Mathematics and its Applications
Central European University

# The chip-firing game on complete and complete bipartite graphs

**Natalia Bila**
MS

supervisor: Pal Hegedus


Budapest, Hungary
2017

# Acknowledgements

I would like to express my thankfulness to my supervisor, Pal Hegedus. I am very grateful to him for introducing me this topic. I also appreciate his wisdom and patience, his guidance in my writing which helped me to produce this thesis.

I also would like to express gratitude to my friend and my classmate Hailemariam for his support and faith in me which helped me to accomplish this thesis.

# Contents

# Introduction

The chip-firing game is quite new topic in the combinatorics. First appearance of it was made in the second half of 20th century. Quite astonishing is that discovering of this concept were made separately by different scientists in the different scientific areas. There are three approaches: physical, through probabilistic abacus [**?**] and combinatorics.

The aim of the given thesis is to get acquainted with the first and last approach. Also we concentrate more on the last approach where we try to investigate finiteness of the game on the complete graph and the complete bipartite graph. The results for complete graph is already known. We try to check efficiency of using the criteria for checking the finiteness of the game. Also we compare computational time for checking criteria and for the playing the game with a given initial configuration. For the complete bipartite graph we propose two criteria. One is based on similar ideas as in [11] and another one uses a little bit different thought. In addition we make simulations and we record computing time for the three methods (playing the game, criteria one and criteria two).

In the first chapter we are acquainted with the self-organized criticality and the major representative of it the Abelian Sandpile model. Here we will show what is achieved in this area and we will give an example of the Bak, Tag and Wiesenfeld sandpile model.

In the second chapter we are doing the review of the chip-firing game on undirected graphs. We will list the main theory and the theorems about this topic. Also we will make some additions in the proofs for the theorem 2.1.

In the third chapter we analyse the chip-firing game on complete graphs. We make simulations of the game with different size of vertex set and the different configurations. Also we measure the running time for playing the game and for the checking criteria. We build histograms which depict proportion of finite and infinite games in a fixed number of trials for different number of chips for the particular graph.

Also we introduce the game on complete bipartite graph. We introduce some theorems on the finiteness of the game on this graph. We propose two criteria for making the decision if a given initial configuration lead us to the termination of the game. One of them uses the similar ideas proposed in [11] and another one is based on a little bit different approach. We will compare these three methods (third method is simply play the game) in the time efficiency.

In the appendix will be given the code in Python with its descriptions for playing the chip-firing game and checking criteria on the both types of graphs.

# Chapter 1

# Physical approach of the game

The scientific investigation of self-organize phenomena is relatively new. It has been around for 30 years since it first emergence in the article of Per Bak, Chao Tang, and Kurt Wiesenfeld [1]. Self-organized criticality (SOC) is one of the most important discoveries in statistical physics in the end of 20th century. In physics, SOC is a property of dynamic systems the macroscopic behaviour of which is spatial and/or temporal scale-invariant and their structure arise without impact from outside, just internal organization of itself to criticality. When the critical point is reached, the stress in a local part is distributed to their neighbours. This action can lead those neighbours to the critical threshold and the process can be repeated.

These systems exhibit complexity as there is no scale which would describe its development. Large dynamical systems have propensity to organize themselves spontaneously into a critical state. Self-organization can expand, display interim state or maintain a stable configuration. The systems show some simple statistical properties which are defined by power laws [2]. Example [2],

$$F(s) = A \cdot s^{-\alpha}$$

where $F$ - number of events, $S$ - size of event, $A$ - some constant, $\alpha$ describes some statistical traits of SOC state.

Some dynamical systems may have the same exponent in the power law function, but they can have totally different microscopic structure.

Even if a system shows power law features it does not mean that it possesses SOC behaviour. The same logic that each healthy born dog has 4 legs, but not every creature with 4 legs is a dog.

There is no consolidate mathematical formalism which can help to recognize if the system exhibits SOC behavior or not. There are well-known mathematical models which seem to have a SOC behavior: Bak-Tang-Wiesenfeld sandpile, Curie–Weiss model [3],

earthquake model and others.

The concept of self-organized criticality is interdisciplinary. It is used to explain complex behavior of different physical, biological, chemical, social, economic [4] systems.

## 1.1 The Abelian Sandpile Model

The ASM was introduced as one of the simplest models which exhibits SOC behaviour.

The definition of sandpile model is following [5, 6]. Consider directed graph $G = (V, E)$ where the number of vertices is equal $n$. We define a variable $z_i$ on each vertex which can be assigned a non-negative integer value and is called the hight of the sandpile. Also, we define a so-called *threshold* value $\bar{z}_i \in \mathbf{N}$.

An *allowed* configuration of the sandpile is a tuple of integer heights $z_i,\ i = 1...n$. We call an allowed configuration *stable* if $z_i < \bar{z}_i\ \forall i \in V$.

An evolution of sandpile model is defined by the *toppling matrix* $\Delta$ and follows the next rules:

1 *Adding a grain.* We randomly pick a site and add a grain of sand there. Probability of choosing the site $i$ is equal to some given value $p_i$ and the sum $\sum_{i=1}^{n} p_i = 1$. The height value of a node $v_i$ is increased by 1 and the heights of other nodes remain the same.

2 *Toppling.* If on a some site we have that $z_i \geq \bar{z}_i$ then the process called *toppling* happens. The site $i$ looses some part of grains and the other sites receive some amount of grain. The law by which it happens is described by the toppling matrix $\Delta$. The configuration $z$ is updating using the next rule after toppling at site $i$:

$$z_j \to z_j - \Delta_{ij}\ \forall j \in V \tag{1.1}$$

In the case the toppling on a site $i$ evoke unstable configuration on the other sites then they are also toppled simultaneously. This procedure continues until for a configuration $z$ the following becomes true $z_i < \bar{z}_i\ \forall i \in V$.

And now we define *toppling matrix*: chapter

$$\Delta_{ii} > 0,\ \forall i \in V \tag{1.2a}$$

$$\Delta_{ij} \leq 0,\ \forall i \neq j \tag{1.2b}$$

$$b_i := \sum_j \Delta_{ij} \geq 0, \forall i. \tag{1.2c}$$

We can use a vector notation for the tuple of elements $\bar{\Delta}_i = \{\Delta_{ij}\}_{j=1..n}$. Then the formula (1.1) can be rewritten as following: $z \to z - \bar{\Delta}_i$ This formula shows that if we

have toppling on the site $i$ then the hight of the $z_i$ decreases and the hight of other nodes increase.

Now, we take a closer look to the original Bak-Tang-Wiesenfeld sandpile model. We consider this model on the two-dimensional plane surface.

$$\Delta_{ij} = \begin{cases} \Delta_{ij} + 4, & \text{if } i = j \\ \Delta_{ij} - 1, & \text{if } i \text{ and } j \text{ are adjacent} \\ \Delta_{ij}, & \text{otherwise.} \end{cases} \tag{1.3}$$

We have two types of a cell on the border. One of them is a corner cell. It has only two neighbours. So if it fires, it keeps only two grains of sand in the pile and another two are lost. Another type of the border cell is a cell which has three neighbours. So, only one grain is lost, the other three are kept in the system.

An example of sandpile model is shown on the Figure 3.9. Here we have $3 \times 3$ grid. After adding one chip on one site we get 2 topples.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 0 | 1 | 2 | 2 | 0 | 1 | 3 | 2 | 0 | 2 | 3 | 2 | 0 |
| 3 | 3 | 1 | 1 | 3 | 4 | 1 | 1 | 4 | 0 | 2 | 1 | 0 | 1 | 2 | 1 |
| 2 | 2 | 3 | 2 | 2 | 2 | 3 | 2 | 2 | 3 | 3 | 2 | 3 | 3 | 3 | 2 |
| 1 | 3 | 0 | 2 | 1 | 3 | 0 | 2 | 1 | 3 | 0 | 2 | 1 | 3 | 0 | 2 |

Figure 1.1: Sandpile model on $3 \times 3$ grid

If we have *unstable* configuration, meaning there exist a vertex $i$ such that $z_i \geq \bar{z}_i$, then we have a toppling and the new obtained configuration is $t_i z = z - \bar{\Delta}_i$. The $t_i$ stands for the toppling operator.

The set of consecutive toppling is called an *avalanche*.

The order in which the topplings happen does not matter. The stable configuration which obtained after firing overloaded vertices does not depend on firing interim nodes. This is the reason why this model is called *Abelian Sandpile model*. In mathematical notation this can be written as: if $z$ is such that $z_i > \bar{z}_i$ and $z_j > \bar{z}_j$ then

$$t_i t_j z = t_j t_i z$$

Let us consider unstable configuration with two overloaded sides $i$ and $j$. We start toppling at site $i$ by the rule 1.1 and after the toppling the site $j$ we obtain $z \to z - (\bar{\Delta}_i + \bar{\Delta}_j)$. The last expression is symmetric under exchange of $i$ and $j$. We get the same final configuration regardless of the order of topplings $i$ and $j$. The same logic are used when we have more than two topplings. We also obtain the same final configuration if toppling at site $i$ is followed by addition a particle in configuration $j$.

# Chapter 2

# Chip-firing game on undirected graphs

In 1986 J. Spencer was studying a certain problem and he used a so-called "balancing game" for its solution. At the beginning we have a pile of $N$ chips in the middle of an infinite line. We divide this pile into two parts of equal sizes $\lfloor N/2 \rfloor$ and put one part to the left side and another part to the right side. In case we have an odd number of chips, we leave one chip in the middle and do the same procedure as above for the even case. Then we continue with this procedure. It was proven by Spencer that this process is well approximated by the "Galton process" for the first $N$ steps.

Taking this fact in consideration, further extension and investigation of this problem were made by E. Tardos et al [7]. They modified the game allowing only to move one chip at a time either to the left or to the right. They described the precise number of steps needed to terminate the game.

Later Björner et al [8] studied this game on simple graphs. They called it *chip-firing game*. The game starts with setting some amount of chips on each vertex. If a vertex contains more chips than its degree, then it distributes a one chip to each of its neighbours. This game continues until all nodes are loaded with fewer number of chips than their degree. If we get repetition of some configuration then this game is infinite, this will be described later.

## 2.1 The chip-firing game

We introduce chip-firing game in mathematical notations. Let $G$ be a simple graph with finite number of nodes $\{v_1, v_2, ..., v_n\}$ (we will use the notions vertex and node interchangeably throughout this thesis). We define *configuration* as a function:

$$\phi : V(G) \to \mathbb{N} \cup \{0\}$$

which assigns the number of chips for each vertex.

We put $\phi(v_i) = c_i$ chips on node $v_i$ for each $i \in \{1, 2, ..., n\}$ and the sum $\sum_i \phi(v_i) = N$.

That will be called an *initial configuration* of the game. We say that a vertex is *ready* if it is loaded with a pile of chips of size equal or bigger than the number of its neighbours. The ready node gives a chip to each adjacent node. We call this procedure *firing*. The game can be either *finite* or *infinite*. It stops if each node in the graph has fewer chips than its degree. We denote the degree of a vertex $v_i$ in usual notations $d(v_i)$. We represent an initial configuration as $\alpha_0 = (\phi(v_1), \phi(v_2), ..., \phi(v_n))$. Each subsequent configuration we denote $\alpha_k$ which was obtained from $\alpha_{k-1}$ by firing some overloaded node $v_j$ in $v\alpha_{k-1}$.

$$\alpha_k(i) = \begin{cases} \alpha_{k-1}(i), & \text{if } v_i \text{ not adjacent with } v_j; \\ \alpha_{k-1}(i) + 1, & \text{if } v_i \text{ adjacent with } v_j; \\ \alpha_{k-1}(i) - d(v_i), & \text{if } v_i = v_j. \end{cases}$$

We call the sequence of configurations $(\alpha_0, \alpha_1, ...)$ as a *fired sequence*, it is accompanied by a corresponding *fired vertex sequence*. By *legal step*, we mean firing one overloaded node at a time and distribute one chip to each of its neighbours. Sometimes firing sequence is also called *legal game*.

The next theorem is of a great importance in the theory of chip-firing games. It was introduced and proved by Björner, Lovász, Shor in their article [8].

**Theorem 2.1** *[8] Given a connected graph and initial distribution of chips, either every legal game can be continued indefinitely, or every legal game terminates after the same number of moves with the same final configuration. The number of times a given node is fired is the same in every legal game.*

In order to proove this theorem we need to define the theory of a language $\mathcal{L}$ and some properties of it which were introduced in [8].

The language $\mathcal{L}$ is a set of fired vertex sequences of all legal games. Consider a finite set $V$ and a language $\mathcal{L}$ over $V$ is a set of finite words which are formed from the elements of $V$. If we delete some letters of a word $\alpha$ in any order, we obtain a *subword* of this word. For example, if we have a string *abbdabbd* then *ada* is a subword.

