Grids of Finite Automata

Kārlis Jēriņš

University of Latvia, Riga, Latvia
karlis.jerins@gmail.com

Abstract. Finite automata are a well known topic in computer science. Automata read words in a given alphabet and determine if those words belong to a specific formal language. They have been the subject of lots of research, and much is known about their capabilities. Parallelism in automata - systems consisting of multiple automata that read the same word - is far less researched. In this paper, a new model of parallel calculation with automata is introduced. It is based on the concept of cellular automata, and uses multiple automata that are arranged in a certain grid and perform state transitions based on the states of their neighbors. Some early results are also shown.

Keywords: deterministic automata, parallel automata, regular languages

1 Introduction

Parallel computation is an important subject in the modern day – there are ever more and more problems that aren't solved by throwing more memory or CPU cycles at them, but instead require several programs to work in tandem, sharing the work load somehow. Research in this direction is also seen in automata theory – one needs to only consider models such as multihead automata [1] or multiprocessor automata [2]. We, however, are interested in a slightly different and somewhat constrained model – one where all units involved in the calculations are identical. This brings to mind the model of cellular automata – infinite grids of small, simple automata that change states based on the current states of other automata around them. Cellular automata have been well-researched in the past – some have even been found to be Turing-complete [3, 4]. However, they have some problems as detailed in the next chapter, and we introduce a new model – grids of finite automata – to compensate for those problems.

2 Grids of Finite Automata

Cellular automata, though certainly an interesting subject of study, make for poor tools for calculation. An infinite number of cells makes them close to impossible to implement, and it is hard to define when a cellular automaton can be said to have finished its calculations, or how to interpret its output. Grids of finite automata (from here on they will be referred to in an abbreviated form – GFA) are an answer to these problems.

A GFA, as implied by its name, consists of multiple finite automata arranged in a grid of some sort. In the examples shown in this paper, the automata will either be

arranged in one line or a rectangular grid, but other arrangements are easily available if necessary. The component automata are all identical save for their position in the grid (which is constant over time) and a number to identify them, and they consist of a set of states and a state transition function. However, unlike other models that use automata, these component automata do not have a read head and don't actually read symbols on an input tape. Instead, the input is transformed into a tuple of states and this tuple is used as the initial states for the components. The state transition function is also different – since the components have no read head, the function instead takes as input the current states of a number of other components of the GFA. For a given component, the components on whose states its state transitions depend are called neighbors, and the neighbor property is symmetric – if component a is one of the neighbors for b, then b is one of the neighbors for a.

One component of the GFA is designated as the main component. It has the same states and transition function as all the other components, and uses them the same way, but the main component provides one crucial function for the entire GFA. The GFA halts the instant its main component halts (other components are allowed to halt earlier if the state transitions indicate that they should). It also provides the answer for whatever calculation the GFA is performing – a function maps the state set of the main component to the set of all answers to that calculation, and thus the GFA gives its answer based on the last state its main component was in prior to halting.

Definition 1. *A GFA component is a tuple* $C = (Q, Q_0, n, F, A, B)$ *such that:*

- Q is a non-empty set of states
- $Q_0 \subseteq Q$ is a non-empty set of initial states
- n is the number of neighbors
- $F: Q \times (Q \cup \{\varepsilon\})^n \to Q \cup \{\varepsilon\}$ is the state transition function
- $A: Q \to N$ is the answer function that maps the halt state of the main component to an answer (of course, N can be substituted by other sets of answers as required).
- $B : \Sigma^* \to Q_0^* \times N$ is the initialization function that transforms input into a tuple of initial states of length equal to the length of the input and the identifying number of the main component. It is important to note that the function must be such that the identifying number is dependent only on the length of the input word, but not on its contents

If the state transition function is defined as written above, components are deterministic automata. GFAs with deterministic components could also be called GDA – to distinguish them from GFAs with other kinds of components. Those other kinds of components would necessitate different definitions of the state transition function, and are beyond the scope of this paper – thus, to avoid needlessly changing names, we will continue to use the acronym GFA.

