`automata`, a Hybrid System for Computational Automata Theory

K. Sutner *

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
sutner@cs.cmu.edu

Abstract. We present a system that performs computations on finite state machines, syntactic semigroups, and one-dimensional cellular automata.

1 Motivation

The `automata` system facilitates computation on finite state machines, syntactic semigroups, and one-dimensional cellular automata. Unlike some other systems such as Grail, AUTOMATE, Amore, see [6] for detailed references, the `automata` package is a hybrid system that is built around a commercial computer algebra system. Specifically, the current implementation uses version 4.1 of *Mathematica* by Wolfram Research, Inc., see [14]. Before commenting more on this approach, we present two typical sample sessions. Fairly detailed descriptions of earlier versions of the package can be found in [8, 9].

1.1 Entropy of Sofic Shifts

Suppose you wish to determine the entropy of the sofic subshift associated with a particular one-dimensional cellular automaton, see [5, 1] for more background information). Here is a short session in `automata` that shows the necessary calculations. The dialogue is captured the way it would appear in the plain text interface. For more elaborate examples using the notebook frontend, see `http://www.cs.cmu.edu/~sutner`. The first command converts the elementary cellular automaton number 92 into a de Bruijn semiautomaton, which is then converted into the corresponding minimal Fischer automaton, see [2, 10]. We extract the transition matrix

* Supported in part by NSF-ITR 0113919.

from the latter, construed as a non-negative integer matrix, and determine its Perron eigenvalue.

```
sa = ToSA[ CA[ 92, 3, 2 ] ];
mf = MinimalFischerFA[ sa ]
```

```
SA[ 8, 2, {{1, 1, 1}, {2, 1, 4}, {3, 1, 1}, {4, 1, 6},
 {5, 1, 1}, {6, 1, 6}, {7, 1, 8}, {1, 2, 2}, {2, 2, 3},
 {3, 2, 5}, {4, 2, 2}, {6, 2, 7}, {7, 2, 5}, {8, 2, 2}} ]
```

```
M = FullTransitionMatrixFA[ mf ];
Log[ 2, Max[ Abs[ N[ Eigenvalues[M] ] ] ] ]
```

```
0.900537
```

In the mode chosen here the semiautomaton `mf` is shown in abbreviated form. There are two invisible fields indicating the actual state set (a set of size 8 according to the first field in the automaton, see section 2 below), and the elements of the alphabet (a set of size 2 according to the second field in the automaton). The first three commands use operations defined in the package, whereas computation of the eigenvalues is handled entirely by *Mathematica*.

1.2 Preserving Regularity

As a second example, consider regularity preserving operations on regular languages. There is a family of such operations based on existential quantification over strings of a certain length. In particular, a function $f : \mathbb{N} \to \mathbb{N}$ is regularity preserving if for any regular language L the language

$$T(L, f) = \{\, x \in \Sigma^* \;\big|\; \exists\, y \in \Sigma^*(|y| = f(|x|) \wedge xy \in L) \,\}$$

is again regular. For specific functions f, the construction of the corresponding machines can be expressed easily in terms of Boolean matrices, see also [4]. For example, consider the function $f(i) = 2^i$. Here is the construction of a DFA for $T(L, f)$ where L is the language of all strings over $\{a, b\}$ whose length is divisible by 3. We construct a DFA for L by hand and define symbol `B` to be the natural homomorphism $\beta : \Sigma^* \to \mathbb{B}^{Q \times Q}$ from words to Boolean matrices of size $Q \times Q$. The ability to use higher

type objects in this effortless way is a significant advantage in actual interactive, experimental computations. Of course, in this particular case this is a bit of overkill, the Boolean sum $\beta(a) + \beta(b)$ is a simple circulant matrix here.

```
m = DFA[ 3, 2, {{2,3,1},{2,3,1}}, 1, {1} ];
TransitionMatrixFA[ m, B, Type->Boolean ];
M = BooleanUnion[ B[a], B[b]]
```

```
{{0, 1, 0}, {0, 0, 1}, {1, 0, 0}}
```

We can now define an operation `dot` that turns $Q' = Q \times \mathbb{B}^{Q \times Q}$ into a semimodule over Σ^*. First, the transition function of the DFA is assigned to `delta`. Operation `dot` then applies `delta` to the first component of any pair $(p, A) \in Q'$, and squares the Boolean matrix. The sub-semimodule generated by (q_0, M) provides the state set for the DFA `mm` recognizing $T(L, f)$. During the generation of the sub-semimodule we also produce the transition function for `mm`. Lastly, the final states (p, A) can be determined by the condition that $I_p \cdot A \cdot I_F$ not be the null vector.

```
TransitionFunctionFA[ m, delta ];
F = ToBitVector[ Final[m], Range[3] ];
dot[{p_,P_},s_] := { delta[p,s], BooleanComposition[P,P] };
final[{p_,P_}] := ToBitVector[ p, Range[3] ] . P . F > 0;
{Q,W,mm} = GenerateDFA[ {{1},M}, dot, 2, final ];
mm
```

```
DFA[ 6, 2, {{2, 3, 4, 5, 6, 1}, {2, 3, 4, 5, 6, 1}},
  1, {2, 3} ]
```

```
W
```

```
{Eps, a, aa, aaa, aaaa, aaaaa}
```