Also, we define the length and the score of the word. The former we denote by $|\alpha|$ and the latter as $[\alpha]$. By the score of the word we mean a vector $s$. Each entry $s_i$ of it shows the number of occurrence of the letter $i$ in the given word. The following three properties of the languages were introduced in [8], these hold for the language of fired vertex sequence:

- $\mathcal{L}$ is *left-hereditary*, if whenever word $\alpha$ belongs to $\mathcal{L}$ then all the beginnings of it belongs to this language. If for example we have a word *badddba* in $\mathcal{L}$ then *badd* $\in \mathcal{L}$. So if our fired vertex sequence is legal then every prefix subsequence of it is legal;

- $\mathcal{L}$ is *locally free* if the next statement is true:
  Let $\alpha \in \mathcal{L}$ and $x \neq y$, two elements of the set $V$ such that $\alpha x$ and $\alpha y$ belong to $\mathcal{L}$. Then $\alpha xy \in \mathcal{L}$. In terms of fired vertices this means if two nodes are ready then we can fire the first one and after it we still can fire second one.

- $\mathcal{L}$ is *permutable* if the following holds:
  Whenever $\alpha$ and $\beta \in \mathcal{L}$, they have the same score and $\alpha x \in \mathcal{L}$ then it is also true $\beta x \in \mathcal{L}$. In the language of fired vertex sequences it means that it does not matter in which order we fire the nodes as long as every step is legal. What only matters is the number of times each node has fired in the fired sequence. For example, let $\alpha = abcbaca$ and $\beta = bacbaca$ , if $abcbacab \in \mathcal{L}$ then $bacbacab \in \mathcal{L}$

These three properties lead to the so-called "strong" exchange property, which is a strong version of the exchange property in greedoids theory [8, 9].

**Strong exchange property**:
If two words $\alpha$ and $\beta$ belong to $\mathcal{L}$ then there exists a subword $\alpha'$ of the word $\alpha$ such that a string $\beta\alpha' \in \mathcal{L}$ and the score of this new word is a point-wise maximum of $\alpha$ and $\beta$.

We call the word *basic* if it is not a prefix of any other word in $\mathcal{L}$. Two words $\alpha$ and $\beta$ are *equivalent* if for any string $\gamma$, $\alpha\gamma \in \mathcal{L}$ if and only if $\beta\gamma \in \mathcal{L}$. The equivalence classes of such words are called *flats*. A *subflat* of the flat $f$ is a flat $g$ such that every word in $g$ can be extended to some word in $f$.

**Lemma 2.2** *Let $\mathcal{L}$ a left-hereditary, locally free and permutable language. Then the following hold:*

1. *Language $\mathcal{L}$ has the strong exchange property. And, conversely, if the language has the strong exchange property then it is permutable and locally free;*

2. *If $\mathcal{L}$ has a basic word, then all basic words have the same length;*

3. *If $\alpha$ and $\beta$ are basic words then they have the same scores $[\alpha] = [\beta]$;*

4. *If the language is finite and $\alpha, \beta \in \mathcal{L}$, then $[\alpha] \leq [\beta]$ if and only if the flat of $\alpha$ is a subflat of the flat $\beta$. And hence the same score $[\alpha] = [\beta]$ iff $\alpha \sim \beta$.*

**Proof.**

1. $\Rightarrow$ Here we use the notion of $l_1$-norm. $|x|_1 = \sum_{r=1}^{n} |x_r|$, where $x = (x_1, x_2, ..., x_r)$. The proof will be based on induction by $|[\alpha] \vee [\beta]|_1$. Let assume that the strong exchange property for $|[a]|_1 < |[\alpha] \vee [\beta]|_1$ is satisfied (where $\vee$ indicates point-wise maximum).

Let pick $\alpha'$ such that it consists of those letters $i \in \mathcal{L}$ which have the score $[\alpha]_i > [\beta]_i$. The number of occurrence of these letters in $\alpha'$ equal $[\alpha]_i - [\beta]_i$.

The fact that $[\beta\alpha'] = [\alpha] \vee [\beta]$ is straightforward. If for some letter $x$ we have $[\alpha]_x < [\beta]_x$, we don't include it in $[\alpha']$, so for this letter in the score vector we have an entry from the $[\beta]$ vector. If $[\alpha]_x > [\beta]_x$ then $\alpha'$ includes precisely $[\alpha]_x - [\beta]_x$ number of times letter $x$. Adding up $[\alpha]_x - [\beta]_x + [\beta]_x$ we get that the score for $x$ in $\beta\alpha'$ equal $[\alpha]_x$.

It will be shown that $[\beta\alpha'] \in \mathcal{L}$.

We pick $\alpha$" to be the longest beginning of the word $\alpha'$ such that $\beta\alpha'' \in \mathcal{L}$. We prove by contradiction assuming that $\alpha'' \neq \alpha'$. Let $x$ be the first letter after $\alpha''$. It means that the number of it occurrence in $\alpha$ is strictly bigger than in $[\beta\alpha'']$. So we take prefix $\alpha_1$ of the word $\alpha$ such the next letter after it in $\alpha$ is $x$. Now the total number of $x$ in $\alpha_1$ is the same as in the $\beta\alpha''$.

The score vector of $\alpha_1$ is point-wise less or equal the score vector of $\beta\alpha''$. Assume that it is not true and we have a letter $y$ in $\alpha_1$ such that $[\alpha_1]_y > [\beta\alpha'']_y$. It means that letter $y$ occur in $\alpha'$ some number of times before marked $x$. As we have matching of letters between $\alpha'$ and $\alpha''$ till the end of the last one then $\alpha$" have the same score for $y$. So the number of $y$ in $\beta\alpha''$ is at least the same quantity as it in $\alpha_1$. We have contradiction, so no such $y$ exist in $\alpha_1$.

As $[\alpha_1] \leq [\beta\alpha'']$, we have that the point-wise maximum of this two vectors equal to vector $[\beta\alpha'']$. As the $l_1$ norm of $[\beta\alpha'']$ is less than $l_1$ norm of $[\beta\alpha']$ then by induction it is applicable strong exchange property. We find subword $\gamma$ of $\beta\alpha''$ such that $\alpha'\beta$ belongs to our language $\mathcal{L}$ and the score of it equal the score of $\beta\alpha''$. There is no $x$ inside $\gamma$, because $[\alpha_1]_x = [\beta\alpha'']_x$. As $\alpha_1 x$ and $\alpha_1\gamma$ belong to $\mathcal{L}$ then by local free property $\alpha_1\gamma x \in \mathcal{L}$. As the score vectors of $\alpha_1\gamma$ and $\beta\alpha$" are equal and $\alpha_1\gamma x \in \mathcal{L}$ then by permutable property $\beta\alpha"\gamma$ also belongs to the language. But this is contradiction to our choice of $\alpha$".

$\Leftarrow$ Consider a language $\mathcal{L}$ with strong exchange property.
Let $\alpha = \gamma y$ and $\beta = \gamma x$, where $\gamma$ is some word in $\mathcal{L}$ and $x \neq y$, such that both of them belong to $\mathcal{L}$. As strong exchange property is applicable in this language then there exists subword $\alpha'$ of $\alpha$ such that $\beta\alpha' \in \mathcal{L}$. We now find point-wise maximum of the $\alpha, \beta$ words. $\beta$ has more letters $x$ than $\alpha$ and $\alpha$ has more letters $y$ than $\beta$. So we can conclude that $[\beta\alpha']_x = [\beta]_x$ and $[\beta\alpha']_y = [\alpha]_y$. The score for other letters are the same in the both words. We put in $\alpha'$ those letters z from $\alpha$ which are preceded by at least $[\beta]_z$ occurrences of $z$.That is true only for last letter $y$ in $\alpha$. Then in $\alpha'$ we can only have one letter $y$. So we showed that if $\gamma x$ and $\gamma y$, $x \neq y$, belong to $\mathcal{L}$ then $\gamma xy \in \mathcal{L}$. That is locally free property.

For the permutable property, let assume that $[\alpha x] \in \mathcal{L}$, $\beta \in \mathcal{L}$, $[\alpha] = [\beta]$. By contradiction, let $\beta x \notin \mathcal{L}$. Applying the strong exchange property for $\beta$, we can find $\alpha'$ such that the score of the word $[\beta\alpha'] = [\alpha] \vee [\beta]$. We have $\alpha' = x$, as only for $x$ we have $[\alpha x]_x > [\beta]_x$ (for other letters the score of both words are equal) and $[\alpha x]_x - [\beta]_x = 1$. This imply that $\beta x = \beta\alpha' \in \mathcal{L}$. We get a contradiction with our assumption, that is why if $[\alpha x] \in \mathcal{L}$, $\beta \in \mathcal{L}$, $[\alpha] = [\beta]$ then $\beta x$ is also in $\mathcal{L}$. So the permutable property is satisfied.

2. We prove 2, 3 in one item. Let assume that we have two basic words $\alpha$ and $\beta$ with different length, $|\alpha| < |\beta|$. We apply strong exchange property: we find $\alpha' \in \mathcal{L}$ such that $\beta\alpha' \in \mathcal{L}$. $\alpha'$ is not empty string as $\alpha$ is not a subword of $\beta$, as basic word. But then $\beta$ is a proper prefix of $\beta\alpha'$. So basic words should have the same length.
   The same if we have basic words of the same length, but with different scores. Then again $\alpha'$ is not an empty string. And with the same logic as above, $\beta$ can't be a basic word. So $[\alpha] = [\beta]$.

3. -

4. Let us assume that $[\alpha] \leq [\beta]$. Using strong exchange property we can find a subword $\gamma$ of $\beta$ such that the score of $\alpha\gamma \in \mathcal{L}$ equal to the score of $\beta$. By permutable property whenever $\alpha\gamma x$ belongs to $\mathcal{L}$ then $\beta x \in \mathcal{L}$. Which means by the definition that $\alpha$ and $\beta$ are equivalent, so they belong to the same equivalence class. This is shows that any word in a flat defined by $\alpha$, we call it $f$, can be extended to a word in a flat defined by $\beta$, we name it $g$. That is why $f$ is a subflat of $g$.

   In other way, let $\alpha$ defines subflat $f$ of the flat $g$ which is defined by $\beta$. Let $\alpha\gamma$ be an extension of $\alpha$ in the equivalence class $g$. Let $\alpha\gamma\delta$ be an extension of $\alpha\gamma$ to a basic word (as language is finite, there exist basic words). Then $\beta\delta \in \alpha$ by the definition of equivalent words. It has the same length as $\alpha\gamma\delta$, so it is also a basic word. But this means that $[\alpha\gamma\delta] = [\beta\delta]$ and $[\alpha] \leq [\beta]$.

□

**Proposition 2.3** .

1. *The fired vertex sequences of legal games constitute a language which is left-hereditary permutable and locally free;*

2. *If the language is finite then two legal games lead to the same configuration iff they have the same score vector.*

**Proof.**

1. It is obvious that the language is left-hereditary, every fired vertex subsequence is a fired vertex sequence. For the locally free property if we have two overloaded nodes after playing game $\alpha$ then after firing the first one, let it be $x$, the second one $y$ will have the same, or bigger by one, number of chips. So it still has to fire. That means $\alpha xy$ a legal game and it belongs to the language. For the permutable property we have if two legal games $\alpha, \beta$ have the same score then they are equivalent. So for the fired vertex sequence $\gamma$ if $\alpha\gamma$ is a legal game then $\beta\gamma$ is also a legal game.

2. $\Rightarrow$ Two legal games $\alpha, \beta$ lead to the same configuration. We continue the game $\alpha$ with a game $\gamma$, if it is a legal game then so is a game $\beta\gamma$. Which means this games are equivalent and by the Lemma 2.2
   $\Leftarrow$ When the games have the same score then they lead to the same configuration as it was proved in first item.

□

**Proof of theorem 2.1**

**Proof.** If the game is finite then it has basic words which are of the same length and the same score, and they lead to the same final configuration. □

## 2.2 The finitness of the game

We are interested what can we say about the duration of the game. Can we reach the configuration where is no node which is ready to fire? Firstly, we familiarize ourselves with the theory of the finiteness of the chip-firing game depending on number of chips loaded on a graph at the beginning of the game. Here we are consider simple connected graph $G$ with $n$ vertices and $m$ edges. The game will be played with $N$ chips.

The following lemmas and theorem, as well as their proofs, were introduced by Björner et al [8] and Tardos [10].

