Universality of the chip-firing game

Eric Goles a, *, Maurice Margenstern b

a Departamento Ingeniería Matemática, Universidad de Chile, Casilla 170-3, Santiago, Chile
b Université de Metz, I.U.T. de Metz, and LITPI/IPB (Paris), Ile du Saulcy, 57045 Metz Cédex 1, France

Communicated by August 1995; revised November 1995
Communicated by M. Nivat

Abstract

We prove that the parallel updating of the chip-firing game on undirected graphs is universal. To achieve that, we simulate any given two-register machine by chip configurations. As corollaries, we prove that for finite graphs there exists exponential transient time to reach periodic configurations as well as exponential cycles. Also, we prove, for infinite graphs, that the reachability problem is undecidable.

1. Introduction

Let $G = (V,E)$ denote an undirected graph, each vertex being finitely many connected, but $G$ being possibly infinite. A distribution of chips is set on the set of vertices of $G$, say $x_i$ chips, $x_i \geq 0$, in vertex $i$. The local rule of the game is the following: if the number of chips, $x_i$, is not less than the degree $d_i$ of vertex $i$, the vertex gives one chip to each of its neighbors. The game can be played asynchronously, (i.e. vertices with enough chips are updated one by one) or in parallel. The last iteration mode can be defined as follows:

$$x_i(t+1) = x_i(t) - d_i 1(x_i(t) - d_i) + \sum_{j \in V} 1(x_j(t) - d_j),$$

where $x(0) \in \mathbb{N}^{|V|}$ is the initial distribution of chips and $1(u) = 1$ iff $u \geq 0$, 0 otherwise.

Fig. 1 gives an example of the parallel dynamics of a finite graph:

It is important to point out that, for finite graphs, the parallel iteration on a configuration $x(t)$, can be simulated by the sequential update of the $|V|$ vertices one by one in order $1,2,\ldots,|V|$. It is clear that after updating every vertex which has enough chips, configuration $x(t+1)$ reached by parallel updating $x(t)$ is also obtained.

* Corresponding author. E-mail: egoles@dim.uchile.cl.
The chip-firing game has independently been introduced in discrete mathematics and statistical physics and is related to many problems. In discrete mathematics it was introduced by Spencer [17] to analyze a balancing game. In physics it was introduced by Bak et al. [2] in order to model sand pile avalanches in the framework of the self organized criticality paradigm. Next is a brief survey of the main results and studies about this model.

Anderson et al. [1] studied the number of steps in function of the initial number of chips placed in one site of \( \mathbb{Z} \), which the game needs to converge to a quiescent configuration (i.e. each site has at most one chip). Later those studies were extended to general graphs by Björner et al. [5]. Tardos [18] showed that if the chip-firing game over an undirected graph terminates, then it terminates in a polynomial number of steps, depending on the number of vertices. Eriksson [6] showed that when the graph is directed, then terminating the game can be exponentially long.

From the physical point of view, the chip-firing game can be seen as a sand-pile model, as it is pointed out in [9, 10]. The sand-pile model consists of a local rule to produce avalanches in a one-dimensional non-increasing configuration of sand-grains [2]. When a local slope is greater than a critical threshold, a grain of sand tumbles to the next right pile. In [10, 11] the equivalence has been studied between this model and the one-dimensional chip-firing game proposed in [1]. Furthermore, the sand-pile point of view allows to give exact expressions, depending on the number of initial chips, on the number of steps needed to reach a quiescent configuration [11].

Most previous models are related with the asynchronous update; i.e. one fires vertices one by one in an arbitrary order. The parallel chip firing game has been studied in [3], where an invariant property is shown about the activity of vertices (i.e. the temporal pattern of firing and no firing situations). From this property, it was also proved that the parallel chip firing game converges towards periods of length at most 2 when the graph is a finite tree.

Prisner [15] studied properties of the parallel chip-firing game on directed graphs and he proved that for every strongly connected eulerian multidigraph there is a parallel chip-firing game that evolves into steady states with period equal to the length of the longest dicycle in the underlying graph. It was also conjectured that this was the longest possible period reached by the parallel chip-firing game on such digraphs. Later, this
conjecture was disproved in [13] by exhibiting chip-firing games on a planar connected undirected graph which admits nonpolynomial periodic configurations on the number of vertices.

In addition, some comments and examples were supplied in [9] in order to show that the parallel chip-firing game has a complex dynamical behavior. In fact, with small configurations, called gliders, some particular algorithms were coded in chip configurations.

The main result of our paper reads as follows.

