.

Definition 3.11: Sampling is to produce a string that is accepted by one automaton.

4 Architecture and principles

CAT is written in C++, based on OO programming method. It can be used to construct an automaton from scratch and simulate its execution. It can also be used to perform calculation on those of same type and transformation among different types.

4.1 Automaton hierarchy

Automata theory was developed after the seminal work of Alan Turing with his well-known Turing machine (Turing, 1936), when simpler abstract computing machines were studied by scientists. After generations, many new types of automaton had been developed and studied, usually established on the foundation of former ones. Thus, the hereditary structure of all types of automata becomes a tree-like hierarchy.

For example, the hierarchical relation among some basic types of automata listed below can be concluded into a diagram as Figure 2. Since the transition function in ‘RegularAutomata’ has a different definition with that in ‘PushDownAutomata’, we divide them into two different branches, though they both have their transition functions denoted as ‘\( \delta \)’.

- DFA = NFA = (\( Q, \Sigma, \delta, 0, F \))
- DSFA = NSFA = (\( Q, \Sigma, \Sigma, \psi, \delta, \delta, 0, F \)) (Maler and Mens, 2014)
- DPDA = NPDA = (\( Q, \Sigma, \Gamma, \delta, 0, Z_0, F \))
- DSPDA = NSPDPA = (\( Q, \Sigma, \Sigma, \psi, \Gamma, \delta, 0, F \)).
successors to inherit. Thus those ‘non-meaningful’ classes are marked abstract and painted in red, and others in black are entity classes that will actually be used to instantiate automaton entities. That becomes the core architecture of the entire toolkit.

**Figure 3** Hierarchy of some basic kind of automata (see online version for colours)

### 4.2 State by state construction

Data semantics displayed in semi-structured (XML documents) data and graph data requires state by state construction and modification. Regular expressions are useful and compressive in representing data semantics. However, it is too abstract to locate each component in an automata. Thus, we organise all states in an automaton to be a directed graph, together with the transition function inside each of the states. That graph can be just regarded as the transition graph of that automaton. Also, there will be a unique label allocated to each created state.

Since the transition function is integrated within states, the states shall vary with different types of automata. That is, the states used in different types of automata should have different structures and behaviours. Then, similar analysis can be applied on the states. As is shown in Figure 4.

**Figure 4** Hierarchy of states (see online version for colours)

To enhance the robustness of the transition system, some conventions must be observed by users:

- all states shall be well-defined on entire alphabet, no matter whether the automaton is deterministic or not
- by default, all transitions in a newly created state point to the state itself (aka. a dead state)
- transitions shall never be set at undefined input characters nor to be pointed at a non-exist state
- a state can be deleted if and only if there’s no transition from other states pointing to it.

Those conventions may be strict toward some coarsely defined automata. However, there is always solutions to certain problems. For example, in a coarsely defined DFA, there may be some partially defined states, in which we cannot find certain transitions corresponding to some input characters (which breaks the first convention). However, this can be fixed by adding one ‘dead state’ and then making up all those missing transitions pointing to the dead state. Then, we will have a new automaton which satisfies the conventions and is equal to the original one.

### 4.3 Running simulation

Next important task is to run big data through the constructed model, deciding that whether the input is acceptable. Here comes a new problem that how can different types of input data be reconciled onto one model.

To solve that problem, we have come up with an assumption that alphabet is already sorted and each character can be mapped to a continuous integer domain which starts from zero. For example, an alphabet composed by Arabic numerals and Latin letters in both lower and upper cases. Firstly, all characters are sorted in an order that numerals comes prior to lower case letters, which preempts upper case letters. Say, \( \Sigma = \{0, 1, ..., 9, a, ..., z, A, ..., Z\} \). Next, we will define a mapping from each letter to an integer starting from zero: \( \Sigma \rightarrow \Sigma' = \{0 \rightarrow 0, 1 \rightarrow 1, ..., 9 \rightarrow 9, a \rightarrow 10, ..., z \rightarrow 35, A \rightarrow 36, ..., Z \rightarrow 61\} \). In this way we have mapped the entire alphabet to a continuous integer domain \( \Sigma' = \{0, 1, ..., 61\} = [0, 62) \). And now the automaton we have built already just has to deal with input strings that are composed of integers.

