The "Number Hides Game" on a Tree

B.J.A. Swinkels
4948297

Faculty of Electrical Engineering, Mathematics & Computer Science
Delft University of Technology
Netherlands
27/06/2021

Contents

1. Introduction **3**
- 1.1. How to play the "Number Hides Game" . . . 3
- 1.2. Content overview . . . 3

2. Game Theory **4**
- 2.1. Relevant terminology . . . 4
- 2.2. Linear optimization . . . 6
  - 2.2.1. Max-Min problem . . . 6
  - 2.2.2. Min-Max problem . . . 7
  - 2.2.3. Strong duality . . . 7
  - 2.2.4. Implementation . . . 8

3. The Original Number Hides Game **9**
- 3.1. Exact solution . . . 9
  - 3.1.1. Optimal strategy for player I . . . 12
  - 3.1.2. Optimal strategy for player II . . . 14
- 3.2. Reduction if m $<$ n . . . 19
- 3.3. Python implementation . . . 20

4. The Number Hides Game on a tree **21**
- 4.1. Numerical Approach . . . 21
- 4.2. The NHG on a spider . . . 25
- 4.3. Exact solution when m $\mid$ b+1 and n $<$ m . . . 28
  - 4.3.1. Optimal strategy for player I . . . 28
  - 4.3.2. Optimal strategy for player II . . . 29
- 4.4. Solution for large branchsizes . . . 29
- 4.5. Solution for small branchsizes . . . 32

5. Final word **35**

References **36**

Appendices **37**

A. Python Code **37**
- A.1. Linear optimization . . . 37
- A.2. Ferguson . . . 38
- A.3. Unittests for linear optimization and ferguson (+manual test) . . . 39
- A.4. Visualization of the NHG . . . 45
- A.5. The NHG on a Tree . . . 47

1. Introduction

In this thesis we will be discussing The "Number Hides Game", which we will regularly abbreviate to the NHG. Let us begin by writing down the rules of this game.

1.1. How to play the "Number Hides Game"

This game has two participants, player I (the 'seeker') and player II (the 'hider'). The board consists of a row of p consecutive coins, which we will label with the integers $\{1, 2, \ldots, p\}$. Player I and player II **simultaneously** pick m and n consecutive coins respectively. Player II pays the number of coins that lie in the overlap of both choices to player I. From now on, we refer to this transaction as the *payoff* of the game. The goal of player I is to maximize the payoff and the goal of player II is to minimize the payoff. With the notation $G(m, n, p)$ we denote the NHG with the parameters m, n and p. An example of a NHG can be found in Figure 1.

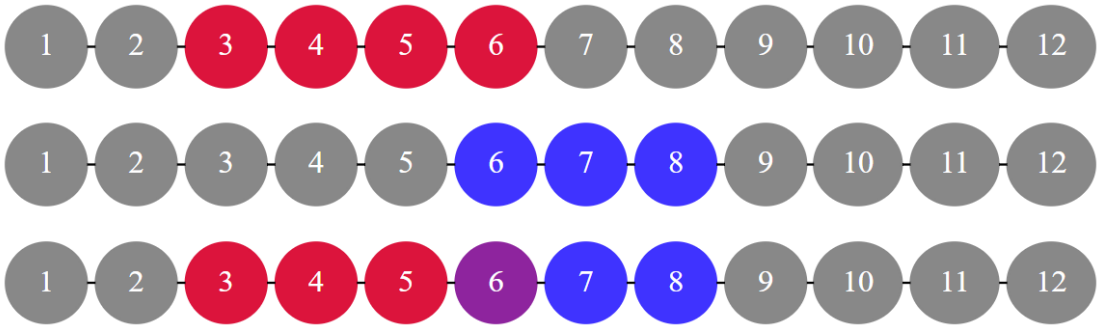

Figure 1: Player I (red) chooses m consecutive coins. Player II (blue) chooses n consecutive coins. The number of elements in the overlap denotes the payoff. In this case, the payoff is 1. The parameters in this example are $m, n, p = 4, 3, 12$.

