REGAL: a library to randomly and exhaustively generate automata

Frédérique Bassino, Julien David, and Cyril Nicaud

Institut Gaspard Monge, UMR CNRS 8049
Université de Marne-la-Vallée, 77454 Marne-la-Vallée Cedex 2, France
{bassino,jdavid01,nicaud}@univ-mlv.fr

Description of the library REGAL

The C++ library REGAL [1] is devoted to the random and exhaustive generation of finite deterministic automata. The random generation of automata can be used for example to test properties of automata, to experimentally study average complexities of algorithms dealing with automata or to compare different implementations of the same algorithm. The exhaustive generation allows one to check conjectures on small automata.

The algorithms implanted are due to Bassino and Nicaud, the reader can refer to [1] for the description and the proofs of the algorithms used. The uniform generation, based on Boltzmann samplers, of deterministic and accessible automata runs in average time $\mathcal{O}(n^{3/2})$ where n is the number of states of the generated automata.

REGAL works with generics automata. To interface it with another software platform, the user has to define some basic methods (adding a state or a transition for example). REGAL also defines an implementation of automata that can be used directly.

To generate automata, either randomly or exhaustively, a generator object has to be instancied with the following parameters: the type of the states, the type of the alphabet, the class of the output automaton, the number of states of the output automaton and the alphabet.

The exhaustive generator provides methods to compute the first automaton, to go to the next automaton, and to test whether the last automaton is reached.

To randomly and equally likely generate automata of a given size, one has to initialize a random generator and then use the method `random()` as shown in the example below.

```
DFAAutomaton<int,char> * result; //Result DFA
Alphabet<char> alpha; //Create an alphabet
alpha.insert('a');   alpha.insert('b');
RandomDFAGenerator<int,char,DFAAutomaton<int,char>> rg(50 ,alpha);
for(int counter=0; counter<10000; counter++) a=rg->random();
```

Fig. 1. Random generation of 10000 DFA with 50 states on $A = \{a, b\}$

[1] available at: http://igm.univ-mlv.fr/~jdavid01/regal.php

Experimental results

Using the exhaustive generator, we computed the exact number of minimal automata on a two-letters alphabet, for small values of n.

Number of states	2	3	4	5	6	7
Minimal automata	24	1 028	56 014	3 705 306	286 717 796	25 493 886 852

Using the random generator, the proportion of minimal automata amongst deterministic and accessible ones can be estimated. The tests in the following array are made with 20 000 automata of each size.

Size	50	100	500	1 000	2 000	3 000	5 000
Minimal automata	84.77 %	85.06 %	85.32 %	85.09 %	85.42 %	85.64 %	85.32 %

On a two-letters alphabet we tested how the size of a random automaton is reduced by minimization. The following array summarizes the results obtained on 4 000 automata of size 10 000:

Reduction of the size	0	1	2	3
Proportion of automata	85.26 %	13.83 %	0.89 %	0.02 %

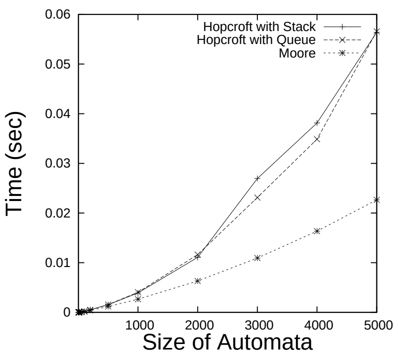

Fig. 2. Time complexities of Moore's and Hopcroft's algorithms

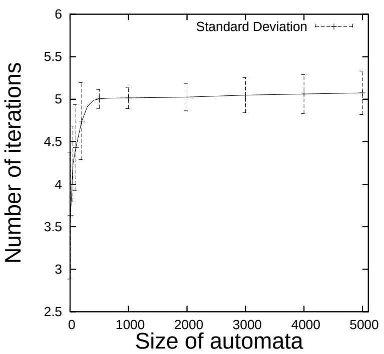

Fig. 3. Number of iterations in the main loop of Moore's algorithm

In Fig.2 the mean time of execution of Moore's and Hopcroft's algorithms has been measured on an Intel 2.8 Ghz. We used 10 000 automata of each size to compute the mean value. Two different implementations have been tested for Hopcroft's algorithm: either with a stack or with a queue.

In Fig.3 the mean number of partitions refinements in Moore's algorithm is analysed. Its very slow growth could explain why this algorithm seems efficient in the average (its worst case complexity of $\mathcal{O}(n^2)$ is reached for n refinements).

Reference

1. F. Bassino, C. Nicaud, Enumeration and random generation of accessible automata, *Theoret. Comput. Sci.*, to appear.
Available at http://www-igm.univ-mlv.fr/~bassino/publi.html