**Lemma 2.4** *In the infinite chip-firing game every vertex fires infinitely often.*

**Proof.** As the game is infinite then there is a vertex $v$ which fires infinitely often. Consider the neighbour $u$ of it. Whenever $v$ is firing, vertex $u$ receives a chip. In our game we have only $N$ chips on the nodes of a graph which are redistributed between the nodes at each step. So at some moment the vertex $u$ should fire as it can't accumulate more than $N$ chips and this will happen infinitely often. As the graph is connected then every vertex will fire infinitely often. □

**Lemma 2.5** *The chip-firing game is finite if it has a vertex which is not fired at all.*

**Proof.** [10] It will be shown that if all nodes have fired during some period of the game then this game is played indefinitely. We consider a vertex $v$ which has been inactive as long as possible. Then each neighbour of it have fired and this node have received at least as many chips as its degree. Then this node can fire again. □

**Theorem 2.6** *[8]*

1. *If $N < m$ then the game is finite;*
2. *If $m \leq N \leq 2 \cdot m - n$ then there exists an initial configuration guaranteeing finite termination and also one guaranteeing infinite game;*
3. *If $N > 2 \cdot m - n$ then the game is infinite.*

# Chapter 3

# Chip-firing game on complete and complete bipartite graphs

## 3.1 The chip-firing game on complete graph

We consider the game on complete graph $K_n$. If we put on the nodes of a graph less than $\binom{n}{2}$ chips, the game will be finite by theorem 2.6. And of cause it will be infinite if we have more than $2 \cdot \binom{n}{2} - n = n^2 - 2 \cdot n$.

We call an initial configuration *finite* if it leads to termination of the game. And we call it *infinite* in the opposite case.

**Theorem 3.1** *[11] Let $\alpha = (0, 1, 2, ..., n-1)$ be a configuration on $K_n$. If a configuration $\beta$ can reach an equivalent configuration $\alpha'$ of $\alpha$ by firing a sequence of vertices, then $\beta$ is an equivalent configuration of $\alpha$.*

**Proof.** [11] Let us assume that $\beta$ reaches configuration $\alpha'$ after firing the sequence of configurations $(\beta, \alpha_0, \alpha_1, ..., \alpha_k, \alpha')$. As graph vertex and edge symmetric we assume that we reach configuration $\alpha'$ by firing vertex $v_1$. Each node received one chip after firing $v_1$, so $c_i > 0, i = 2, .., n$. As $\alpha'$ equivalent to $\alpha$ we must have one node with 0 chips. This means on vertex $v_1$ there is no chips. In other words $\alpha'(1) = 0$ and $\alpha_k(1) = n - 1$. For other vertices we have $\alpha_k(i) = \alpha'(i) - 1, i = 2, ..., n$. This shows that $\alpha_k$ and $\alpha$ are equivalent. Using the same logic for each configuration in the sequence we obtain that all $\beta, \alpha_0, \alpha_1, ..., \alpha_{k-1}$ are equivalent to $\alpha$. This finishes the proof. □

Next criteria is applicable to claim if initial configuration is $infinite$ for case when $N = \binom{n}{2}$

**Theorem 3.2** *[11] A chip-firing game on $K_n$ with initial configuration $\alpha$ and $N = \binom{n}{2}$ is infinite iff $\alpha \cong (0, 1, 2, ..., n-1)$*

**Proof.** $\Leftarrow$ Let us numerate nodes as $v_1, v_2, ..., v_n$. Each node has the number of chips accordingly to its index in $(0, 1, 2, ..., n-1)$. The node $v_n$ with $n-1$ chips fires as degree of every node in $G$ equals $n-1$. Each node, except $v_n$, receives 1 chip and the fired node finishes with 0 chips. Then $v_{n-1}$ obtained 1 chip and now it has $n-1$ chips, so it fires. We continue this process, and, after firing $v_2$, node $v_1$ storage $n-1$ chips, as $n-1$ nodes fired before. This implies that each node has fired. By the lemma 2.5, the configuration $\alpha$ is infinite.

$\Rightarrow$ [11] Consider configuration $\alpha_0 = (c_1, c_2, ..., c_n)$ be an infinite on a complete graph with $n$ nodes, such the sum of all chips is equal $\frac{n\cdot(n-1)}{2}$. We order the number of chips from the smallest one to the biggest one, we renumbering indexes in such way that the node $v_1$ will receive the smallest number of chips and the node $v_n$ the biggest, $c_1 \leq c_2 \leq ...c_n$(we can do so, as graph is symmetric). As $\alpha_0$ is an infinite configuration then vertex $v_n$ has more than or equal to $n-1$ number of chips.

The game is played using two rules:

1. We firstly fire the node with maximum number of chips;
2. In case of having several nodes with maximum number of chips, pick a node with maximum index.

It will be proven that $c_1 = 0$.

By contradiction, let us assume that $c_1 > 0$. As the configuration $\alpha_0$ is infinite then we have fired sequence $(\alpha_0, \alpha_1, ..., \alpha_k, ...)$ with fired vertex sequence $(v_{i_1}, v_{i_2}, ..., v_{i_k}, ...)$. Let $v_{i_k} = v_1$ and $v_{i_j} \neq v_1, j < k$. Then, as amount of chips on this vertex is the smallest one, $k \geq n$, as each vertex has fired at least once before vertex $v_1$ fires. Now we make a little change in the game, we take off one chip from vertex $v_1$. The new configuration will be denoted as $\alpha_0'$. We play the game firing along the sequence of vertices $v_{i_1}, v_{i_2}, ..., v_{i_{k-1}}$. The correspondent fired sequence of configurations is denoted as $\alpha_0', \alpha_1', ..., \alpha_{k-1}'$. As $k-1 \geq n-1$ then after firing a vertex in configuration $\alpha_{k-2}$ in a following configuration $\alpha_{k-1}$ vertex $v_1$ has at least $n-1$ chips. So it can be fired. This shows that every vertex is fired and configuration $\alpha_0$ is infinite by Lemma 2.5.

But we have that sum of the chips equals $\binom{n}{2} - 1$. By the Theorem 2.6, $\alpha_0'$ is finite configuration. But this is a contradiction. So $c_1$ should be 0.

As we play using two rules of maximum number of chips and maximum subscript then $v_{i_1} = v_n$. After firing a node $v_n$ in configuration $\alpha_0$ each of the other vertices receives a chip. But we know that acquired configuration also infinite that is why it

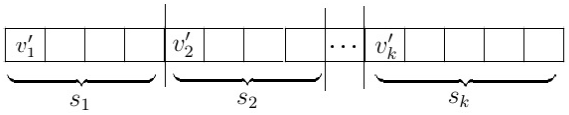
should have one vertex with 0 chips as it was shown above. This implies that $\alpha_1(n) = c_n = 0$ and then $\alpha_0(n) = c_n = n-1$. By the same logic we have that $v_{i_2} = v_{n-1}$ and $\alpha_2(n-1) = 0$ & $\alpha_0(n-1) = n-1$. Continue till the configuration $\alpha_{n-1}$ where we get that $\alpha_{n-1}(2) = 0$ & $\alpha_0(2) = 1$. Now we can summarize it as $\alpha_0 = (0, 1, 2, ...n)$ □

Next Lemma shows that we can reach configuration where we have a bound on a size of a chips pile on a node in the case $\binom{n}{2} \leq N \leq 2 \cdot \binom{n}{2} - n$. This idea will be used in the criteria for finiteness of the game on complete graphs.

**Lemma 3.3** *[11] Let $\alpha$ be an initial configuration of $K_n$ with $N$ chips, $\binom{n}{2} \leq N \leq 2 \cdot \binom{n}{2} - n$. Then by firing some sequence of vertices starting at $\alpha$, we can reach a configuration $\beta$ such that $\beta(i) \leq 2 \cdot n - 3$ for each vertex $v_i \in V(K_n)$.*

**Proof.** [11] The idea of proof is that we consider another configuration $\alpha'$ such that on each node we have $k$ chips if in $\alpha$ it has either $2 \cdot k$ or $2 \cdot k + 1$ chips. So if it two times less than in initial configuration then upper bound for the quantity of chips in the game is $\binom{n}{2} - \frac{n}{2}$. By the 2.6 the game with $\alpha'$ configuration is finite. Consider a sequence of firing nodes $v'_1, v'_2, ..., v'_p$ in $\alpha'$ such that after firing all of them the game terminates. If we return to the initial game then let us fire the same sequence of nodes, but doubling the number of firings of each, meaning $v'_1, v'_1, v'_2, v'_2, ..., v'_p, v'_p$. Let the corresponding fire sequence be $(\alpha, ..., \beta)$. Then the last configuration is the one we have searched for. In the game, with $\alpha'$ as starting arrangement, in the end each node has $\phi(v_i) \leq n-2, i = 1, ..., n$ chips then going backward, doubling the number of chips or doubling and adding 1, in $\beta$ we get $\phi(v_i) \leq 2 \cdot n - 4$ or $\phi(v_i) \leq 2 \cdot n - 3$. This finishes the proof. □

We consider now that amount of chips belongs to $(\frac{n \cdot (n-1)}{2}, n^2 - 2 \cdot n)$ and we are in configuration where nodes can fire no more than twice. Order $\phi(v_i), i = 1, ..., n$ in the descending order and renumbering indexes from 1 to $n$. We separate a sequence of vertices $v_1, v_2, ..., v_n$ in blocks by the following rule. When we have that $\phi(v_i) - \phi(v_{i+1}) \geq 2$ then we make a separation point and $v_i$ becomes the end of one block and $v_{i+1}$ becomes the beginning of another. Let assume that we have $k$ blocks. We denote the beginning node of each block as $v'_i,\ i = 1, ..., k$. Denote the amount of chips on these nodes by $c^*_i,\ i = 1, ..., k$. The number of vertices in each block $i$ will be represented by the $s_i$.




**Property 3.1** *[11] If we have a vertex that can fire at least three times, then the game is infinite.*

The next theorem shows criteria by which we can determine the finiteness of the game without actually playing it. We take in consideration simple, complete graphs with initial configuration $\alpha_0 = (\phi(v_1), \phi(v_2), ..., \phi(v_n))$, and amount of chips in range $\binom{n}{2} \leq N \leq 2 \cdot \binom{n}{2} - n$. We arrange chips in the descending order $\phi(v_1) \geq \phi(v_2) \geq ... \geq \phi(v_n)$ and $\alpha_0$ is separated in $k$ blocks by the above procedure. The amount of chips on the vertex is bounded by amount of $2 \cdot n - 3$ chips. The number of vertices in each block is $s_i,\ i = 1..k$ and amount of chips on first vertex of each block is $c_i^*,\ i = 1, ..., k$.

**Theorem 3.4** *[11] A chip-firing game with initial configuration $\alpha_0$ is finite if and only if there exists two integers $i, j \in \{1, 2, ..., k\}$. $j \leq i \leq k$ such that $c_i^* + \sum_{b=1}^{i-1} s_b + \sum_{c=1}^{j-1} s_c < n-1$, $c_j^* + \sum_{b=1}^{i-1} s_b + \sum_{c=1}^{j-1} s_c - n < n-1$*

**Proof.** $\Rightarrow$ We assume that the game is finite. We know that any node can't fire more than twice with the written above conditions. We consider two sets with blocks of nodes. First set, which we denote $H_1$, consists of such blocks whose vertices have fired al least once. Second set $H_2$ includes blocks with nodes which have fired twice. $H_1 = \{b_1, b_2, ..., b_{i-1}\},\ H_2 = \{b_1, b_2, ..., b_{j-1}\}$. It is obvious that $j \leq i$ as if nodes fired twice then they had a first firing, conversely is not true. As the game is finite then there is a vertex which didn't fire at all by Lemma 2.5 and in our case at least the last block shouldn't fire. Then starting from the block $b_i$ there is no firing at all and there is no more then one firing starting from the block $b_j$. It could be written as $c_i^* + \sum_{b=1}^{i-1} s_b + \sum_{c=1}^{j-1} s_c < n-1$ and $c_j^* + \sum_{b=1}^{i-1} s_b + \sum_{c=1}^{j-1} s_c - n < n-1$

$\Leftarrow$ Let $i, j \in \{1, 2, ..., k\}$ be two minimal integers such that the next inequalities are satisfied $j \leq i \leq k$, $c_i^* + \sum_{b=1}^{i-1} s_b + \sum_{c=1}^{j-1} s_c < n-1$, $c_j^* + \sum_{b=1}^{i-1} s_b + \sum_{c=1}^{j-1} s_c - n < n-1$. These inequalities show that the vertex $v_i'$ can't fire at all and $v_j'$ can't fire twice. If $j = 1$, this means no node has fired twice. In this case, as $k$ can't be more than $n$, we have less than $n$ firings and by Lemma 2.5 the game with initial configuration $\alpha_0$ is finite. In another case, when $j \neq 1$, we have that if inequality of not firing at all is satisfied for $i$ then it is also satisfied for the last block $k$.