By the assumption above, \( CAT \) will merely regard certain alphabet as one integer. And that’s the only parameter used to stand for the entire alphabet. Moreover, when it comes to symbolic automata, this upper bound can even be ignored by assuming that all alphabets adopted by symbolic automata are infinite sets. Since the characters that don’t really exist can be mapped to an extra symbolic character leading the automaton to a dead state. For example, an automaton \( A \) is defined on an integer domain \([0, 100]\), its symbolic version \( A_S \) can be defined on \([0, +\infty)\), where \(\forall a \in [100, +\infty)\) are all mapped to one dedicated symbolic character \(\alpha\), leading to a dead state in \( A_S \).

This assumption can be applied not only to alphabet \( \Sigma \) but also stack alphabet \( \Gamma \) in PDAs. It plays an important role in the design and implementation of \( CAT \), which discovers a great balance between generality and simplicity. Since there’s no extra restriction for the mapping function \( \psi \) and \( \rho \), they can be not only staircase functions,
but also some (higher order) logistic formulas, such as \(x < 10\), \((x^2 + y^2 - 2xy \geq 0) \lor (z \neq 100)\) etc. However current version of \(\text{CAT}\) only adopts the staircase function.

### 4.4 Calculus

\(\text{CAT}\) not only provides the solution for automata construction and running simulation, but also implements some basic calculus among supported automata.

Automata are by all means abstract computing machines that run upon strings and decide whether to accept them or not. Thus, they represent languages, namely, sets of strings. Hence the calculus among automata is actually computation among sets and strings. As has been proved in automata theory (Hopcroft et al., 2006), regular languages are closed under boolean operations and string operations such as concatenation, reverse, and closure etc. Therefore (non)deterministic finite automata possess fine properties and typical algorithms in those calculations. Subclasses of RegularAutomata in \(\text{CAT}\) have implemented those methods respectively. That provides a new approach to construct new automata as well. Unfortunately, in the field of PDAs, not all those properties holds. Thus the basic calculus implemented for them are limited.

Except the calculus between automata of same type, there’s one special computation among different types, transformations. According to the theory of automata, DFA and NFA both represent same type of language – regular language. Thus they can be equally transferred to each other. However, for DPDA, it has a weaker expressiveness in comparison with NPDA. Thus the transformation from DPDA to NPDA is single-way. But they are both more powerful than regular language. As for symbolic automata, they merely add a mapping procedure when reading input characters and stack symbols, which will not expand the language set an automaton can express. Hence, their transformation relation is just the same as their corresponding concrete forms, and thus the transformation between them is bi-directional. The overall view of the transformation relation can be summarised as Figure 5. In current version of \(\text{CAT}\), we implemented part of the transformations.

**Figure 5** Transformation relationships

Besides, there are also two other methods of calculus: minimisation and sampling. For minimisation, there are already some algorithms that can efficiently solve the problem (unfortunately, for regular languages only). For sampling, we now coarsely implemented two types of this operation: random and minimum. In minimum sampling, the automata will return an accepted string minimum in lexicographical order. In random sampling, the automata will randomly return a string in the language it defines. Sampling provides a mechanism that will generate accepted strings even though we don’t actually know what the automaton expresses. In ideal circumstances, random sampling should select each string at an identical probability. In the current version of \(\text{CAT}\), we can somehow randomly generate strings accepted by certain automaton. However, we cannot guarantee absolutely even probability.

### 5 Test

Before a new tool can be put in practice use, it should be proved useful. Therefore, we have to answer two questions:

1. Does this tool work as it was designed?
2. How efficient can this tool be?

In this test, we plan to answer the first question. We have programmed several simple demo to test automaton calculus and applied \(\text{CAT}\) to the language learning algorithm \(L^*\) to test more operations and their combination.