Definition 2. *A GFA is a tuple* $G_k = (C, k, f)$ *such that:*

- C is a GFA component as defined above
- $k \in N$ is the number of such components (all identical)
- $f : [1,k] \to ([1,k] \cup \{\varepsilon\})^n$ is the neighbor function that maps each component to a n-tuple of its neighbors

In the state transition function definition above, ε refers to components that are either absent or have halted – the state transition function doesn't distinguish between these two cases. In the neighbor function, ε refers to absent components – for example, if, for the given GFA, components are arranged in one line and each component has two neighbors, one to the left and one to the right, then the components at the very ends of the line have only one neighbor each. The neighbor function indicates that by placing ε where the missing neighbor would otherwise be.

The GFA definition above (in particular, the definition of the initialization function) also highlights a limitation of this model – a GFA with k components can only process words of length k. The function could be modified to work with inputs of length up to $k * l$ for any $\in N$. As defined, the function usually will set the m-th component to a state $q_{s_m} \in Q_0$ such that s_m is the m-th symbol of the input, and such modifications would initialize components to states corresponding to a string of symbols of length l. That, however, would lead to rapid growth in state complexity for the components – usually the set of initial states is as big as the input alphabet (and, if necessary, a constant number of other initial states), but making each state correspond to a string of length l would increase the initial state set by a power of l. Since that would only decrease the number of components necessary for recognizing a word of given length linearly, we choose not to make this tradeoff.

The previous paragraph may give the impression that a GFA cannot really be said to solve a problem, since for every GFA there is a maximum input length it can process, and working with longer inputs necessitates creating a new GFA. This is not so – while it is true that the neighbor function needs to be adapted to different input lengths and different component counts, it is important to note that the structure of the components themselves is independent of input length – whether the input is one symbol or a billion, the components used in solving the task at hand do not change.

Let us now define more formally the way a GFA operates:

Definition 3. *A configuration of a GFA $G_k = (C,k,f)$ with components $C = (Q,Q_0,n,F,A,B)$ is a k-tuple $c \in (Q \cup \{\varepsilon\})^k$.*

Definition 4. *A GFA configuration is called an initial configuration if $c \in Q_0^k$*

Definition 5. *A GFA configuration is called a terminal configuration if $q_H = \varepsilon$, where H is the number of the main component*

Definition 6. *The GFA configuration transition relation $\vdash_{G_k}$ holds for configurations $c = (q_1, q_2, \ldots, q_k)$ and $c' = \left(q_1', q_2', \ldots, q_k'\right)$ if and only if all these conditions are true:*

- $q_H \neq \varepsilon$
- $\forall i \in N : (q_i = \varepsilon) \rightarrow \left(q_i' = \varepsilon\right)$
- $\forall i \in N : (q_i \neq \varepsilon) \rightarrow \left(q_i' = F\left(q_i, q_{f_1(i)}, q_{f_2(i)}, \ldots, q_{f_n(i)}\right)\right)$, where $f_j(i)$ denotes the j-th element of the tuple $f(i)$

If all these conditions are true, the configuration c' is said to follow the configuration c (this is denoted by $c \vdash_{G_k} c'$).

The first condition specifies that no configurations follow a terminal configuration. The second specifies that, once a component has halted, it cannot start again. The third describes how state transitions take place in a GFA.

Definition 7. *The output of G_k for an initial configuration c_0 is a natural number z such that there exist G_k configurations $c_1 c_2, \ldots, c_x$ ($c_i = (q_{1_i}, q_{2_i}, \ldots, q_{k_i})$) (for some natural number x), for which these conditions hold:*

- $c_0 \vdash_{G_k} c_1 \vdash_{G_k} c_2 \vdash_{G_k} \ldots \vdash_{G_k} c_x$
- c_x is a terminal configuration
- $A\left(q_{H_{x-1}}\right) = z$

If the output of G_k for initial configuration c_0 is z, this is denoted by $G_k(c_0) = z$.