By the initial ordering $c_1^* \leq c_k^*$ and by the conditions of theorem $c_1^* \leq c_k^* + 2 \cdot n - 3$. From here we derive that $0 \leq c_1^* - c_k^* \leq 2 \cdot n - 3$. Substituting $-c_k^*$ by $-n+1+\sum_{b=1}^{i-1} s_b + \sum_{c=1}^{j-1} s_c$ in the

last inequality we get $c_1^* + \sum_{b=1}^{i-1} s_b + \sum_{c=1}^{j-1} s_c - n + 1 \leq 2 \cdot n - 3$ or $c_1^* + \sum_{b=1}^{i-1} s_b + \sum_{c=1}^{j-1} s_c - 2 \cdot n \leq 2n - 1$. The last one shows that node $c_i^*$ can't fire more than twice. So, no one of the vertices $v_1', v_i', v_j'$ is firing at this moment. This shows that $\alpha_0$ is finite. □

### 3.1.1 Computational time of the game

In this section we compare computational time for playing the game and the time for checking inequalities in order to determine if the game terminates.

The code of the program given in the Appendix A. Firstly, let us describe how the game is played. When the program starts it asks to enter the number of vertices $n$. The input should be an integer. Next, it is asked if you would like to see visualisation of the process. But here we have restrictions, the game can be shown only for the graphs with no more than 40 nodes. The number of chips which is put on the nodes in the graph can be generated by a program or it can also be set manually. It is also possible either randomly set a configuration of chips on the nodes or to do it manually.

In the game we find nodes with the number of chips equal or more than their degrees. We put them in the list named by "overload nodes". It takes $O(n)$ time to finish this procedure. After that we are going through the newly formed list and we are making firings. After firing the node we delete it from the list. If further we still have firing nodes we add them to the list and repeat this procedure. We fire a node while it possible at once. But if we fired $n$ or more times the nodes the program stops and we are saying that the game is infinite. If we fired less than $n$ times and the list of overloaded nodes is empty, then we print the game is finite. It also takes $O(n)$ time to compute this part of a program. We can conclude that overall it takes $O(n^2)$ time to play the game.

We played the game for the graph with 300 and 600 nodes Figure 3.1, 3.3. For each type of a graph we went through different number of chips in the game starting with a lower bound $\binom{n}{2}$ and ending with an upper bound $2 \cdot \binom{n}{2} - n$. For each amount of chips we made 200 trials and the peaks on plots show their averages. Also we included error bars.

The plot shows that when the number of chips increasing then the time for deciding if the game finite or not also increasing gradually for checking criteria. If we look at the graph for the playing the game on the plots 3.1, 3.3 we see that it is strictly increasing to some point and then it starts to decrease. The reason of it is the following. When we encounter an infinite game with the small amount of chips in the game, we have more firings of nodes one in a time, but with increased number of chips we have more nodes which can fire twice and more in a time. As in our algorithm we fire the node at once as

many times as possible then we use less time to figure out that we have $n$ or more firings, so the game is infinite.

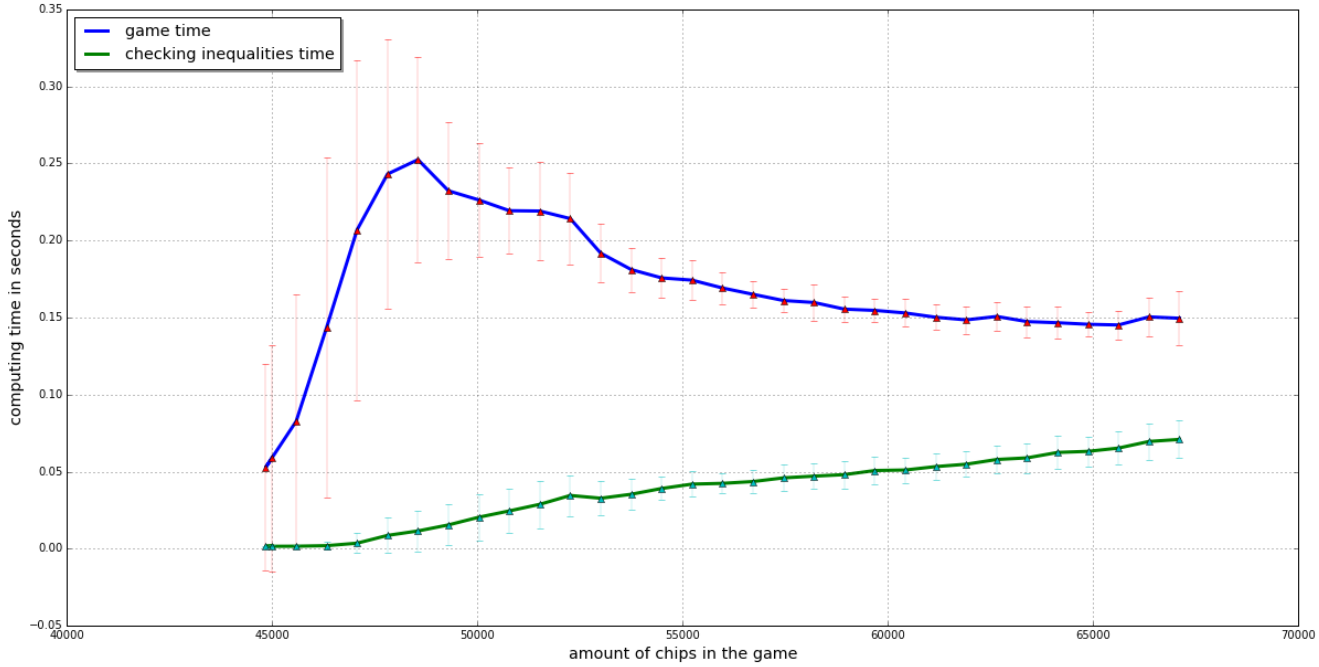



Figure 3.1: Time for playing game and checking criteria with $n = 300$

Now, we analyse the computational time of checking criteria for making conclusion of the finiteness of the game Figure 3.1, 3.3. Firstly, we are firing nodes till we bound amount of chips on nodes by $2 \cdot n - 3$. The while loop runs less than $\frac{n}{2}$ times and inside this loop we are going through each node and we add a chip on each node. Partition nodes into the blocks takes also $O(n)$ time. In the for loop we are going through the list which contains the number of vertices in each block. This for loop includes another nested for loop. The former loop looks for the number of block which doesn't fire at all and the latter looks for the number of block which does not fire twice. This takes less than $n^2$ steps. So, overall to find these two numbers which satisfy the inequalities or show that there no such it takes $O(n^2)$ steps.

So, the running time of two methods are virtually equivalent and the way how we implement them into the program code will give difference in computing time. We can see this in the simulation of the game on complete graph with $n = 600$ nodes. Here we can see that the time performance for some number of chips is better for the first method and for another number of chips it is better for the second method 3.3, 3.5.