#### 5.1 Test environment

This test was complied under C++ 03 standard via GCC 4.8.2 (Ubuntu 4.8.2-19ubuntu1), and was performed on Linux Ubuntu 14.04.

#### 5.2 Coding example

An important principle we adopted in the development is KISS: *Keep It Simple and Stupid*. Thus, the usage of \(\text{CAT}\) is relatively simple and intuitive. That will be illustrated in the simple program shown in the next page.

#### 5.3 Basic tests

According to the example codes above, we can do our basic tests using that DFA A and B. Figure 6(1) to 6(7) show the result of the calculation done in the example. (1) and (2) give the primary automata A and B, (4) (5) (6) (7) give the minimised result for intersection, union, subtraction and complementation. However, each operation may give not necessarily the minimal result, it may require further minimisation operation. As is shown in (3), the raw result is not a minimal automata.

#### 5.4 \(L^*\) algorithm

\(L^*\) algorithm was first proposed by Angluin (1987). It has been widely used in grammatical inference (De la Higuera, 2010) and model checking researches (Cobleigh et al., 2003), and been adapted to be used in other circumstances (Angluin and Fisman, 2014). There are two modules in
the algorithm: teacher and learner. Teacher by some means maintains a certain data semantics (e.g. regular language). The goal of the algorithm is to let the learner learn that semantics via queries, and then display the language by some type of automata.

To make a further test of CAT, we implemented L* algorithm with CAT and counted total queries used by learner. The teacher, we supposed, somehow maintains a regular language that is defined on set \{0, 1\}, and each string in the language starts with 1 and equals 5 if it is treated as a binary integer (non-negative). For example, 101, 1010, 1111 are all accepted by this language. After the learning procedure is finished, the learner displays a DFA shown in Figure 7.

It took learner 186 membership queries and 5 equivalence queries in total.