The other fields `Q` and `W` contain the carrier set of the semimodule and a collection of corresponding witnesses, respectively. Either one could be used as the underlying state set of `mm` if need be. At any rate, the resulting machine has 6 states and a little arithmetic shows that $T(L, f)$ should consist of all words of length i where $i \equiv 1, 2 \pmod 6$. We can verify

this computationally by generating a few words in the language, or by evaluating its census function up to length 20.

```
LanguageFA[ mm, -6 ]
LanguageFA[ mm, -20, SizeOnly->True ]
```

```
{{}, {a, b}, {aa, ab, ba, bb}, {}, {}, {}, {}}

{0, 2, 4, 0, 0, 0, 0, 128, 256, 0, 0, 0, 0, 8192, 16384,
 0, 0, 0, 0, 524288, 1048576}
```

As a last example, we calculate the syntactic semigroup of a regular language. Consider $L = \{\, x \in \{a,b\}^* \;\big|\; x_{-3} = a \,\}$, the set of all words having an a in the third position from the end.

```
m = MinimizeFA[ IthSymbolFA[ a, -3 ] ];
{S,W,eq} = SyntacticSG[ m, Equations->True ];
S
```

```
SG[T[2, 3, 5, 7, 5, 7, 3, 2], T[1, 4, 6, 8, 6, 8, 4, 1], T[3,
5, 5, 3, 5, 3, 5, 3], T[4, 6, 6, 4, 6, 4, 6, 4], T[2, 7, 7, 2,
7, 2, 7, 2], T[1, 8, 8, 1, 8, 1, 8, 1], T[5, 5, 5, 5, 5, 5, 5,
5], T[6, 6, 6, 6, 6, 6, 6, 6], T[7, 7, 7, 7, 7, 7, 7, 7], T[8,
8, 8, 8, 8, 8, 8, 8], T[3, 3, 3, 3, 3, 3, 3, 3], T[4, 4, 4, 4,
4, 4, 4, 4], T[2, 2, 2, 2, 2, 2, 2, 2], T[1, 1, 1, 1, 1, 1, 1,
1]]
```

The semigroup is presented as an explicit set of functions $[8] \to [8]$. We also determine the canonical rewrite system for the semigroup, which turns out to consists of all directed equations $\sigma_1\sigma_2\sigma_3\sigma_4 = \sigma_2\sigma_3\sigma_4$. There are 8 idemptotents in the semigroup, and they happen to coincide with the right nulls.

```
id = IdemSG[S]
```

```
{T[5, 5, 5, 5, 5, 5, 5, 5], T[6, 6, 6, 6, 6, 6, 6, 6], T[7, 7,
7, 7, 7, 7, 7, 7], T[8, 8, 8, 8, 8, 8, 8, 8], T[3, 3, 3, 3, 3,
3, 3, 3], T[4, 4, 4, 4, 4, 4, 4, 4], T[2, 2, 2, 2, 2, 2, 2, 2],
T[1, 1, 1, 1, 1, 1, 1, 1]}
```

```
id == RightNullSG[S]
```

```
True
```

---

2 Experimentation, Prototyping and Production Code

One of the goals of `automata` is to demonstrate the feasibility of a computational environment that supports interactive computation, rapid prototyping of complicated algorithms, and the use of production scientific code. As a case in point, take the function `MinimalFischerFA` that was used in the entropy computation. Originally this function consisted of a short segment of *Mathematica* code, interactively developed and based on primitives provided by the package, whose sole purpose it was to compute a few Fischer automata arising from a some examples. The code was later collected into an experimental function available in the package, but not yet officially supported. In the last step, the function became a fully supported part of the package, complete with an implementation as external C++ code. The external code communicates with the kernel via *MathLink*, see [14], and allows one to deal with machines with many thousands of states. It can also be used as a part of a C++ library, in the form of shell scripts, or as a command in an interactive calculator written entirely in C++.

Similar comments apply to other operations that tend to be computational bottlenecks: computation of a power automaton, minimization, generation of syntactic semigroups, to name a few. All these operations are implemented both in *Mathematica* and externally in C++. Note that this double implementation has some advantages with respect to checking correctness: the implementation languages *Mathematica* and C++ are sufficiently different to make it unlikely that the same error would appear in both implementations. As indicated in the semimodule computation above, some operations can optionally produce certificates that can be used to verify the correctness of the output.