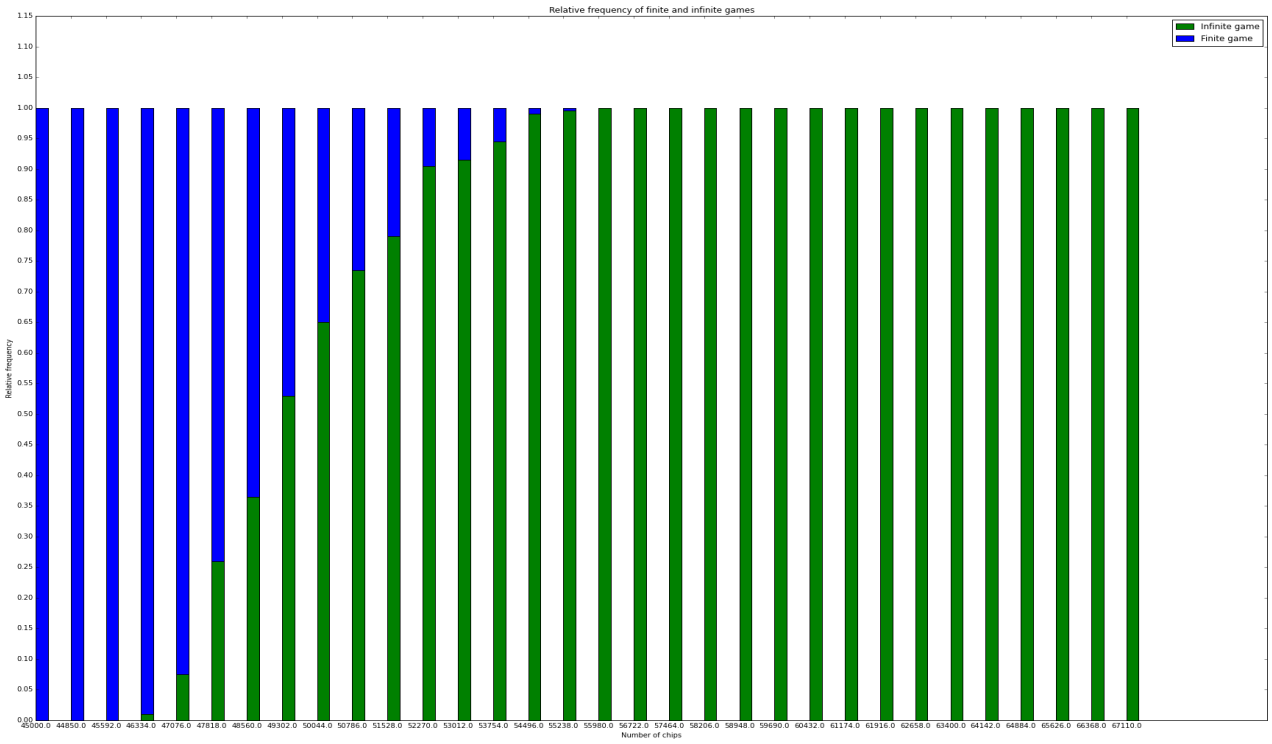



Figure 3.2: Bar chart for ratio of finite and infinite games for $n = 300$

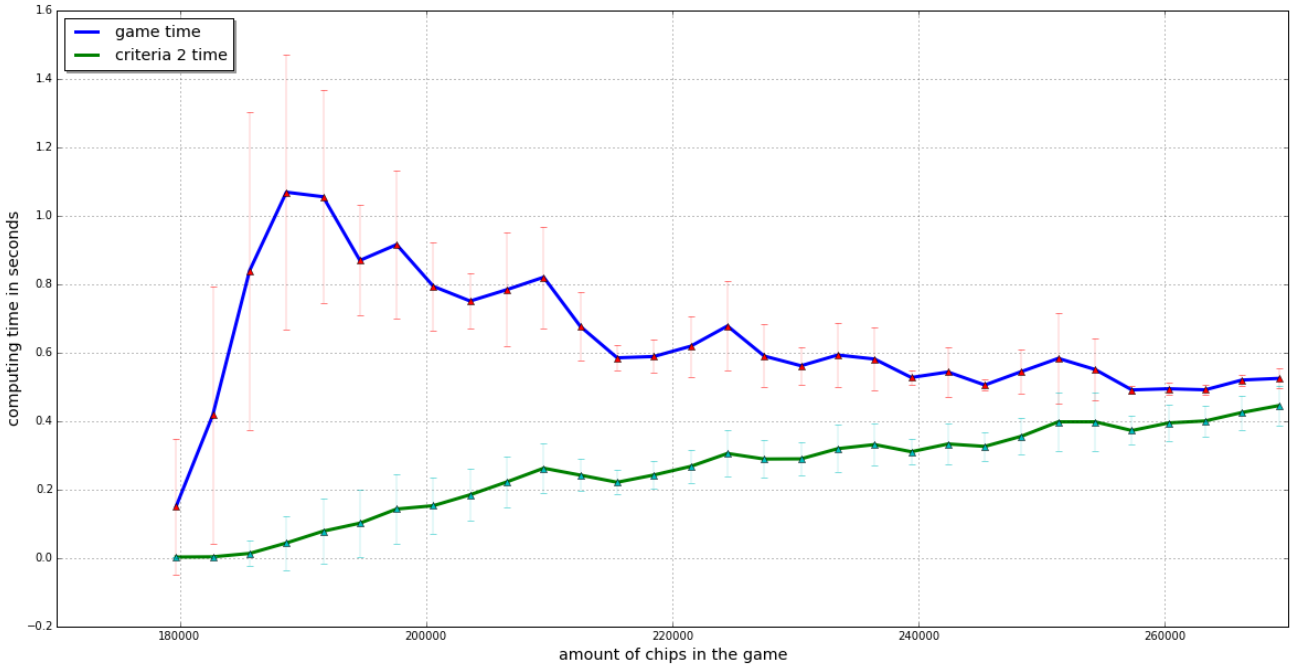



Figure 3.3: Time for playing game and checking criteria with $n = 600$

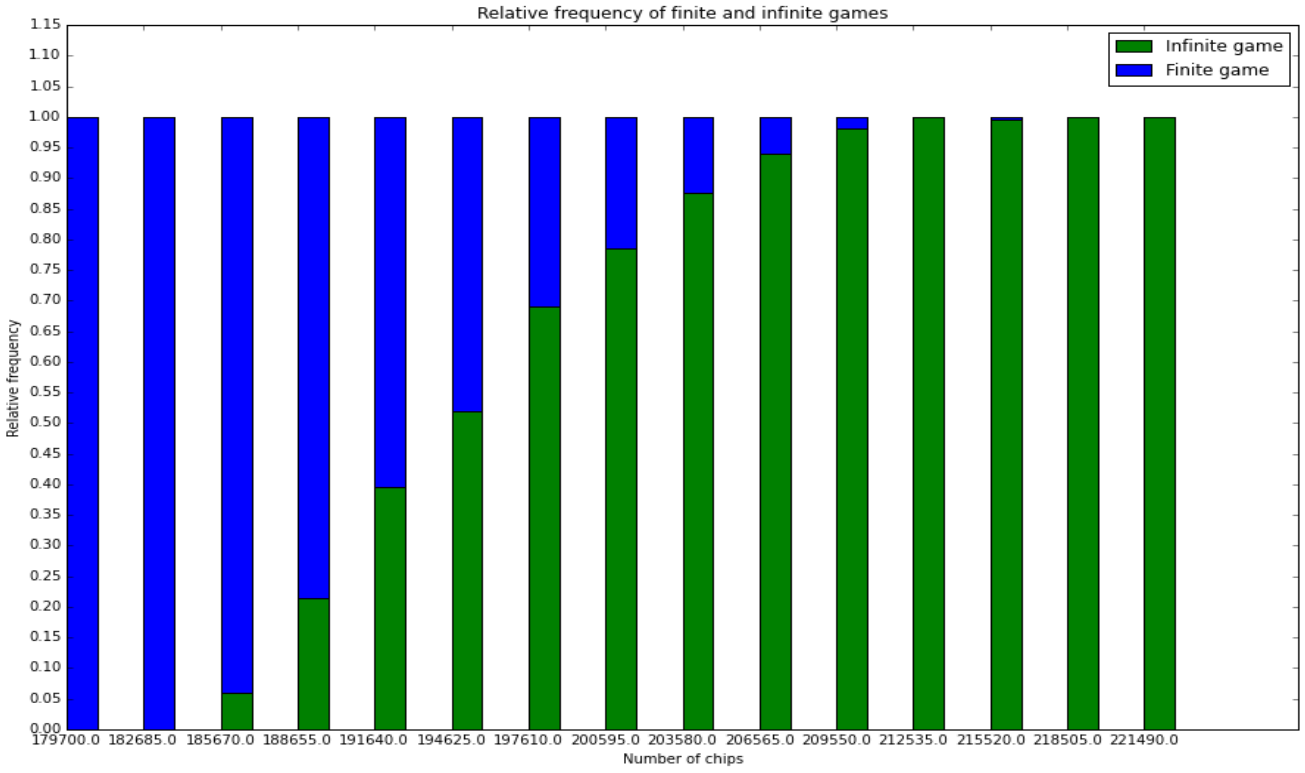



Figure 3.4: Bar chart for ratio of finite and infinite games for $n = 600$

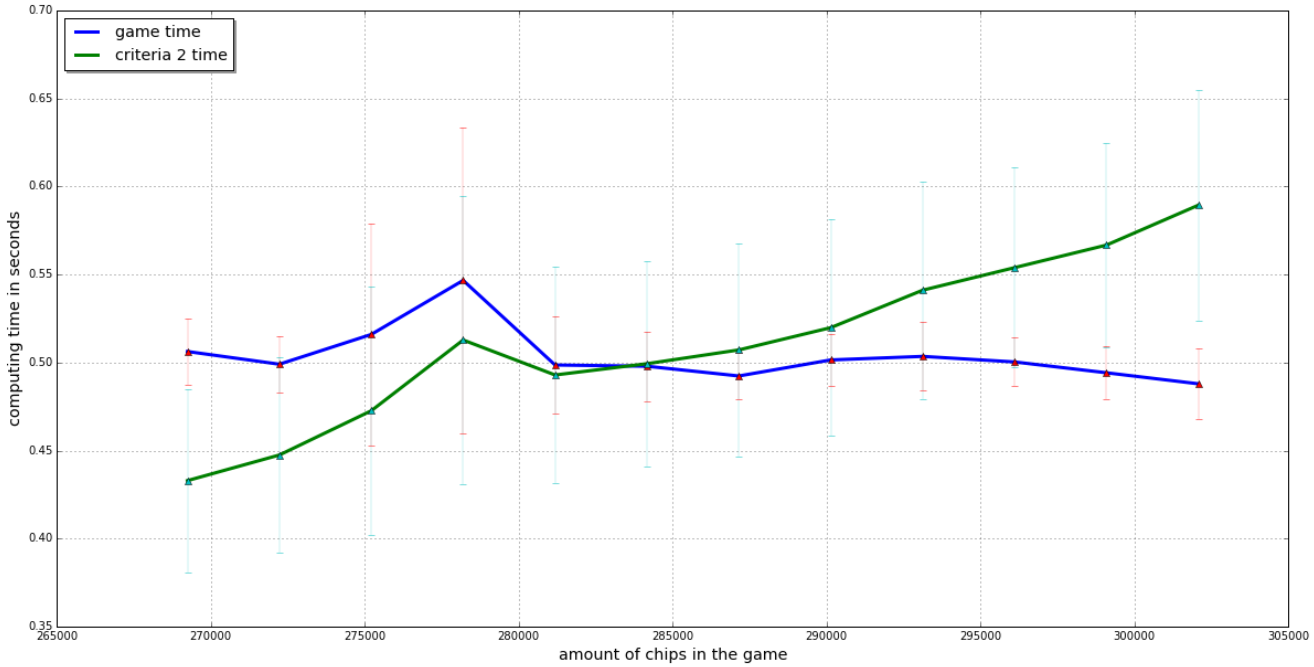



Figure 3.5: Time for playing game and checking criteria with $n = 600$

## 3.2 Chip-firing game on complete bipartite graph

**Complete bipartite graph** is a graph whose vertices can be separate in two disjoint sets, each node of the one set is connected to every node in the other set, but not inside its own set. The usual notation for it is $K_{nm}$

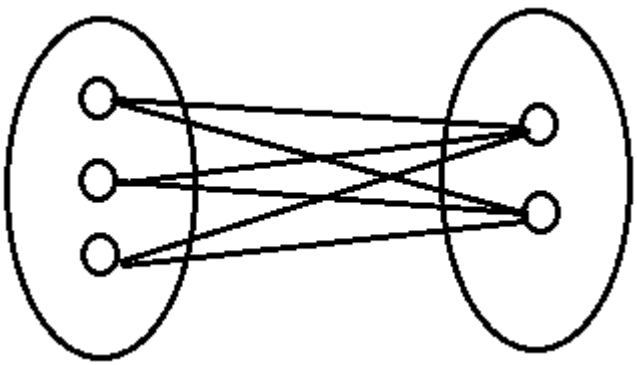


Figure 3.6: Complete bipartite graph $K_{m,n},\ m = 3, n = 2$

We consider the chip-firing game on this graph. The number of chips lays in the interval $m \cdot n \leq N \leq 2 \cdot mn - (m + n)$.

**Lemma 3.5** *Let $G$ be a complete bipartite graph with two sets of nodes $A$ and $B$. Let the size of $A$ be $m$ and the size of $B$ be $n$. If either the set $A$ has fired $m$ and more times or the set $B$ has fired $n$ and more times then the game is infinite.*

**Proof.** Let assume that nodes in the set A has fired $m$ times. This means that each node has received at least $m$ chips. Then each node from the set B can fire. They distribute $n$ chips to each node in the set A. Then all vertices in the set A are overloaded and can be fired. By the Lemma 2.5 the game is infinite. □

**Theorem 3.6** *Let $G$ be a complete bipartite graph with two node sets $A$ and $B$ of size $m$ and $n$ respectively. Let $\alpha$ be an initial configuration of chips on the vertices of the graph $G$. Then the game with this configuration is finite iff we can find such $i,\ j \in \mathbb{N} \cup \{0\}$ that after adding $i$ $(i < n)$ chips to each node of the set $A$ and $j$, $(j < m)$ chips to each node of the set $B$ we get exactly $j$ firings in the set $A$ and exactly $i$ firings in the set $B$.*

**Proof.** $\Rightarrow$ Let assume there exist such $i, j$ that after adding $i$ chips to each node in the set A and $j$ chips to each node of the set B, we have exactly $j$ firings from the set A and exactly $i$ firings from the set B.

If $i \geq m$ or $j \geq n$ then we have $m$ or more firings from the set B and $n$ or more firings from the set A. And by the Lemma 3.5, the game is infinite.

Let us assume the case when $i < m$ and $j < n$. In the set B we have less than $m$ firings and in the set A we have less than $n$ firings. So, by the Lemma 3.5 the game is finite.

$\Leftarrow$ Let us assume that the chip-firing game with a given initial configuration is finite. Then by the Lemma 3.5 there are a node in A and a node in B which didn't fire. This means that the set A has to have no more than $n-1$ firings and the set B has to have no more than $m-1$ firings. Let assume that during the game the set A has got $i$ chips for each of its nodes and the set B has fired $l$ times. Suppose that $i \neq l$. It follows that either some amount of chips has been lost during the game or it has appeared from outside the graph. As we play the game where the number of chips in the graph is fixed and it doesn't change during the game. So, from here we derive that $i = l$ for the set A. By the same logic we prove that $j = k$ for the set B, where $k$ is how many firings was from the set A.

This concludes the proof of the theorem. □

**Lemma 3.7** *Let $\alpha$ be an initial configuration of $K_{mn}$ with $N$ chips in the interval $mn \leq N \leq 2 \cdot mn - (m+n)$. Let denote the one vertex set as $A$ and the other vertex set as $B$ with the sizes $m$ and $n$ respectively. After firing some sequence of nodes starting at configuration $\alpha$, we can reach a configuration $\beta$ such that $c_i \leq 2 \cdot n - 1$ for the nodes in the set $A$ and $c_i \leq 2 \cdot m - 1$ for the nodes in the set $B$ of the graph $K_{nm}$.*

**Proof.** Consider a configuration $\alpha'$ on $K_{nm}$. Here $c_i = k$ for the vertex $v_i$ in the set A if in the configuration $\alpha$ we have $c_i = 2 \cdot k$ or $c_i = 2 \cdot k + 1,\ k \in \mathbf{N}$. Consider the same for the vertex $v_j$ in the set B. The sum of chips is less than $mn$. By the Theorem 2.6, the game is finite. Let W be the fired vertex sequence with which the game terminates having $\alpha$ as initial configuration. We double the number of firings in the W sequence and denote it as U. Then, after firing the last vertex in U, we obtain a desired configuration $\beta$. □

**Theorem 3.8** *Let $G$ be a complete bipartite graph with two sets of nodes $A$ and $B$ of the sizes $m$ and $n$ respectively. Let the configuration $\alpha$ be such that every node of the set $A$ is less than $2 \cdot n - 1$ and every node of the set $B$ is less than $2 \cdot m - 1$. The game with an initial configuration $\alpha$ is finite if and only if there exist numbers $i, j, k, l \in \mathbf{N} \cup \{0\}$ which satisfy the following inequalities:*

$$\begin{aligned} A[l+1] + i + j &< 2 \cdot n \\ A[k+1] + i + j &< n \\ B[j+1] + l + k &< 2 \cdot m \\ B[i+1] + l + k &< m \end{aligned} \tag{3.1}$$

**Proof.** $\Leftarrow$ Let us assume that the game is finite. We define two sets $A_1$ and $A_2$ as follows: the set $A_1$ consists of nodes in the vertex set A that fire at least once and the set $A_2$ consists of those nodes which fire twice. Equivalently, we define two sets $B_1$ and $B_2$ for the vertex set B. We denote vertices which belong to the set A as $v_1, v_2, ..., v_m$ and the vertices of the set B as $v_{m+1}, v_{m+2}, ...v_n$. We consider that the chips on the vertices of the set A and B are ordered in the descending order. Assume that $A_1 = \{v_1, ..., v_k\}$ and $A_2 = \{v_1, ..., v_l\}$. For the set B we assume that $B_1 = \{v_{m+1}, ...v_{m+i}\}$ and $B_2 = \{v_{m+1}, ...v_{m+j}$. It is obvious that $l \leq k$ and $j \leq i$. As the game is finite then by the Lemma 3.5 $k < m$ and $i < n$. The termination of the game means that the vertex $v_{k+1}$ does not fire at all and the vertex $v_{l+1}$ can not fire twice. The same for the nodes of the set B, $v_{i+1}$ and $v_{j+1}$. We can represent these reasoning by the inequalities (3.1). $\Rightarrow$ Now, let assume that we have minimal integers $i, j, k, l$ such that $j \leq i \leq n$ and $l \leq k \leq m$ and the inequalities 3.1 are true. Since $A[k+1] + i + j < n$ then $A[m+1] + i + j < n$. By the above consideration and the bound condition of number of chips on the nodes, we have that $A[m] \leq A[1] \leq 2 \cdot n - 1 + A[m]$ or equivalently $0 \leq A[1] - A[m] \leq 2 \cdot n - 1$. Substituting $-A[m]$ with the not greater value $i + j - n$, we get $A[1] + i + j - n \leq 2 \cdot n - 1$. Putting it differently $A[1] + i + j - 2 \cdot n \leq n - 1$. This means that the node $v_1$ is not firing at this moment. Also nodes $v_k$ and $v_l$ are not firing at this moment. By the same logic it can be shown that $v_{m+1}$ is not firing at this moment.