**Theorem 1.** There is a universal parallel chip-firing game on an infinite, connected undirected graph.

Proof is made by simulating an arbitrary two register-machine on a chip-firing game. To carry out our construction, we first simulate bits 0 and 1 as well as logical gates OR, AND and negation by particular configurations of chips.

Various corollaries of those simulations are derived or proved, namely:
- the construction of exponential transients and cycles on chip-firing games in finite graphs,
- the existence of universal games on a graph with maximum degree three,
- the undecidability of the reachability problem.

2. Logical gates

First, let us remark that in a graph of degree 2 (i.e. each vertex has precisely two neighbors), the configuration with 0 chips in vertex \( k \), 2 chips in vertex \( k + 1 \) and a single chip everywhere else, is a glider shifting to the right. Symmetrically, with 2 chips in vertex \( k \) and 0 chips in vertex \( k + 1 \), a glider to the left is obtained:

\[
\begin{align*}
1 & 1 \ldots 1 0 2 1 1 \ldots \\
1 & 1 \ldots 1 1 2 0 1 \ldots 
\end{align*}
\]

And so 0 2 in a background of 1's breaks the symmetry of the graph and allows us to convey information.

It is also possible to duplicate such information by putting bifurcations as follows:

\[
\begin{array}{c}
1 1 \ldots 1 0 \ 2 \ 1 \ 1 \ldots \\
\end{array}
\]

where \( d_x \) is the degree of vertex \( x \).
We will develop this point in Section 3.

The existence of gliders suggests that information bits 1 and 0 should be encoded as follows:

That is to say, bit 1 is coded by $1 \, 1 \, 0 \, 2$ and bit 0 by $0 \, 2 \, 1 \, 1$. We have chosen this encoding to model the necessary notions of input and output in logical systems as well as to model negation. Starting from that, logical gates may be simulated as follows:

**Lemma 1.** The logical NOT, OR and AND gates are, respectively, given by the following subgraphs:

**NOT-GATE:**

**OR-GATE:**

**AND-GATE:**
Proof. The reader can easily check that after 6 steps of computation, the following configurations, with obvious notations, are true:

Let \( 1 \) denote signal \( 1102 \), let \( 0 \) denote signal \( 0211 \) and let \( n \) denote the neutral signal \( 1111 \). Hence the following rules, in obvious notations:

\[
\begin{align*}
1,0 & \rightarrow n,n \triangleleft 2,1 \\
0,0 & \rightarrow n,n \triangleleft 2,0 \\
1,0 & \rightarrow n,n \triangleleft 1,0 \\
1,1 & \rightarrow n,n \triangleleft 1,1
\end{align*}
\]

It can be easily checked that \( \triangledown \) and \( \triangledown \) behave, respectively, like OR and AND operators on partial signals \( 1 \) and \( 2 \). As \( a \lor b = \overline{a} \land \overline{b} \) and \( a \land b = \overline{a} \lor \overline{b} \), Lemma 1 is proved. \( \square \)

3. The machine structure

As known from Minsky [14], any Turing machine can be simulated by a two register-machine. Simulating register architecture will now be dealt with. As an integer stored in a register may have an arbitrary size, our registers must have an infinite structure.
To model a register machine, it suffices to build two registers together with a program controller. Therefore, we first need to define two new subgraph structures, which will be performed in Section 3.1. We shall use computer architecture conventions namely those of PLA (Programmable Logical Array), for designing subgraphs devoted to logical tasks. Other ways can be used for that purpose, but our intention is to stress the connection with architectural features.

3.1. Preliminary subgraphs

Recall that configuration (1), below, allows us to duplicate any signal 0 or 1:

![Diagram 1](image1)

By cascading, such a duplication can be extended to any power of 2 in a balanced way, using a suitable binary tree \( \mathcal{A} \), inducing the same delay in conveying for every replicate of the signal. This can be extended to any number \( n \geq 2 \), provided branches could be freely “killed” in tree \( \mathcal{A} \), only using a finite sub-graph. The simple mechanism as illustrated in (2), above, provides a way of absorbing any signal 1102 or 0211.

So a 1-wire signal can be replicated \( n \) times with the same delay. This can obviously be extended to a 2-wire signal.

3.2. Registers

We define each register in our simulating graph as cells set on a half-line. Each cell is a subgraph defined as follows.

The register uses four global signals, say \( a, s, r \) and \( d \), each one using a wire for conveying information. Each cell has a local wire, denoted \( m \), which stores signal 1 or 0 as memory unit. Recall that signal 1 means configuration \( 1^{n} \) on “parallel” branches of the graph and that signal 0 means configuration \( 0^{n} \).