Definition 8. *A GFA component $C = (Q, Q_0, n, F, A, B)$ calculates a function $\Phi : \Sigma^* \to N$ if and only if for every $x \in dom(\Phi)$ there is a GFA $G_{|x|} = (C, |x|, f)$ such that $G_{|x|}(B(x)) = \Phi(x)$.*

3 Language Recognition with GFAs

As with many other kinds of automata, GFAs are also intended for formal language recognition – the task is to determine if a given input word belongs to a language or not. That is essentially the same as calculating a function $\Phi : \Sigma^* \to \{0,1\}$, and it is already shown how GFAs calculate functions. This chapter will demonstrate some examples of GFAs that recognize languages.

Theorem 1. *For every regular language there is a GFA that recognizes that language.*

Proof. If a language $L \in \Sigma^*$ is regular, then there is a minimal one-way deterministic automaton $DFA = (Q_{DFA}, \Sigma, f_{DFA}, F_{DFA}, q_{0_{DFA}})$ (in order: set of states, input alphabet, state transition function, set of accepting states, initial state), where $Q_{DFA} = (q_{DFA_1}, q_{DFA_2}, \ldots, q_{DFA_X})$ and $\Sigma = (s_1, s_2, \ldots, s_y)$ for some natural x and y, that recognizes L.

Let us now create a GFA component for recognizing the language:

- $Q = \left\{q_{DFA_1}, q_{DFA_2}, \ldots, q_{DFA_x}, q_{s_1}, q_{s_2}, \ldots, q_{s_y}\right\}$ (the component has as many states as the DFA, plus as many as the size of the input alphabet)
- $Q_0 = \left\{q_{s_1}, q_{s_2}, \ldots, q_{s_y}\right\}$
- $n = 2$ (each component has two neighbors – essentially, all components are arranged into a single line)
- $A(q) = \begin{cases} 1, if\, q \in F_{DFA} \\ 0, if\, q \notin F_{DFA} \end{cases}$
- $F(q, q_L, q_R) = q'$, defined as follows:
  - $(q, q_L \in Q_0) \to \left(q' = q\right)$
  - $(q \in Q_0 \wedge q_L \in Q \backslash Q_0) \to \left(q' = f_{DFA}(q_L, s_i)\right)$, where s_i is such that $q = q_{s_i}$

- $(q \in Q_0 \wedge q_L = \varepsilon) \rightarrow \left(q' = f_A\left(q_{0_{DFA}}, s_i\right)\right)$, where s_i is such that $q = q_{s_i}$
- for all other cases, $q' = \varepsilon$

– $B(w) = \left(\left(q_{s_{w_1}}, q_{s_{w_2}}, \ldots, q_{s_{w_{|w|}}}\right), |w|\right)$ (w is the input word and $w = w_1 w_2 \ldots$). In essence, the i-th component is initialized to a state that corresponds to the i-th symbol of the input, and the very last component is set to be the main one

With the component defined, the task of creating the actual GFA for words of any given length is trivial. Words of length k are read by $G_k = (C, k, f)$, where $f(i) = \begin{cases} (\varepsilon, 2), if\ i = 1 \\ (i-1, \varepsilon), if\ i = k \\ (i-1, i+1)\ otherwise \end{cases}$.

To properly understand how the GFA created in the proof works, one needs to look at the way the state transition function is defined. The first rule says that a component that is in a starting state while its left neighbor is in a starting state will remain in the current state. The second rule says that, if a component is in a starting state matching some symbol s_i and its left neighbor is in a non-starting state q_{DFA_j}, the current component will move to the state that the DFA would go from state q_{DFA_j}, given symbol s_i. The third rule says that a component that has no working component to the left and is itself in a starting state q_{s_i} will proceed to whatever state the DFA would've proceeded to from the starting state, given symbol s_i. By following these three rules, the GFA completely simulates the operation of the DFA on the input word. After a number of steps equal to the number of components, the last component (the main one) will have reached a non-starting state, therefore one that matches a state from the DFA. And then the fourth rule guarantees that the main component, and therefore the entire GFA, halts and produces output.