Hence, no vertex can be fired and the game terminates. □

We will refer to the Theorem 3.6 as the first criteria of finiteness and to the Theorem 3.8 as the second criteria.

We made simulations and checked criteria for the bipartite graph with $m = 10$ and $n = 20$. We put on the graph an amount of chips which belongs to the interval $m \cdot n \leq N \leq 2 \cdot mn - (m+n)$. We checked 20 different values of the number of chips starting from the beginning of the interval and finishing in the middle of interval. We didn't include here the second half of the interval. The reason of this is that probability of appearance of finite game is too small. That is why using 200 trials (we tried also more trials, 1000 and greater, but there is no much difference in it) for each value of the number of chips couldn't show the finite case in the second half of the interval. The Figure 3.7 shows that with the grows of the number of chips the running times of all three methods almost equal. In these trials the most steady and quick was the first criteria. If the number of chips is increased then there is no big difference between these three methods for the graph of given above size.

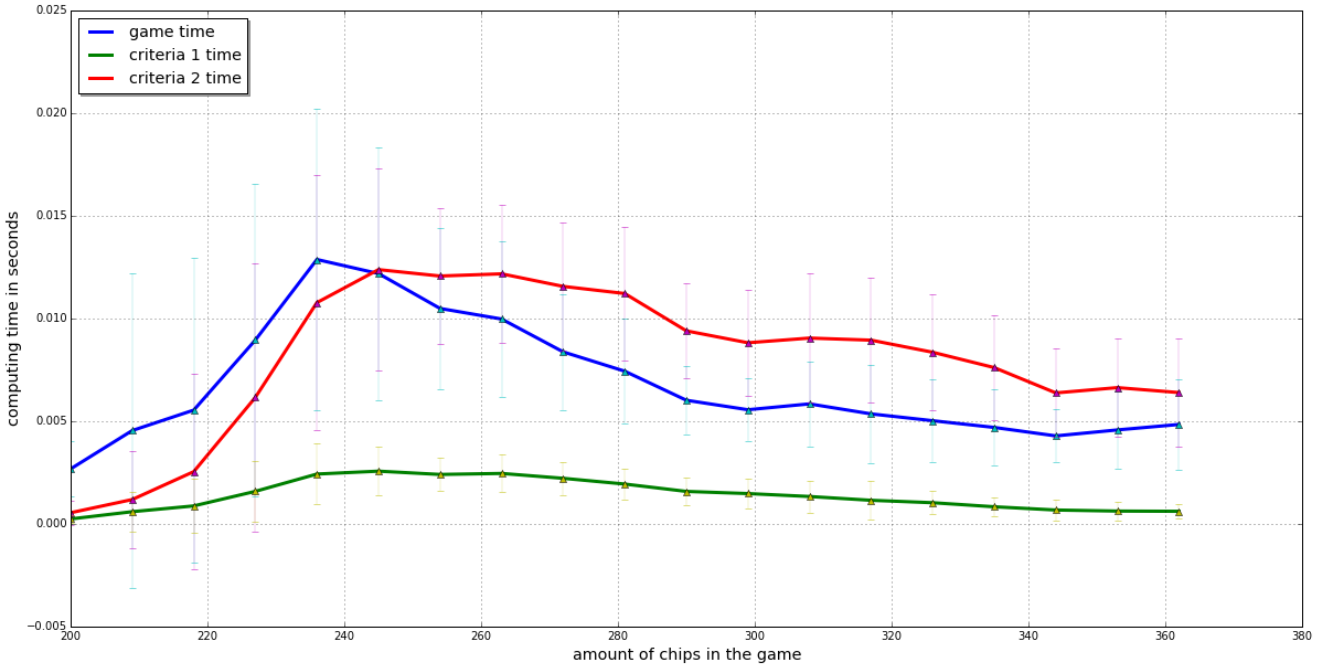



Figure 3.7: Computational time for playing the game and checking two criteria $m = 10,\ n = 20$

When we increase the number of nodes in a bipartite graph (we simulated for the $m = 40$ and $n = 60$, see Figure 3.9), the first criteria and playing a game have almost the same time tender for making a decision if the game finite or not. We see at the Figure 3.9 that computing time of the second criteria is substantially different from other methods. The graph of it grows to some point and then it slowly decreases.

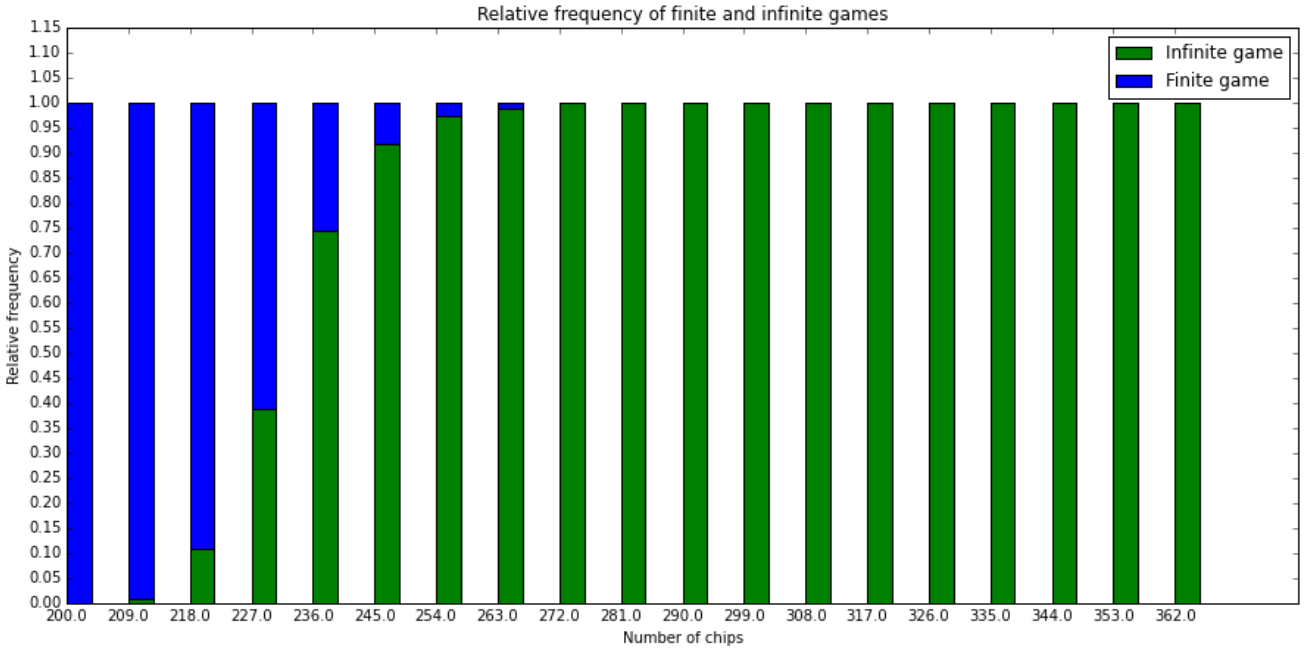



Figure 3.8: Bar chart with ratio of number of finite and infinite games $m = 10,\ n = 20$

This method works more slowly compare to others as it involves four loops in its algorithm (see def second_ criteria in Appendix B). Firstly, we need actually to play the game till we obtain a configuration in which all nodes have bounded number of chips on each of them. More precisely, for the set A we should have no more than $2 \cdot n - 1$ chips and for the set B we should have no more than $2 \cdot m - 1$ chips. And then we go through four loops. One loop has the length no more than the half of the size of the set A and another loop - no more than the half of the set B. Two others loops can have length up to the size of a set of a nodes. So, the running time of the algorithm is roughly $O(m^2 \cdot n^2)$.

We observe a growth of the line till some moment for the second criteria and than the line start to decrease. The reason of it is the following. With the growth of the number of chips in the game the number of nodes which can fire at least once at the beginning is increased. So the loop goes through more narrow interval of integers. This explains only downward characteristic of the plot. Why do we have an increasing part at the graph? The answer for it is that we encounter more infinite games while the range of integers which the loops go through is still wide. Put it differently, we do not have many nodes which fire at least once from the beginning.

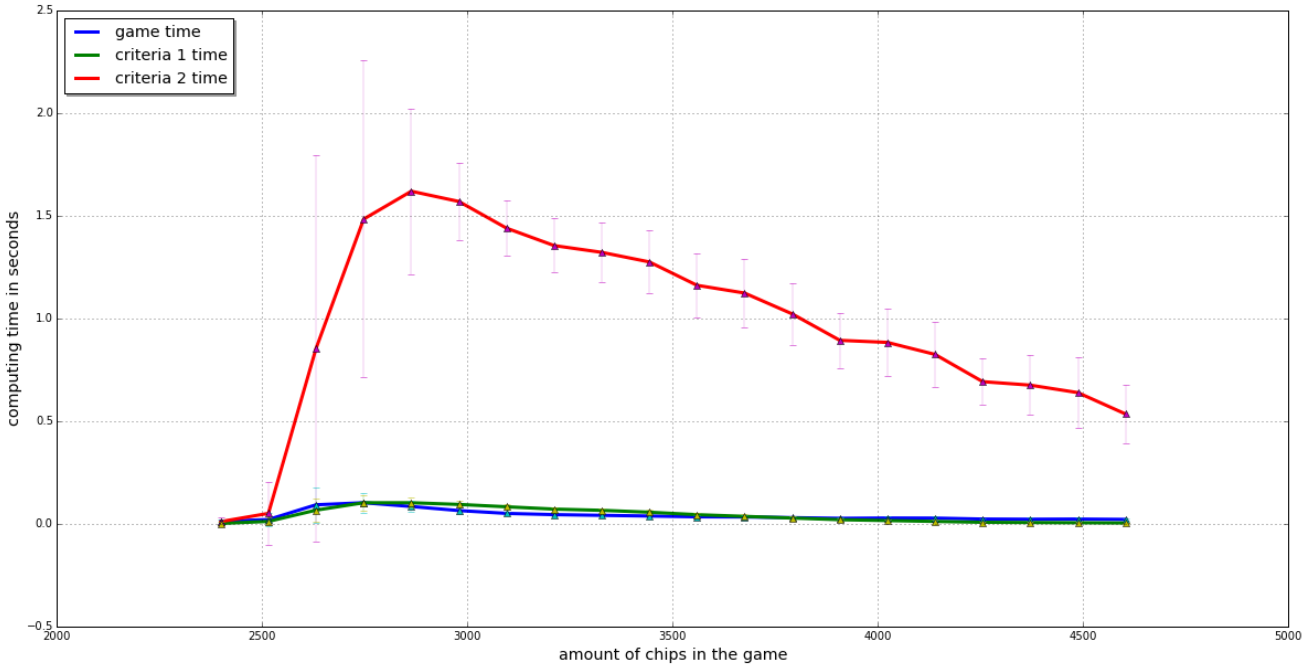



Figure 3.9: Computational time for playing the game and checking two criteria $m = 40,\ n = 60$