Numbers are written in unary notation. For example, 4 is stored as

\[
11110\ldots0\ldots
\]

and the empty register is stored as

\[
00000\ldots0\ldots
\]
Table 1

<table>
<thead>
<tr>
<th>Case $d = 0$:</th>
<th>$s = 0$</th>
<th>$s = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = 0$</td>
<td>$d' = 0$</td>
<td>$m = 1$ $r' = 0$ $d' = 0$</td>
</tr>
<tr>
<td></td>
<td>$r' = 0$</td>
<td>$m = 0$ $r' = 1$ $d' = 1$ $s' = 0$</td>
</tr>
<tr>
<td>$a = 1$</td>
<td>$m = 1$ $r' = 0$ $d' = 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$m = 0$ $m' = 1$ $r' = 1$ $d' = 0$ $a' = 0$</td>
<td>not used</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case $d = 1$:</th>
<th>$s = 0$</th>
<th>$s = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a = 0$</td>
<td>$m = 1$ $m' = 0$ $r' = 0$ $d' = 0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$m = 0$ not used</td>
<td>$d' = 0$</td>
</tr>
</tbody>
</table>

Signals $a$ and $s$ tell the register which operation to perform as well as they order it to remain in waiting state. Signals $r$ and $d$ give an account of how the operation was performed.

The structure of a register cell is illustrated below:

![Register Cell Diagram]

Let $a, s, m, r$ and $d$ denote the input signals entering the cell and let $a', s', m', r'$ and $d'$ be the corresponding outputs. The correspondence between outputs and inputs is given by Table 1.

In order to make the tables more readable, it is assumed that not explicitly affected signals, say $\sigma$, obey the rule $\sigma' = \sigma$. In the second table, case $a = 1$ does not occur.

As an illustration, Figs. 2–5 below give the representation of a cell register in terms of PLA (Fig. 2) as well as in terms of graphs (Figs. 3–5).

Intuitively, instructions of the form *add 1 to the content of the register* are performed by sending a signal $a = 1$, $s = 0$, which “calls” each cell, passing over those which
contain signal 1 and stopping on the first occurrence of signal 0 which is changed into signal 1. The operation is completed and so signal \( r = 1 \) is sent back to the program controller. For instructions of the form *subtract 1 from the content if it is positive*, the signal is \( a = 0 \) and \( s = 1 \). The same calling occurs and it also stops on the first 0 encountered. But this time signal \( m \) is unchanged and signal \( d = 1 \) is sent back to
Fig. 4.

Fig. 5.
reach the "left" neighbor of the cell. It then changes occurring 1 into 0, except if the cell which sent signal \( d = 1 \) is the first cell of the register. In that latter case, signal \( d = 1 \) is received by the program controller (in the other cases it always receives \( d = 0 \)), which means that the content of the register was equal to 0 and, as a result, decrementing it was not possible.

The corresponding subgraph is represented above in Figs. 3 and below in Fig. 4, which have to be joined as shown below in Fig. 5.

Before turning to the program controller, it must be mentioned that the length of the path joining output \( m' \) to input \( m \) allows us to synchronize the first significant part of signal \( m \) with the corresponding parts of the other inputs.

### 3.3. Program controller

As the program controller basically behaves like a finite automaton and as its structure, described in [12] thoroughly, cannot be contained in this paper, we illustrate its general working on the simple finite automaton \( A \) defined in Fig. 6. The representation of the corresponding finite graph is given in PLA form in Fig. 7.

![Fig. 6.](image)

![Fig. 7.](image)
It should be noticed, in these figures, that an input wire is assigned to each state and to each letter of automaton $A$. Then, the automaton transition table is translated by the column of AND's in Fig. 7. The output wire of each end transmits the “firing” signal to the output state of the automaton transition table. Output wires “go back” to the former part of the graph as input wire conveying the automaton new input state, since only one of these wires conveys a 1 signal. Notice that input wires corresponding to the letter read by automaton $A$ at the current time are supposed to be fed by an appropriate device.

The program controller typically works the same way. The rôle of the automaton input letters is played by signals corresponding to which degree of completion is reached while performing the simulated instruction. An illustrative example is provided in [12].

To ensure completeness of presentation, it must be indicated how to initialize the graph. In fact, signal 0 is put at every place corresponding to a top of synchronization, except in two places: entry of wire $r$ into the program controller with signal 1 and entry of wire for instruction 0 also with signal 1; all other vertices are fitted with a single chip. This involves infinitely many chips at initial time.