Considerable effort has gone into integrating the two components as tightly as possible. For example, the internal algorithms dealing with finite state machines support an option `Normalize` which allows the user to preserve the natural state set of a machine. Thus, in a power automaton construction the state set of the new machine is naturally a subset of $\mathrm{pow}(Q)$, where Q is the state set of the nondeterministic machine. In a product automaton, the state set is a subset of $Q_1 \times Q_2$ and minimization produces a partition of the state set of the given machine. By default the state set is always normalized to $[n]$, but whenever necessary we can preserve the structure even during nested operations:

```
m1 = ToDFA[ InfixFA[ aba ], Normalize->1 ];
MinimizeFA[ m1, Normalize->2 ] // States
```

```
{{{1}}, {{1, 2}}, {{1, 3}}, {{1, 2, 4}, {1, 3, 4}, {1, 4}}}
```

The same options are also available in the external code; indeed, nested lists of atoms (integers, strings, finite state machines, semigroups, cellular automata) are the basic data structure in the external code.

Another important point is the quality of the frontend. The visual presentation of mathematical information is a challenging task, and one should not expect satisfactory solutions from ad-hoc efforts. The *Mathematica* frontend on the other hand produces near-publication quality results, and provides easy access to a large collection of operations. When the data produced by the core algorithms of the system are complicated in nature (e.g., the D-class decomposition of a semigroup), the notebook frontend greatly helps to display, manipulate and further analyze the data, conveniently within the whole system.

For research applications yet another aspect of considerable importance is the archiving and indexing of data. Often the actual calculation of the data requires a relatively small program based on the machinery in the system. Ideally, the program, accompanying text, the actual data, and their analysis should all be bundled in a single unit. Dealing with collections of separate files becomes quickly unwieldy, and often forces tedious and time-consuming recomputation. A coherent, platform independent interface such as a *Mathematica* notebook addresses all these issues.

Needless to say, maintenance of a hybrid system poses significant challenges. The latest version of `automata` uses XML as the sole repository for the *Mathematica* code. The various components are automatically assembled into a so-called add-on package via XSL style-sheets. This bundling process produces a software package that can simply be deposited in the user's home directory, and that will then load automatically. Short help on a per-function basis is available from the notebook interface, and there is a large collection of notebooks that demonstrate the use of the package. We are currently working on full integration with the extensible *Mathematica* help-browser. In the external code, the STL is used as the main source for standard data structures, see [7, 3]. If desired, the user can provide memory managers to speed up the external code (the standard memory manager is taken from the STL). Apart from the nested lists of atoms the external code tries to avoid inheritance in favor of parametrized types; thus it is relatively easy to read, extend, and modify.

The package has been brought to bear on a number of problems that might otherwise well have proven intractable, see [11, 13, 10, 12]. The code is available at `http://www.cs.cmu.edu/~sutner`.

References

1. M.-P. Beal and D. Perrin. Symbolic dynamics and finite automata. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2, chapter 10. Springer Verlag, 1997.
2. R. Fischer. Sofic systems and graphs. *Monatshefte für Mathematik*, 80:179–186, 1975.
3. G. Glass and B. Schuchert. *The STL <Primer>*. Prentice Hall, 1996.
4. D. Kozen. Lower bounds for natural proof systems. In *Proc. 18-th Ann. Symp. on Foundations of Computer Science*, pages 254–266. IEEE Computer Society, 1977.
5. D. Lind and B. Marcus. *Introduction to Symbolic Dynamics and Coding*. Cambridge University Press, 1995.
6. R. Raymond, D. Wood, and S. Yu. *First International Workshop on Implementing Automata*, volume 1260 of *Lecture Notes in CS*. Springer Verlag, 1997.
7. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.
8. K. Sutner. Finite state machines and syntactic semigroups. *The Mathematica Journal*, 2(1):78–87, 1992.
9. K. Sutner. Implementing finite state machines. In N. Dean and G. Shannon, editors, *Computational Support for Discrete Mathematics*, volume 15, pages 347–365. DIMACS, 1994.
10. K. Sutner. Linear cellular automata and Fischer automata. *Parallel Computing*, 23(11):1613–1634, 1997.
11. K. Sutner. σ-automata and Chebyshev polynomials. *Theoretical Computer Science*, 230:49–73, 2000.
12. K. Sutner. The size of power automata. In J. Sgall, Ales Pultr, and Petr Kolman, editors, *Mathematical Foundations of Computer Science*, volume 2136 of *SLNCS*, pages 666–677, 2001.

13. K. Sutner. Decomposition of additive CA. Submitted to Complex Systems, 2002.
14. S. Wolfram. *The Mathematica Book*. Wolfram Media, Cambridge UP, 4th edition, 1999.