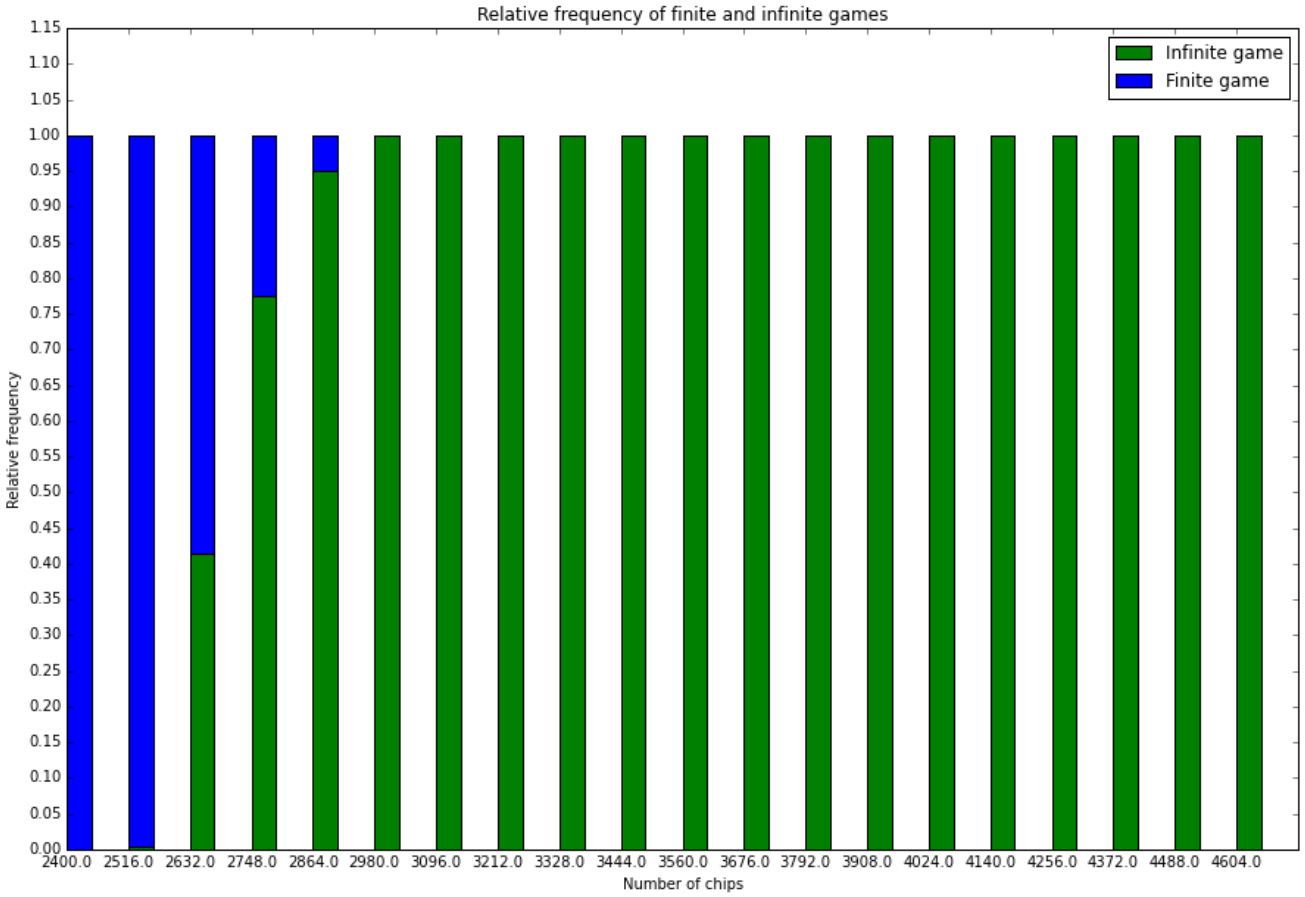





Figure 3.10: Bar chart with ratio of number of finite and infinite games $m = 40,\ n = 60$

Further investigation showed that with a growth of the size of the vertex set the playing of the game outperform the first criteria.

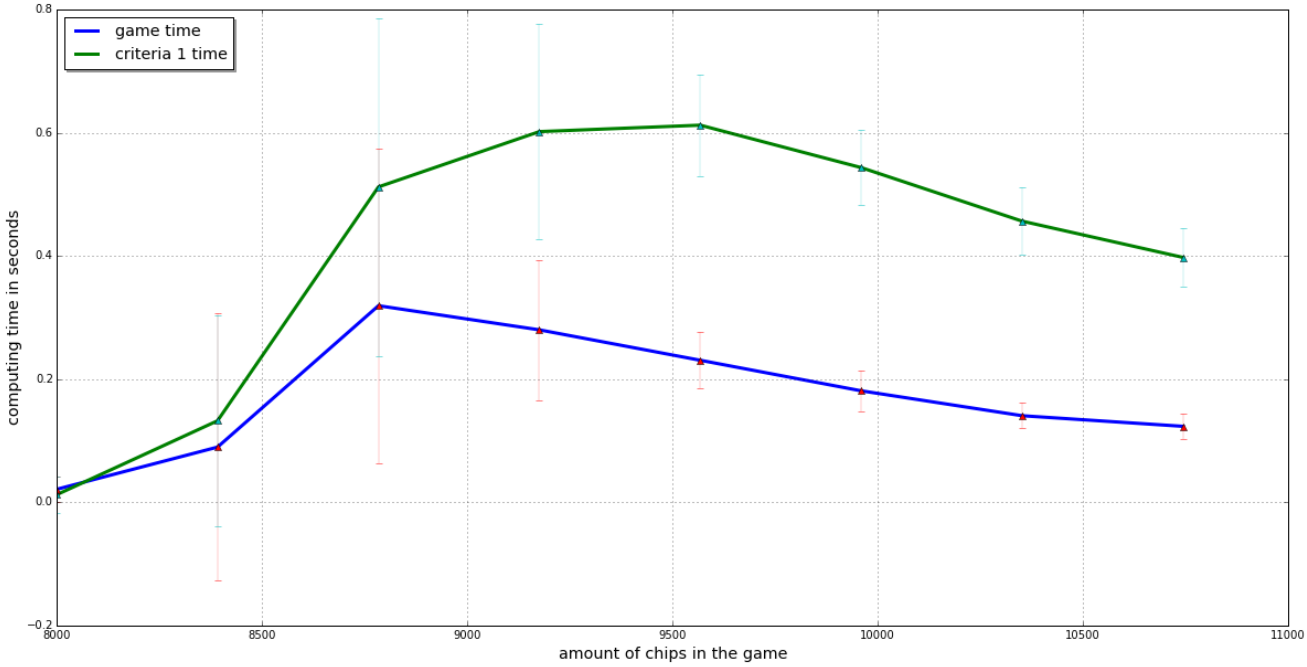



Figure 3.11: Computational time for playing the game and checking first criteria $m = 80,\ n = 100$

# Conclusion

In this thesis we studied the chip-firing game. We have shown two approaches of defining this game. The main focus was made on playing this game on complete and complete bipartite graphs. We can summarize key ideas as the following:

1. We showed that for the complete graph in some cases it is computationally more efficient to use criterion 3.4 and in some cases it is better just to play the game;
2. We proposed two criteria for checking the finiteness of the game on the complete bipartite graph with a given configuration;
3. We gave necessary condition for the game to be finite on complete bipartite graph;
4. We came to conclusion that with a the size of vertex set greater than around 150 in the complete bipartite graph to play the game shows the best time performance;
5. We created two program codes for complete graph and complete bipartite graph which can help not only get the answer if the game finite or not, but also they help to visualise the process if the number of nodes is small.

# Bibliography


[1] P. Bak, C. Tang, and K. Wiesenfeld. Self-organized criticality: An explanation of 1/f noise. *Phys. Rev. Lett.*, 59:381, 1987.

[2] Vladyslav A. Golyk. Self-organized criticality.

[3] Raphaël Cerf and Matthias Gorny. A curie–weiss model of self-organized criticality. *Ann. Probab*, 44(1):444–478, 2016.

[4] C. Correia Ramos, Nuno Martins, Ricardo Severino and J. Sousa Ramos. A curie–weiss model of self-organized criticality. 2006.

[5] D. Dhar. Self-organized critical state of sandpile automaton models. *Phys. Rev. Lett.*, 1990.

[6] D. Dhar. Theoretical studies of self-organized criticality. *Physica A: Statistical Mechanics and its Applications*, 2006.

[7] R. Anderson, L. Lovasz, P. Shor, J. Spencer, E. Tardos, S. Winogra. Disks, balls, and walls: analysis of a combinatorial game, amer. mat. *Amer. Math. Monthly*, 96(6):481–493, 1989.

[8] A. Björner, L. Lovász, P. W. Shor. Chip-firing games on graphs. *EUROPEAN J. COMBIN*, 12:283–291, 1991.

[9] B. Korte and L. Lovász. Structural properties of greedoids. *Combinatorica 3*, pages 359 – 374, 1983.

[10] G. Tardos. Polynomial bound for a chip firing game on graphs. *SIAM J. Discrete Math.*, 1(3):397–398, 1988.

[11] Zhuang, W., Yang, W., Zhang, L. Properties of chip-firing games on complete graphs. *Bulletin of the Malaysian Mathematical Sciences Society*, 38(4):1463–1469, 2015.