This game was first described by W. H. Ruckle (1983) in his book *geometric games and their applications*, [1]. This book contains many geometric games and unsolved problems. One of these unsolved games was the NHG. This game was solved 6 years later by Baston, V. J., Bostock F. A. and Ferguson T. S. (1989), [2].

1.2. Content overview

In Section 2, we will review the required theory to mathematically solve games. Afterwards, we will be discussing the findings of Baston, Bostock and Ferguson on the NHG in Section 3. We will look at the value of the NHG, as well as the optimal strategies for both players. Finally, we will be expanding the NHG by increasing the complexity of the board layout in Section 4.

2. Game Theory

2.1. Relevant terminology

In this section we will translate a two player game with simultaneous decision making into mathematical terms. In such games, both players have a certain amount of moves to pick from. The *strategy set* of a player is a collection of all possible moves for that player. In the NHG, player I and player II respectively have $p - m + 1$ and $p - n + 1$ possible moves to make. Therefore, we can denote their strategy sets by $S_1 = \{0, 1, \ldots, p - m\}$ and $S_2 = \{0, 1, \ldots, p - n\}$, like in Figure 2.

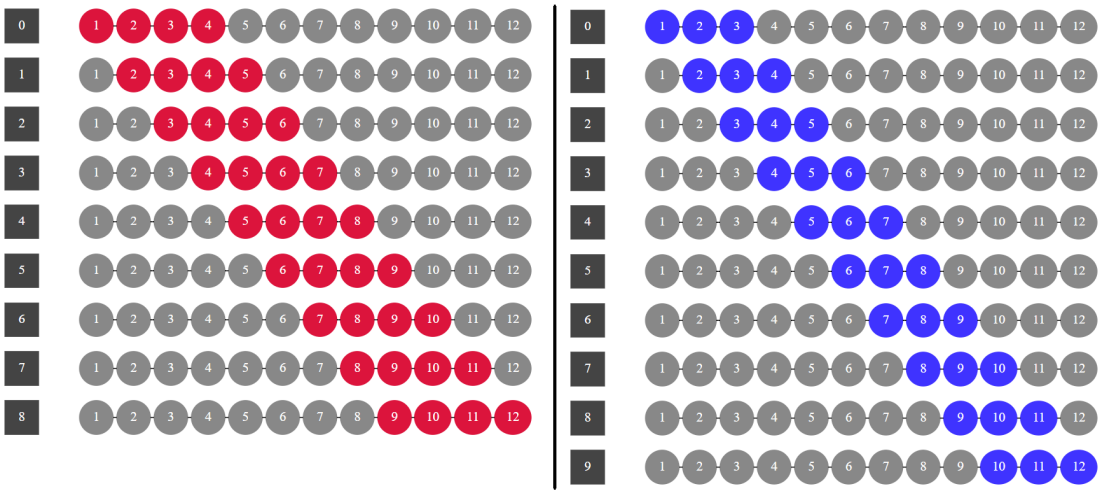

Figure 2: Strategy sets for Player I ($p - m + 1 = 12 - 4 + 1 = 9$ strategies) and Player II ($p - n + 1 = 12 - 3 + 1 = 10$ strategies) in the NHG $G(m, n, p) = G(4, 3, 12)$. The labels in each row indicate the move number.

A *payoff matrix* is a $|S_1| \times |S_2|$ matrix in which the rows represent the strategies of player I and the columns represent the strategies of player II. The element at position (i, j) represents the payoff when player I and II utilize strategies i and j respectively. In the example below, we find the payoff matrix that belongs to the strategy sets from $G(m, n, p) = G(4, 3, 12)$ like depicted in Figure 2.

$$A = \begin{pmatrix} 3 & 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 3 & 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 3 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 3 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 3 & 3 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 3 & 3 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 3 & 3 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 3 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 3 \end{pmatrix} \tag{1}$$

Before we dive into the details of strategies, please recall that the goal of player I is to maximize the payoff and the goal of player II is to minimize the payoff. This is because player II has to pay the amount of coins in the overlap to player I. This is what we defined to be the *payoff* in Section 1.1.