With some more thought, it is possible to create GFAs that recognize non-regular languages as well. For example, the following GFA recognizes the context-free language $L = \{w\#w^{rev} \mid w \in \Sigma^* \wedge \# \notin \Sigma\}$.

Let us assume $\Sigma = \{s_1, s_2, \ldots, s_y\}$, $\# \notin \Sigma$. Then the GFA components that can recognize L are built like this:

– $Q = \left\{q_{s_1}, q_{s_2}, \ldots, q_{s_y}, q_\#, q_{reject}, q_{done}, q_{s_1,s_1}, \ldots, q_{s_1,s_y}, \ldots, q_{s_y,s_1}, \ldots, q_{s_y,s_y}\right\}$
– $Q_0 = \left\{q_{s_1}, q_{s_2}, \ldots, q_{s_y}, q_\#\right\}$
– $n = 2$ (again, all components arranged in a straight line)
– $F(q, q_L, q_R) = q'$ such that:
  - $(q, q_L, q_R \in Q_0) \rightarrow \left(q' = q\right)$
  - $(\exists i \in N : q = q_{s_i} \wedge (q_L = \varepsilon \vee q_R = \varepsilon)) \rightarrow \left(q' = q_{s_i,s_i}\right)$
  - $\left(\exists i, j, k \in N : q = q_{s_i} \wedge \left(q_L = q_{s_j,s_k} \vee q_R = q_{s_j,s_k}\right)\right) \rightarrow \left(q' = q_{s_i,s_k}\right)$
  - $\left(\exists i, j, k, l \in N : q = q_{s_i,s_j} \wedge \left(q_L = q_{s_k,s_l} \vee q_R = q_{s_k,s_l}\right)\right) \rightarrow \left(q' = q_{s_i,s_l}\right)$
  - $\left(\exists i, j \in N : q = q_{s_i,s_j} \wedge (q_L = \varepsilon \vee q_R = \varepsilon)\right) \rightarrow \left(q' = q_{done}\right)$
  - $\left(\exists i, j \in N : q = q_{s_i,s_j} \wedge (q_L = q_{done} \vee q_R = q_{done})\right) \rightarrow \left(q' = q_{s_i,s_i}\right)$
  - $\left(\exists a, b \in N : q = q_\# \wedge q_L = q_{s_a} \wedge q_R = q_{s_b}\right) \rightarrow \left(q' = q_\#\right)$

- $\left(\exists a,b,c \in N : q = q_{\#} \wedge q_L = q_{s_a,s_b} \wedge q_R = q_{s_c,s_b}\right) \rightarrow \left(q^{'} = q_{\#}\right)$
- $\left(\exists a,b,c,d \in N : q = q_{\#} \wedge q_L = q_{s_a,s_b} \wedge q_R = q_{s_c,s_d} \wedge b \neq d\right) \rightarrow \left(q^{'} = q_{reject}\right)$
- $(q = q_{\#} \wedge ((q_L = \varepsilon \wedge q_R \in Q_0) \vee (q_L \in Q_0 \wedge q_R = \varepsilon))) \rightarrow \left(q^{'} = q_{reject}\right)$
- For all other cases, $q^{'} = \varepsilon$

– $A(q) = \begin{cases} 1, if\, q = q_{\#} \\ 0, if\, q \neq q_{\#} \end{cases}$

– $B(w) = \left(\left(q_1, q_2, \ldots, q_{|w|}\right), H\right)$, where the i-th component is initialized to a state that corresponds to the i-th input symbol and $H = \begin{cases} \left\lceil \frac{|w|}{2} \right\rceil, if\ |w|\ is\, odd \\ 1, if\ |w|\ is\, even \end{cases}$

Creating the GFA itself is, again, fairly trivial once the component structure is complete. In fact, the neighbor function f is exactly the same as in the previous proof.