All tests in this section show that CAT is able to run correctly and reached its design purposes.

```cpp
#include "CAT.hpp"
define ALPHABET 2 // Set alphabet as {0, 1}

int main()
{
    /* Create four empty DFAs*/
    DFA A(ALPHABET);
    DFA B(ALPHABET);
    DFA C(ALPHABET);
    DFA D(ALPHABET);
    /* Construct DFA A by adding states*/
    Label_T q0 = A.AddState(false); // Add a non-accepting state
    Label_T q1 = A.AddState(false);
    Label_T q2 = A.AddState(true); // Add an accepting state
    Label_T q3 = A.AddState(false);
    A.Transition(q0, 0, q1); // Set transition: q0-[0]->q1
    A.Transition(q0, 1, q3);
    A.Transition(q1, 0, q2);
    A.Transition(q1, 1, q2);
    A.Transition(q2, 0, q3);
    A.Transition(q2, 1, q3);
    // There's no need to set transitions on q3. Because all transitions on q3 will point
    // to itself by default.
    A.StartState(q0); // Set the start state, or it cannot run.
    A.Restart(); // Initiate automaton
    /* Construct DFA B in the same way*/
    Label_T s0 = B.AddState(false);
    Label_T s1 = B.AddState(false);
    Label_T s2 = B.AddState(true);
    Label_T s3 = B.AddState(false);
    B.Transition(s0, 0, s1);
    B.Transition(s0, 1, s2);
    B.Transition(s1, 0, s3);
    B.Transition(s1, 1, s2);
    B.Transition(s2, 0, s3);
    B.Transition(s2, 1, s3);
    B.StartState(s0);
    B.Restart();
    /*Do some computation*/
    C = A & B; // Intersection of A and B
    C = A | B; // Union of A and B
    C = A - B; // Subtraction of A and B
    C = ~A; // Complement of A
    A == B; // Judge if A and B are equal
    A != B; // Judge if A and B are not equal
    C = A & B;
    D = C; // Copy assignment;
    D.Minimize(); // Minimization
    return 0;
}
A customised automata algorithm and toolkit for language learning and application

6 Yet another case study

It is also vital for a toolkit to be efficient to process huge amount of data. To answer the second question in Section 5, we adopted a performance test in this section. In this test case, an DFA has been constructed to scan certain input to search for a given pattern. We fed it input string of different size and see whether it can run correctly and how fast it could be. As a control group, Linux ‘GREP’ command had been chosen to perform the same task on exactly the same input data.

6.1 Test environment

This case study is complied under C++ 03 standard via GCC 4.9.2 (Ubuntu 4.9.2-10ubuntu13), and is performed on Linux Ubuntu 15.04 operating system.

6.2 Text search

To construct the experimental input data, an alphabet is chosen on lower case Latin letters: \( \{a, b, c, \ldots, z\} \) which can be mapped to an ordered set of integers: \( \{0, 1, 2, \ldots, 25\} \). And then, we generated randomly various sized input strings, of respectively length 1M, 10M, 20M, \ldots, 100M.

After that, every data was fed both into the DFA and ‘GREP’. We tested performance of searching for target string whose length are 3, 5 and 10. All performance results are listed in Tables 1 and 2.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Performance of ‘GREP’ (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>10M 20M 30M 40M 50M 60M 70M 80M 90M 100M</td>
</tr>
<tr>
<td>3</td>
<td>35   404 531 745 903 1068 1301 1460 1640 1792</td>
</tr>
<tr>
<td>5</td>
<td>36   141 85 758 907 1071 1330 1487 1635 1806</td>
</tr>
<tr>
<td>10</td>
<td>23   78 61 116 107 145 186 183 220 208</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Performance of CAT (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>10M 20M 30M 40M 50M 60M 70M 80M 90M 100M</td>
</tr>
<tr>
<td>3</td>
<td>45   383 757 1132 1523 1882 2270 2650 3014 3472 3822</td>
</tr>
<tr>
<td>5</td>
<td>53   382 754 1125 1503 1918 2237 2638 3030 3398 3772</td>
</tr>
<tr>
<td>10</td>
<td>42   396 823 1170 1538 1927 2308 2720 3113 3466 3872</td>
</tr>
</tbody>
</table>

From Figure 8(1) to Figure 8(4) we can observe clearly that the simple DFA we constructed with CAT has a highly linear execution time, and no matter how many characters there are in target string, the performance is relatively stable. In contrast, ‘GREP’, as a mature tool for pattern matching and regular expression searching with many optimisation attempts, had a better performance and ran under different speed according to different input data.

In most cases, unoptimised constructed-by-CAT DFA gains an overhead of approximately 100%, and was fixed ever since. If we drop off all target-specific optimisation
applied in ‘GREP’, the gap will be much smaller as it gets closer to net overhead caused by framework difference. All statistics in this case study show a tolerable performance overhead in current architecture of CAT.

7 Conclusions

Although there are already many researchers and engineers that have designed and implemented several tools that can deal with automata, these tools are not really suitable to manually construct automata. And in order to use these tools well, one has to read the documents really carefully and practice a lot. Simplicity and freedom are the mainly advantages CAT has against them.

Despite the advantages, CAT has now still many aspects to be improved. Firstly, the calculus it provides is just basic and simple, not yet comprehensive. As is mentioned in Section 4, some methods (e.g. Random sampling) does not meet our expectation. Secondly, this simple architecture cannot guarantee the performance that one instance of automata can have. It jumps a lot (maybe in a far distance) in memory. Efficiency should be improved if it is going to be put into big data processing pipelines. Last but not least, current CAT merely allows users to build various structures on all predefined automata types instead of truly customising a new kind of automata. Namely, one cannot build a register automata or timed automata with CAT since they are not defined in the architecture of the tool kit. However, in data analytics, it is already enough to construct transmission system to represent data semantics for industrial purposes.

Acknowledgements

This project is supported by the National Natural Science Foundation of China (No. 61472240, 91318301, 61572268).

References