To play a game, one needs to utilize a strategy. Let us consider $G(4, 3, 12)$ again and let's say player I always chooses move $0 \in S_1$. This move corresponds to the red move labeled 0 in Figure 2 and is represented by the first row in the payoff matrix A from (1). This move has a clear bias towards the coins on the left side of the board. When using this move, player I hopes player II will also choose a move with coins on the left side of the board to maximize the payoff. Moves $0, 1 \in S_2$ are particularly beneficial for player I, since this will lead into the maximum overlap size of 3. (These two moves correspond to the 1st and 2nd column of A) However, if player II instead chooses a strategy from $\{4, 5, \ldots, p-n+1\}$, the payoff will be 0. So we can conclude that choosing a fixed move is vulnerable against some specific choices of the opponent.

The strategy we described above is called a *pure strategy* because it involves the utilization of a single move. The corresponding stochastic vector is $y = (1, 0, 0, 0, 0, 0, 0, 0, 0)$. This vector tells us player I chooses his first strategy with probability 1. We have seen that this strategy is vulnerable against strategies with coins of the right side of the board of player 2. We can optimize this strategy by spreading our moves to cover up for our vulnerabilities. Let's say we update the stochastic vector to $y = (\frac{1}{3}, 0, 0, \frac{1}{3}, 0, 0, \frac{1}{3}, 0, 0)$. This means we choose either strategy 0, 5 or 8. (see the red moves in Figure 2) Before we play the game we roll a dice to determine which move will be utilized during the game. This means that after repeatedly playing the game, we have chosen move 0, 5 and 8 an approximately equal amount of times.

In general, a *stochastic vector* $y = (y_0, y_1, \ldots, y_{p-m})^T$ is such that $y_i \geq 0$ for each i and $\sum_{i=0}^{p-m} y_i = 1$. Element y_i represents the probability player I will utilize move i. Strategies that involve the random selection of moves according to weights in a stochastic vector are called *mixed strategies*. Similarly, player II can utilize a mixed strategy that corresponds to a stochastic vector $x = (x_0, x_1, \ldots, x_{p-n})^T$, in which each element x_j represents the probability player II will utilize move j.

Let y, x be stochastic vectors like described above. The payoff a_{ij} is the payoff when players I and II use moves i, j respectively. According to the stochastic vectors, this specific scenario happens with probability $y_i x_j$. Thus, we can calculate the *expected payoff* with the matrix multiplication $y^T Ax$.

Finally, an *optimal strategy* for player I is a stochastic vector y such that $\min\{y^T Ax : x \text{ stochastic vector for player II}\}$ is maximum. And an *optimal strategy* for player II is a stochastic vector x such that $\max\{y^T Ax : x \text{ stochastic vector for player I}\}$ is minimum.

In other words, an optimal strategy is such that it beats the worst case scenario.

2.2. Linear optimization

The *value* of a game is defined by $v = y^T Ax$ whenever y and x are optimal strategies. (assuming such optimal strategies exists) In this section we will see that this value is well-defined by considering the problem from two different perspectives. First, we will look at the case where player I maximizes the value. Then, we will look at the case where player II minimizes the value. For the sake of short notation, we say that A is an m x n matrix in this subsection.[1] Furthermore, y is a m x 1 stochastic vector and x is a n x 1 stochastic vector.