To clarify how this GFA works, one can start by looking at the initialization function. The intention is to make sure that, for odd word lengths, the main component is precisely in the middle of the word (even-length words aren't part of the language and can automatically be rejected). Next, looking at the answer function shows that the GFA only accepts the word if the main component terminates from state $q_{\#}$ - and a quick look at the state transitions shows that that state cannot be reached by a component that wasn't initialized to that state. All this combined means that the only words that stand a chance of being accepted are ones where the number of symbols before # and after it is the same.

In the cases where # is in fact in the middle of the word, the GFA operates by sending symbols from both ends to the middle. This sending is started by the second rule for function F – it says that when a component is on one end of the line, it enters a new state, and its neighbor detects the change and also enters a new state (the meaning for a state q_{s_i,s_j} is that the component was initialized with the symbol s_i but one of its neighbors is attempting to send symbol s_j to the middle). And a state that has sent its own symbol proceeds to q_{done} and then halts. In this way, the main component receives a pair of symbols from the opposite ends of the line and compares them. If they don't match, it goes to $q_{reject}$and then halts, rejecting the word. Otherwise, it continues until everything except the main component has halted. If it is still in $q_{\#}$ at that time, the GFA accepts the word.

This approach of sending symbols allows one to also recognize the context-free language $L = \{0^n 1^n \mid n > 0\}$. Again, the main component is placed in the middle of the word (either the last zero or the first one) for even-length words or in the beginning for odd-length words. This time the main component must be sent a zero from the left and a one from the right, otherwise it rejects the word.

Even better results can be accomplished if we don't limit the component arrangement to a single line. For example, allowing a rectangular placement (and making each component have four neighbors – top, down, left, right) lets us create a GFA that recognizes the context-sensitive language $L = \{0^n 1^n 2^n | n > 0\}$. The GFA that recognizes it will not be described completely here, but the main idea is to arrange the components in three rows and use the fact that, for words that belong to the language, the first row will be initialized to a state corresponding to 0, the second row – to 1, the third row – to

2. The main component should be placed on the left side of the whole arrangement, and the components should check if there is a 1 below every 0, a 0 above every 1, a 2 below every 1 and a 1 above every 2. Should one of these things not hold for any component, it should enter a state q_{reject} which would then propagate in all directions until it reached the main component, which would then halt and reject the word. If this doesn't happen, components should halt, one column at a time, from the right. In this situation, the main component should accept the word when it halts.

A similarly built GFA could also recognize the context-sensitive language $L = \{ww \mid w \in \Sigma^*\}$. That GFA would only need two rows of components, and the components would have to check if the i-th component of the first row is in the same state as the i-th component of the second row.

4 Future Work

Obviously, there is still a lot of work to be done in this area. So far we have only succeeded in placing a lower bound on the computational power of GFAs – they can recognize all regular languages and some non-regular ones. It is not yet clear whether all context-free languages, and perhaps even all context-sensitive languages, can be recognized by GFAs. It is quite likely that the arrangement of components (single line, rectangle, hexagonal grid, or other options) is also a factor that affects the computational power.

And, when all of the above is discovered, there is also the question of "what happens if the components aren't deterministic?' They could be simply non-deterministic, or probabilistic, or even ultrametric. We expect this to also be a significant factor in this model's computational power.

References

1. Holzer, M., Kutrib, M., Malcher, A.: Complexity of multi-head finite automata: Origins and directions. In: Theoretical Computer Science, vol. 412(1), pp. 83–96 (2011)
2. Buda, A.O.: Multiprocessor automata. In: Information Processing Letters, vol. 25(4), pp. 257–261 (1987)
3. Cook, M.: Universality in elementary cellular automata. In: Complex Systems, vol. 15(1), pp. 1–40 (2004)
4. Weisstein, E.W.: Universal Cellular Automaton. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/UniversalCellularAutomaton.html`