Progress report: cellular automata rule identification

Witold Bołt <witold.bolt@hope.art.pl>

September 18, 2012

1 Problem definition

In this research we develop an automated method of identifying a cellular automata rule from spatio-temporal images (i.e. observations of system evolution in time). Such problem in general is of great importance – we hope to develop a method that could be used to analyze real-world observations and build models based on that.

To examine different methods of constructing such models we start with a simplified version of the problem. Firstly a randomly chosen, one dimensional, binary CA rule R is chosen. Only rules with symmetric neighborhoods with positive radius are considered. Then a set S of selected initial conditions (states of 1D finite lattice) is formed by randomly filling 0's and 1's. After that each space state $s \in S$ is evolved according to R. Exactly T evolutions (iterations) are calculated for each $s \in S$, so for each s we have a spatio-temporal image $i(s)$. This sets our evaluation criteria - we will try to discover the rule R by means of evolutionary computing, relying only on samples $i(s)$.

We shall use following notation:

1. S - set of starting conditions,
2. $s \in S$ - selected starting condition,
3. R - cellular automata 1D rule,
4. $rad(R)$ - radius of the neighborhood of rule R,
5. $i(s)$ - spatio-temporal image of starting condition s containing samples for time $t = 0, 1, \ldots, T$,

6. $|s|$ - count of cells in 1D space of starting state s,
7. $i_t(s)$ - state in time t of evolution $i(s)$,
8. $i_t(s)[j]$ - is value of j-th cell in time t in image $i(s)$,
9. we will use rule symbol R as a function symbol, so that we will write: $R(i_t(s)) = i_{t+1}(s)$. Following this notation applying rule R on state $i(s)$ multiple times (t-times) is denoted with: $R^{(t)}(i(s)) = i_t(s)$.

2 Genetic algorithm

We start by randomly selecting N rules r_i^0, $i = 1, \dots, N$, with radiuses $0 < \mathcal{R}_1 \leq rad(r_i^0) \leq \mathcal{R}_2$, where $\mathcal{R}_1 \leq rad(R) \leq \mathcal{R}_2$.

Then we evaluate all those rules according to fitness function. For some rule r the fitness function is given by:

$$fit(r) = \frac{1}{|S|} \sum_{s \in S} 1 - \frac{\sum_{t=1}^{T-1} 2^{-t} \left(\sum_{j=1}^{|s|} |r^{(t)}(i_0(s))[j] - i_t[j]| \right)}{|s|(1 - 2^{-T+1})}.$$

To make this formula more readable, lets denote the number of errors (cells in wrong state) of rule r with initial state s at time t by $e_t(r, s)$. Then:

$$e_t(r, s) = \sum_{j=1}^{|s|} |r^{(t)}(i_0(s))[j] - i_t[j]|.$$

Note that for any rule r, $t \in [1, T]$ and $s \in S$ we have:

$$0 \leq e_t(r, s) \leq |s|.$$

We can now rewrite $fit(r)$ using this symbol:

$$fit(r) = \frac{1}{|S|} \sum_{s \in S} 1 - \frac{\sum_{t=1}^{T-1} 2^{-t} e_t(r, s)}{|s|(1 - 2^{-T+1})}.$$

Now we can see that if a particular rule r is bad enough to have $e_t(r, s) = |s|$ for every $s \in S$, than:

$$fit(r) = \frac{1}{|S|} \sum_{s \in S} 1 - \frac{|s| \sum_{t=1}^{T-1} 2^{-t}}{|s|(1 - 2^{-T+1})} = \frac{1}{|S|} \sum_{s \in S} 1 - \frac{\sum_{t=1}^{T-1} 2^{-t}}{(1 - 2^{-T+1})} = 0.$$

On the other hand, if $e_t(r, s) = 0$ for all $s \in S$, then obviously $fit(r) = 1$.

Generally for any rule r we have $0 \leq fit(r) \leq 1$, and $fit(r) = 1$ if $r = R$ or if r is good enough to fully simulate R on given image set.