# Appendix A

# Python code for chip-firing game on complete graph

```
import networkx as nx
import matplotlib.pyplot as plt
import random
import operator as op
from functools import reduce
import time
import sys


def nCr(n,r): #function for computing combination
    r = min(r, n-r)
    if r == 0: return 1
    numer = reduce(op.mul, range(n, n-r, -1))
    denom = reduce(op.mul, range(1, r+1))
    return numer//denom




def set_up_of_chips(N_chips,G,n):
#here we randomly assign number of chips to
#each vertex of the graph
        r = [random.random() for i in range(n)]
    s = sum(r)
```

```
    r = [i/s for i in r]
    r = [round(i*N_chips) for i in r]
    if sum(r)>N_chips:
        difference = sum(r) - N_chips
        for i in range(len(r)):
            if r[i]>difference:
                r[i]-=difference
                break
    elif sum(r)<N_chips:
                difference = N_chips - sum(r)
                r[-1]+=difference
    for i in range(n):
        G.node[i]['chips'] = r[i]
    return G
"""
Here we sort vertices by the number of chips in descending order.
We are separating vertices in the blocks.
The difference number of chips between two neighbouring nodes is
no more than 1 chip.
Difference in chips between the last vertex of one block
and the first vertex of another block is two or more chips.
We are forming a list with the number of chips on the first vertex
of each block and a list with number of vertices in each block.
"""

def partition(chips):
    sorted_config = sorted(chips.items(),
                                    key=lambda x: x[1], reverse=True)
    large_chips = [sorted_config[0][1]]
    number_of_vert = []
    a = 0
    for i in range(n-1):
        if sorted_config[i][1]-sorted_config[i+1][1]>=2:
            large_chips.append(sorted_config[i+1][1])
            b = i+1
            number_of_vert.append(b-a)
            a = i+1
    number_of_vert.append(n-a)
    return number_of_vert, large_chips, sorted_config
```

```
#Here we are trying to determine if game is finite by
#the criteria for complete graphs.

def check_inequalities(chips,n):

    while not all(x<=2*n-3 for x in chips.values()):
        for i in chips.items():
            if i[1]>2*n-3:
                chips[i[0]]-=n-1
                break
        for j in chips.items():
            if j[0]!=i[0]:
                chips[j[0]] +=1

    temp = partition(chips)
    number_of_vert = temp[0]
    large_chips_block = temp[1]

    for j in range(len(number_of_vert)):
        for i in range(j,len(number_of_vert)):
            a = sum(number_of_vert[:i])
            b = sum(number_of_vert[:j])
            if (large_chips_block[i] + a + b < n-1)
                and (large_chips_block[j] + a + b - n < n-1):
                return True,i,j,chips
    return False

"""
Here we are coloring nodes in the graph after firing or in the case
of the initial configuration. If the number of chips on the a node
is bigger than its degree than we color a node in blue.
We color a node with this condition and with maximal number of chips
in red. This node is going to fire at this moment. Other nodes we
color in green.
"""

def node_color_1(G, node):
```

```
    for i in G.nodes():

            if (G.node[i]['chips'] >= G.degree(i)) and (i!=node):
                G.node[i]['color'] = '#87cefa'
            elif (G.node[i]['chips'] >= G.degree(i)) and (i == node):
                G.node[i]['color'] = 'r'
            else:
                G.node[i]['color'] = 'g'

    return G

"""
Here we are coloring the neighbours of a firing node in yellow.
By this we emphasize nodes which receives a chip
after the firing a node.
"""

def node_color_2(G, node):

    for i in G.nodes():

            if i in G.neighbors(node):
                G.node[i]['color'] = 'y'

    return G

"""
Here we are playing chip-firing game as following, we are firing
a given node. We have a list which contains information of
the number of chips on the graph. After firing some node v, we
rewrite this list, adding 1 to a neighbours of a firing node and
subtracting n - 1 chips from v. We fire a given node at once
as much as it possible. Also we can picture this process in case
if we put True for an argument of parameter visual. We are drawing
a graph only when we have no more than 40 nodes.
"""

def chip_firing_visual(G, node, visual = False, *arg):
```

```
    condition = (visual == True) and (len(G) <= 40)
    node_color_1(G, node)
    if condition:
            posit=nx.shell_layout(G)
            plotting(G,posit)

    node_color_2(G, node)

    if condition:
            plotting(G,posit)

    times = int(G.node[node]['chips']//G.degree(node))
    for i in G.neighbors(node):
        G.node[i]['chips'] += times
    G.node[node]['chips']-=G.degree(node)*times

    node_color_1(G, node)
    if condition:
            plotting(G,posit)

    return G

#This function is responsible for drawing a graph.

def plotting(G,posit):
     node_labels = nx.get_node_attributes(G,'chips')
         nx.draw(G, pos = posit, with_labels = True, \
     labels = node_labels, \
     node_size=[(v + 1) * 200 for v in node_labels.values()],\
     node_color = [G.node[node]['color'] for node in G.nodes()])

     plt.pause(1.5)
     plt.clf()

     return

#This function searches for overloaded nodes and
makes list of number of chips in the descending order.
```

```
def maximum(chips):
      overload_nodes = []
      for i in chips.items():
          if G.degree(i[0])<=G.node[i[0]]["chips"]:
                overload_nodes.append(i)
      max_ = sorted(overload_nodes, key = lambda x:\
      x[1],reverse=True)
      return max_


"""
This function take a graph with initial chips arrangement
on nodes. We are finding nodes which are ready to fire
from the beginning of the game and write them in the list
overload_nodes. When we exhaust this list we are creating
a new one if there still nodes which can fire and if we fire
no more than n - 1 times.
"""
def game(G, visual = False):

     count = 0
     chips=nx.get_node_attributes(G,'chips')
     overload_nodes = maximum(chips)

     if overload_nodes:
          node = overload_nodes[0][0]
          count += int(overload_nodes[0][1]//(n-1))
          G = chip_firing_visual(G,node,visual)
          overload_nodes = overload_nodes[1:]

          if not overload_nodes:
                overload_nodes = maximum(chips)


          while overload_nodes:

               node = overload_nodes[0][0]
               count += int(overload_nodes[0][1]//(n-1))
```

```
            G = chip_firing_visual(G, node, visual)

            if count >= n:
                return ("infinite game")

            if overload_nodes[1:]:
                overload_nodes = overload_nodes[1:]
            else:
                chips=nx.get_node_attributes(G,'chips')
                overload_nodes = maximum(chips)

        if not overload_nodes:
           return ("finite game")

    else:
        return ("nothing to fire")

def input_data(input_parameter):
    while True:
            if input_parameter == 'y':
                input_parameter = True
                break
            elif input_parameter == 'n':
                input_parameter = False
                break
            else:
                input_parameter = input("Try again...only \
                [y/n] or Enter to quit:")
                if not input_parameter:
                        sys.exit()
    return input_parameter

"""
This is the body of the program.
It takes an input the size of the graph as integer and
give the option to show or not the game visually.
We are consider only those quantities of amount of chips
where we can't say immediately if the game finite or not.
A program can manually or randomly assign number of chips
```

```
for the game. You can also choose either manually or randomly
assign amount of chips on each node.

"""
n = int(input("How many nodes in a complete graph? "))
visualisation = input("Do you need visualisation , [y/n]? ")
visualisation = input_data(visualisation)

automatically = input('Do you want automatically assign \
number of chips [y/n]? ')

automatically = input_data(automatically )

lower_bound = int(nCr(n,2)) #lower bound
upper_bound = 2*lower_bound - n #upper bound
print('lower_bound - upper_bound',(lower_bound, upper_bound))

if automatically:
    N_chips = random.randint(lower_bound, upper_bound)
    print("N_chips =", N_chips)
else:
    N_chips = int(input("Type the number of chips as positive \
        integer the above range = "))

G = nx.complete_graph(n)
m = nx.number_of_edges(G)

configuration = input('Do you want automatically arrange \
chips on each node [y/n]? ')
configuration = input_data(configuration)

if configuration:
        set_up_of_chips(N_chips,G,n)
else:
        for i in G.nodes():
                G.node[i]["chips"] = int(input("number of chips \
                on the node {} : ".format(i)))
```

```
chips=nx.get_node_attributes(G,'chips')
check_inequalities(chips,n)
print()
print(game(G, visual = visualisation))
```

# Appendix B

# Python code for chip-firing game on complete bipartite graph

```
"""
Here we use the following function from the first Appendix:
set_up_of_chips, node_color_1, node_color_2, chip_firing_visual,
plotting, maximum.
"""

import networkx as nx
import matplotlib.pyplot as plt
import random
import sys

"""
This function cleans the file if such exists and create a new one
if not.
"""

def file_cleaning(file_name):
    file = open("{}.txt".format(file_name),'w')
    file.close()
    return

"""
This function check the first criteria of finiteness for the
complete bipartite graph:
"""
```

```
def first_criteria(G):
"""
We take attributes of nodes. This is a dictionary. The keys of it
are the number of a node and the values are the numbers of chips
on these nodes.
"""
        chips = nx.get_node_attributes(G,'chips')

"""
We form the list of values. Then we divide it into two sublists.
The first one consists of the nodes of a set A and the second one
consists of the nodes of a set B. Then we sort it in the descending
order.
"""

    set_A = sorted(list(chips.values())[:m], reverse = True)
    set_B = sorted(list(chips.values())[m:], reverse = True)

"""
Here we find a one node from each set with the minimal number of
chips on it.
"""
    min_A = set_A[-1]
    min_B = set_B[-1]

"""
Here we are searching how many nodes of a set A have the number of
chips greater than n.
"""

    for x in range(m):
        if set_A[x] < n:
            break

"""
Here we are searching how many nodes of a set B have the number of
chips greater than m.
"""
```

```
    for y in range(n):
        if set_B[y] < m:
            break
    if x == 0:
        x = 1
    if y == 0:
        y = 1

"""
In these loops we are searching for the integers i,j. The number
of firings from the set B and the set A respectively. The number
of firings i from the set B can not be less than the number of
nodes ready to fire from the starting configuration. Also it can
not be greater than the difference between the size of the set B
and the minimal number of chips from the set A. By the same logic
we put the bounds for the j.
"""

    for i in range(y-1,n-min_A):
        for j in range(x-1,m-min_B):

"""
Here we add i chips on each node of the initial configuration
for the set A and j chips for the set B.
"""
            set_A_check = list(map(lambda x: x + i, set_A))
            set_B_check = list(map(lambda x: x + j, set_B))
            k = 0
            l = 0
"""
Here we are checking how many nodes ready to fire in the set A
after adding some number of chips.
"""
            for p in set_A_check:
                if p<n:
                    break
                else:
                    temp = p//n
                    k +=temp
```

```
#Here k is a number of firings from the set A.

            for q in set_B_check:
                if q<m:
                    break
                else:
                    temp = q//m
                    l +=temp

#Here l is a number of firings from the set .
            if (k == j) and (l == i):
                return True, (j,i)
    return False

"""
Here we are playing game firing firstly all overloaded
nodes from the one set and then all nodes from the other
set. We are doing this interchangeably.
"""
def game(G, visual = False):
    count_firing_A = 0
    count_firing_B = 0

    chips=nx.get_node_attributes(G,'chips')
    overload_nodes = maximum(chips)

    while overload_nodes:
        node = overload_nodes[0][0]
        G = chip_firing_visual(G, node, visual)
        if node in bottom_nodes:
            count_firing_A += int(overload_nodes[0][1]//n)
            overload_nodes = [t for t in overload_nodes[1:] \
            if t[0]<m]
        else:
            count_firing_B += int(overload_nodes[0][1]//m)
            overload_nodes = [t for t in overload_nodes[1:] \
            if t[0]>=m]
```

```
            if not overload_nodes:
                overload_nodes = maximum(chips)

            if (count_firing_A >= m) or (count_firing_B >= n):
                return ("infinite game")

    return ("finite game")
"""
In the second criteria we firstly arriving to the configuration
where the number of chips on each node in the set A is less
or equal than 2*n - 1 and in the set B is less or equal
than 2*m - 1.

Then we figure out the numbers a,b,c, d which represent how many
nodes are ready to fire at least one time from the set A and
the set B and how many nodes are ready to fire twice from
the set A and the set B respectively.
"""
def second_criteria(G):
    chips = nx.get_node_attributes(G,'chips')
    while not (all(x<=2*n-1 for x in list(chips.values())[:m]) \
                and all(x<=2*m-1 for x in list(chips.values())[m:])):
            for i in chips.items():
                if (i[1]>2*n-1) and (i[0]<m):
                    G.node[i[0]]['chips']-=G.degree(i[0])
                    break
                elif (i[1]>2*m-1) and (i[0]>=m):
                    G.node[i[0]]['chips']-=G.degree(i[0])
                    break
            for j in G.neighbors(i[0]):
                G.node[j]['chips']+= 1

            chips = nx.get_node_attributes(G,'chips')


    set_A = sorted(list(chips.items())[:m], key = lambda x: x[1], \
    reverse = True)
    set_B = sorted(list(chips.items())[m:], key = lambda x: x[1], \
    reverse = True)
```

```
    for a in range(m):
        if set_A[a][1] <n:
            break
    for c in range(m):
        if set_A[c][1]  < 2*n:
            break
    for b in range(n):
        if set_B[b][1]<m:
            break
    for d in range(n):
        if set_B[d][1]<2*m:
            break

#l counts the nodes which can fire twice in the set A

    for l in range(c,int(m/2)+1):

#as we can fire from the set B to set A no more than m-1 times

"""
k counts the nodes which fire at least once and this number
can not be less than l.
"""

        for k in range(max([a,l]),m-l):
            for j in range(d,int(n/2) + 1):

"""
as we can fire from the set A to set B no more than n-1 times.
Here we count nodes which fired 2 times
"""

                for i in range(max([b,j]),n-j):
                    cond_1 = set_A[l][1]+ i + j < 2*n
                    cond_2 = set_A[k][1]+ i + j < n
                    cond_3 = set_B[j][1]+ l + k  < 2*m
                    cond_4 = set_B[i][1]+ l + k < m
                    if cond_1 and cond_2 and cond_3 and cond_4:
```

```
                        return True, (k,l,i,j)
    return False

"""
This function assign for the given parameter True or False
depending on input. If input is not 'y' or 'n' then it gives
an opportunity to re-enter it again or just leave the program.
"""

def input_data(input_parameter):
    while True:
            if input_parameter == 'y':
                input_parameter = True
                break
            elif input_parameter == 'n':
                input_parameter = False
                break
            else:
                input_parameter = input("Try again ... only [y/n] \
                or Enter to quit:")
                if not input_parameter:
                        sys.exit()
    return input_parameter

"""
We assign number of nodes in the bottom and upper sets
of a graph. Also we are asked if we need the visualisation
of the game. We can automatically assign number of chips or
do it manually. We can automatically set up a configuration
or also do it manually.
"""

m = int(input("How many nodes in a bottom set of a \
complete bipartite graph? "))
# number of nodes in the bottom set
n = int(input("How many nodes in a upper set of a \
complete bipartite graph? "))
# number of nodes in the upper set
```

```
G = nx.complete_bipartite_graph(m, n)

bottom_nodes, top_nodes = nx.bipartite.sets(G)
number_of_edges = m*n

visualisation = input("Do you need visualisation , [y/n]? ")
visualisation = input_data(visualisation)

automatically = input('Do you want automatically assign \
number of chips [y/n]? ')

automatically = input_data(automatically )

"""
number of chips for game be possibly infinite or finite
the interval
"""

lower_bound =  number_of_edges
upper_bound = 2*lower_bound - (m+n)
print(lower_bound,upper_bound)

if automatically:
    N_chips = random.randint(lower_bound, upper_bound)
    print("N_chips =", N_chips)
else:
    N_chips = int(input("Type the number of chips as \
    positive integer in the above range ="))

configuration = input('Do you want automatically \
arrange chips on each node [y/n]? ')
configuration = input_data(configuration)

if configuration:
        set_up_of_chips(N_chips,G,n+m)
else:
        for i in G.nodes():
        if i[0]<m:
            G.node[i]["chips"] = int(input("number of chips \
```

```
            on the node {}, set A: ".format(i)))
        else:
            G.node[i]["chips"] = int(input("number of chips \
            on the node {}, set B: ".format(i)))



lower_bound =  number_of_edges
upper_bound = 2*lower_bound - (m+n)
print((lower_bound,upper_bound))


#Here we check two criteria and the game on finiteness.
first_criteria(list(chips.values()))
second_criteria(G)
print(game(G, visual = visualisation))
```