4. The chip-firing game calculator

Clearly, Theorem 1 follows from Section 3.3. Besides, since the maximum number of chips on a vertex and duplication of signal and multiple entries for AND and OR gates can be carried out by cascading corresponding binary operations (see Section 3.1), one may construct a universal game with maximum degree 3 in each node. This can be summarized as follows:

Corollary 1. There is a universal parallel chip-firing game on an infinite connected undirected graph such that the degree of any vertex is not greater than 3 and such that, at any time, the greatest number of chips to be found in any vertex is at most 4.

Corollary 2. Given any finite automaton with states $\mathcal{A}$ and symbols $\Sigma$, one can simulate it on a finite graph, whose number of vertices is bounded by a quadratic function of $|\mathcal{A}|$ and $|\Sigma|$.

Proof. This directly follows from the construction of our program controller.

Corollary 3. The parallel chip-firing game on finite graphs admits exponential cycles.

Proof. The length of cycles and transitions of boolean functions which are linear over $\mathbb{Z}/\mathbb{Z}_2$ have thoroughly been studied, see for instance [8]. Exponential cycles can be observed for the following transform:

$$y_i = x_{i-1} \ XOR \ x_{i+1} \quad \text{for} \ 1 < i < n,$$

$$y_1 = x_n \ XOR \ x_2 \quad \text{and} \quad y_n = x_{n-1} \ XOR \ x_1.$$
Fig. 8 illustrates this case for $n = 3$ in PLA convention whereas Fig. 9 does the same for $n = 5$, directly in graph representation.

In Fig. 8, notice that after passing through the OR-gates, wires are duplicated into pairs of conjugate wires in terms of signals 0 and 1. This is more easily performed in Fig. 9.

These figures clearly show that the constant contained in $O(n)$ is a small one since about 40 vertices are used for each pair of XORs the number of which is $n$. 
Notice that those results improve what was proved in [13], namely that there are nonpolynomial cycles for parallel chip-firing games on finite graphs.

Moreover, as regards parallel updating, very slow convergence towards periodic configurations can be obtained.

**Corollary 4.** The parallel chip-firing game may take exponential time (in the number of vertices) to reach a periodic configuration.

**Proof.** It suffices as in Corollary 3 to notice that there exist XOR-matrices with dimension $n$ such that, starting from a boolean state $x(0)$, configuration $(1, \ldots, 1)$ is reached after $2^n - 1$ parallel steps, simulating the same number of iterations. Whence the result follows, since such matrices can be simulated by chips configurations.

The universality of parallel firing chip games clearly entails that this model admits undecidable problems. One of them is the reachability problem: given two configurations $c$ and $c'$ on the chip firing game, decide whether or not $c'$ can be reached from $c$ by the application of the chip rule.

A decidable situation of that problem was studied in a related game in [7]. For finite graphs the reachability problem is clearly decidable because starting from a finite initial number of chips, the number of configuration is finite. An efficient algorithm is given in [4]. Now, in our case, we have the following result.

**Corollary 5.** The reachability problem is undecidable on infinite, connected, undirected graphs.

**Proof.** It is well known that a two register machine may simulate a universal Turing machine [14]. Consequently, the problem to know whether starting from particular values $r_1, r_2$ in its registers the machine reaches values $r'_1, r'_2$ and halts can be reduced to the following problem of recursivity theory: is the set $\{x, y, e; \phi_e(x) = y\}$ recursive? An easy application of Rice theorem (see [16]) shows that it is not. The corollary directly follows from Theorem 1.

5. Conclusions

As we have pointed out in the introduction, parallel updating for finite graphs can be simulated by sequentially updating vertices of the current configuration in order $1, 2, \ldots, |V|$. It directly follows from this that, for this particular asynchronous dynamics (cyclically updating vertices in order $1, \ldots, n$) Corollaries 2–4 also hold.

Drawing on previous studies, with finitely many chips on an infinite graph, the parallel chip-firing game remains decidable. It is important to notice that in order to convey gliders we need to put at least one chip in infinitely many vertices in the initial configuration. Consequently, our hypotheses are necessary to obtain an undecidability result. Our present work shows that they are also sufficient.
It remains open whether the universality property of the game can be proved for planar graphs, since our construction is based upon the simulation of negation by crossing wires.

Acknowledgements

This work was partially supported by ECOS (E.G., M.M.), E.C. project in Applied Mathematics (E.G.) and FONDECYT # 1940520 (E.G.), the Laboratoire de Mathématique d'Orsay from University Paris 11.

References