So the goal of genetic algorithm is to maximize fit function, by selecting best matches, reproducing and manipulating them.

2.1 Selection

We select rules randomly, with weighted chances based on the fitness (i.e. rules with higher fitness have bigger chances of being picked). For the population of N individuals we select $N/2$ pairs and feed them to crossover procedure, and then mutate each of the effecting organisms.

In the implementation we store all fitness values in a table `fitness`. Using this table, we then calculate helper table `progressiveFitness` which is defined as:

```
prograssiveFitness[0] = fitness[0]

for (int i = 1; i < populationSize; i++)
progressiveFitness[i] = progressiveFitness[i-1] + fitness[i];
```

Then we take a nonnegative random (with uniform distribution) number ρ not greater than the last entry in progressiveFitness table and pick a rule numbered i such that:

$$progressiveFitness[i-1] < \rho \leq progressiveFitness[i].$$

This way, rules with higher fitness are more likely to be chosen.

It's up to future research wether different ways of selecting best individuals give better results.

2.2 Cross-over

Crossover between rules r_1 and r_2 can only happen when $rad(r_1) = rad(r_2)$. It it's the case, than we use one-point crossover. We randomly select a point in rules LUT and cross-over entries in LUT starting from this point.

In case where $rad(r_1) \neq rad(r_2)$ then we randomly pick one of the rules, and up-scale or down-scale the other one, to match radiuses.

Up-scaling is always possible and its "lossless", since class of rules with bigger radius is richer than those with lower one. Down-scaling on the other hand looses some information of the rule definition, but it's also quite straightforward.

2.3 Mutation

We have 3 types of mutations: LUT entry level mutation, random up-scaling and down-scaling.

First kind of mutation simply switches entries in LUT with some small probability p_m.

The additional type of mutations down-scale or up-scale the radius of a rule at random with some small probabilities p_{down}, p_{up}, to give more diversity to the population and to enable searches in different radiuses.

2.4 Sucess criteria

The evolution of genetic algorithm continues with iterative applying of fitness recalculation, selection, cross-over and mutation. In each iteration we additionally keep the best individual from previous iteration without any modifications.

We end the algorithm when there is a rule with $fit(r) = 1$ or when a fixed number of iterations elapsed.

3 Running the code

Current version is in the git repository (https://github.com/houp/ca-identify). You can download the code or clone the repository using git.

To run and compile the code you need JDK 1.7 (1.6 might work) and ant build tool. If you have all that installed simply run `ant` command in the root tree of repository, and then go to `dist` subdirectory and use `run.sh` script to run the program. Note that those scripts were only tested on UNIX-like systems (Linux, OSX) but should also work on Windows (maybe with some small changes).

Typical output of current version looks like this (one iteration):

```
t=3, max_f=0,863360, avg_f=0,585043, rule=Rule: 2295292771 @ radius: 2
Time = 82
dist[0] = 0, dist[1] = 60, dist[2] = 253, dist[3] = 187,
```

It means that this is evolution step 3, maximum fitness for now is 0.863360, average fitness is 0.585043, current best rule is: 2295292771 with radius: 2, time of calculating this evolution was 82 ms, and the distribution of radiuses among the population is as shown in the `dist` table.

The parameters of the simulation are hard-coded in the source tree for now, in `ga.base.Params` class:

```
public class Params {
public final static double mutationProbability = 0.07; // point mutation
public final static int populationCount = 500;
public final static int minRadius = 1;
public final static int maxRadius = 3;
public final static double upScaleProbability = 0.07; // up-scale mutation
public final static double downScaleProbability = 0.07; // down-scale mutation
public final static int maxRunCount = 5000; // iterations
public final static long rule = 2294967295L; // rule to look for
public final static int radius = 2; // radius of the rule to look for
}
```

You can adjust those, recompile (with ant) and re-run to see the results.

4 Results

TODO: charts, tables, samples..