Senior Thesis Abstract

Kevin Mc Inerney

March 21, 2008

I am studying one-dimensional cellular automata, in particular, I am looking at statements that can be made about arbitrary cellular automata. A one-dimensional cellular automata is an infinite series of cells, each of which is in a finite number of states at a current discrete time step t. The automata also has a transition relation which defines the state of the cells at time $t+1$. Formally, a cellular automata $A = (Q, \delta)$ is a tuple where Q is the set of potential states of the cells (the alphabet) and the transition relation $\delta : Q^k \rightarrow Q$ is a relation taking a vector of k cells into a new cell. The relation is applied locally to each cell and it's neighborhood (the $k-1$ cells adjacent to it) to obtain the new state of the cell at the next time step. The local nature of the transition relation is key to the defintion of a cellular automata. It is not necessary to know any information beyond a cell's neighborhood to perform the update on that cell. Moreover, the relation is the same for all cells of the automata, simplifying computation and making the entire structure position-independent and local.

We may simulate the computation of a cellular automata $A = (Q, \delta)$ by stating a current state and an update function. The state $S \in Q^{\omega\omega}$ (where $Q^{\omega\omega}$ is the set of all possible two-way infinite words on the alphabet Q) is a two-way infinite vector which contains the state of each of the component cells of the automata. Note that although the order of the cells is important, it is not necessary (or even possible) to state which is the "center" cell. To obtain the next step of the automata, we define an update function $F_\delta : Q^{\omega\omega} \rightarrow Q^{\omega\omega}$ that is obtained by applying the transition relation $\delta$to each cell of the current state and that cell's neighboorhood and combining the results to obtain the next state of the automata. By repeatedly applying F, we can simulate the computation of the cellular automata on a given input state S.

Because the transition relation is local and position-independent, we do not need to store any information about the state beyond the current cell and it's neighborhood. Therefore, it is possible to simulate the application of F on a state S of the automata by runnning a finite state transducer on S. (A Finite State Transducer (FST) is similar to a finite state machine except you also specify an output for each transition of the FST.) This allows us to use a simple, elegant model to study the update function of an arbitrary cellular automata. Forming the FST for the update function of a given cellular automata is not computationaly difficult. For a given transition relation $\delta : Q^k \rightarrow Q$, we create $|Q|^{k-1}$ states, one for each of the possible combinations of the last $k-1$ symbols

read. From each state, we create $|Q|$ transitions, labeling each with the letter of the alphabet read, and the symbol to be output. We can then run FSM reduction algorithms on the FST generated to get the minimal FST. When this FST is run on a given state S, it will produce a new state S' that is the result of running the orginal cellular automata for one step.

As an aside, note that there are extensions of the concept of a Finite State Machine that work on infinte words. One such automata is the Büchi automata. Büchi automata are nondeteriministic finite state machines with modified acceptance conditions. A Büchi automata is said to accept an infinte word if starting in the initial state, a run of the Büchi automata passes through a state in the set of final states infinitely often. It can be shown that the Büchi automata model accepts all regular infinite words. However, trying to determinize a Büchi automata directly does not work, as we find that the acceptance condition is not sufficent. We must use a new kind of automata, known as a Rabin automata. Again, Rabin automata are extensions of the Finite State Machine concept to infinite words, but with a different acceptance condition. A Rabin automata's acceptance conditions is a set of pairs of states (L, R) where the automata accepts a string if for one of the (L, R) pairs, the states in L are not hit infinitely often, and R is a set of states some of which must be hit infinitely often.

We define a "steps to" relation. The steps to relation is a logical extension of the update function defined above. Specfically, for a given automata and two states S, S' of that automata, we are asking if $F(S) = S'$, where F is the update function. We write this in logic as $S \to S'$. Note that we can check this by creating a new automata based on the FST generated by F. This automata is defined over a new alphabet, the set of all pairs of characters in the orginal alphabet of the cellular automata in question. We use the FST, except we make all of the transitions occur if the first character matches the transition, and the second matches what the FST would output. We also add a sink, which we transition to in all other cases. All states of the machine are final states except the sink. (Note that this machine accepts two states S, S' only if the second is right shifted so that a cell lines in S' lines up with the end of it's neighborhood in S.)

There is an algorithim, Safra's algorithim, that will deteriminize a Büchi automata and produce a deterministic Rabin automata. However, Safra's algorithm is quite complex, and the blowup in states is $2^{O(n \log n)}$ in the worst case. One aim of the project (as we will see below) is to find automata where Safra's algorithm has a less substatial blowup.

We are going to express properties of cellular automata(CA) as statements in prepositional language extended with the steps to relation. We may then ask questions such as: Does this CA have a three cycle? $(\forall x, y, z.(x \to y \wedge y \to z \wedge z \to x \wedge x \neq y \wedge y \neq z))$. Moreover, we may express each of the component parts of these statements as automata, and use standard techiniques (described below) to combine them into prepositional logic statements.

One major difficulty is that the infinite word automata generated are designed for one-way infinite words, and we must stretch them to create automata for two way infinite words. Specfically, we generate two automata one for the

forward infinite word, and one for the backwards infinite word. Because of the position-independent nature of cellular automata, these two automata will be identical. We then combine the two automata. Since states of cellular automata are shift invariant, we must be careful how we combine the automata. Specifically, we must allow variaition in the inital states of the two automata.

We have already seen how to express the steps to relation as an automata. To express the does not equal relation, we create a simple automata over a new alphabet, the set of all pairs of characters in the orginal alphabet of the cellular automata in question. We then create a two state machine, which remains in the inital state so long as both elements of the pair of characters are the same, and transitions to the final state at the first point where they differ. The machine then remains in the final state for the rest of the computation. This machine accepts the string if the two states represented differ in at least one location. We may also make a simple extension of the machine to check if the two strings are unequal where one of them is left or right shifted by a finite amount.

We can now express the conjunction or disjunction of two prepositional statements A, B by taking the resulting FSM's and performing the intersection of them to get $A \wedge B$, and the union of them to get $A \vee B$. Both of the combinations are computationaly efficent.

To obtain $\neg A$ from the FSM representing A, we must first determinize the FSM using Safra's algorithim, resulting in a Rabin automata. We can then change the acceptance condition of the Rabin automata to get the complement.

To obtain $\exists x(A(x))$, we create the FSM for A as usual, except wherever we would have a character from x, we instead place a wildcard, allowing any character to be used. This will likely cause the automata generated to become nondeterministic.

To obtain $\forall x(A(x))$, we simply use the logical equivalence $\forall x(A(x)) \Leftrightarrow \neg\exists x(A(x))$.

Once we have the desired automata, we can perform a simple graph algorithm to determine whether an accepting path exits. Specfically, for a Rabin automata, for a given acceptance pair (L, R) whether there exists a path $p_o p_1 \dots p_i \dots p_n p_i \dots p_n$ such that p_o starts at an inital state and ends in a state in R, and $\forall k, k < i$ p_k starts and ends in a state in R. For all p_k with $k \geq i$, we also introduce the condition that the path does not pass through any states in L. A relatively simple search technique will discover such a path if it exists. If any such path exists, we can say that the statement the automaton represents is true. If no such path exists, we can say that the statement the automaton represents is false.

Note that because of the potentially exponetial blowup in states caused by Safra's algorithm, this technique is not, in general, tractable. However, there are some Büchi automata where the blowup due to Safra's algorithm is manageable. Therefore, the goal of this thesis is to find a set of logical statements such that the resulting automaton has a mangeable number of states.