2.2.1. Max-Min problem

First, we will look at the perspective of player I. Player I wishes to maximize the value. This can be done by adjusting the stochastic vector y such that the value $y^T Ax$ is maximized in the worst case scenario. That is, we have to consider the possibility that player II chooses his stochastic vector x such that $y^T Ax$ is minimized. In other words, we have to solve the following equation:

$$v = \max_y \min_x y^T Ax \tag{2}$$

We can rewrite this problem into the following linear problem.

$$\begin{aligned} &\text{maximize} && v && \\ &\text{subject to} && v \leq (y^T A)_j && \forall j \in \{1, 2, \ldots, n\} \\ &&& y_i \geq 0 && \forall i \in \{1, 2, \ldots, m\} \\ &&& \sum_{i=1}^{m} y_i = 1 && \end{aligned}$$

This problem *maximizes* the value v, subject to constraints in three different categories:

- The inner-minimum constraints $v \leq (y^T A)_j$ account for the inner minimization problem of (3). The value v must be smaller than each element of $y^T A$, since player II can freely choose the vector x to minimize the value $y^T Ax$. With this constraint we make sure player I's mixed strategy attains the value v against the worst case scenario.
- The positivity constraints $y_j \geq 0$ makes sure probabilities in the stochastic vector are positive
- The sum constraint $\sum_{j=1}^{m} y_j$ makes sure the probabilities in the stochastic vector add up to 1.

[1] Note that this m and n do not equal the parameters m and n from a NHG. In that case, A would be a $(p - m + 1)$ x $(p - n + 1)$ matrix instead.

2.2.2. Min-Max problem

Similarly, we can look at the perspective of player II. Player II wishes to minimize the value. This can be done by adjusting the stochastic vector x such that the value $y^T Ax$ is minimized in the worst case scenario. That is, we have to consider the possibility that player I chooses his stochastic vector y such that $y^T Ax$ is maximized. In other words, we have to solve the following equation:

$$v' = \min_x \max_y y^T Ax \tag{3}$$

We can rewrite this problem into the following linear optimization problem.

$$\begin{aligned} \text{minimize} \quad & v' \\ \text{subject to} \quad & v' \geq (Ax)_i && \forall i \in \{1, 2, \ldots, m\} \\ & x_j \geq 0 && \forall j \in \{1, 2, \ldots, n\} \\ & \sum_{j=1}^{n} x_j = 1 \end{aligned}$$

This problem *minimizes* the value v', subject to constraints in three different categories:

- The inner-maximum constraints $v' \geq (Ax)_i$ account for the inner maximization problem of (2). The value v' must be greater than each element of Ax, since player I can freely choose the vector y to maximize the value $y^T Ax$. With this constraint we make sure player II's mixed strategy attains the value v' against the worst case scenario.
- The positivity constraints $x_j \geq 0$ ensures probabilities in the stochastic vector are positive
- The sum constraint $\sum_{j=1}^{m} x_j$ ensures the probabilities in the stochastic vector add up to 1.

2.2.3. Strong duality

Now will we show that the max-min (2) and min-max (3) problems are exactly primal/-dual optimization problems. First, we write down the primal max-min problem (2) in standard form[2]:

$$\begin{aligned} \text{maximize} \quad & \begin{bmatrix} 0_m \\ 1 \end{bmatrix} \cdot \begin{bmatrix} y \\ v \end{bmatrix} \\ \text{subject to} \quad & \begin{bmatrix} -A^T & 1_n \\ 1_m^T & 0 \end{bmatrix} \begin{bmatrix} y \\ v \end{bmatrix} \geq \begin{bmatrix} 0_n \\ 1 \end{bmatrix} \\ & y \geq 0 \\ & v \text{ free} \end{aligned} \tag{4}$$

[2] 1_m and 1_n denote column-vectors of size m and n of all ones. 0_m and 0_n denote column-vectors of size m and n of all zeroes.

Now we can consider the dual problem of (4) by transposing the constraint matrix $\begin{bmatrix} -A & 1_m \\ 1_n^T & 0 \end{bmatrix}$, flipping the inequality signs, changing the positions of $\begin{bmatrix} 0_m \\ 1 \end{bmatrix}$ and $\begin{bmatrix} 0_n \\ 1 \end{bmatrix}$ and minimizing instead of maximizing:

$$\begin{aligned} \text{minimize} \quad & \begin{bmatrix} 0_n \\ 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ v' \end{bmatrix} \\ \text{subject to} \quad & \begin{bmatrix} -A & 1_m \\ 1_n^T & 0 \end{bmatrix} \begin{bmatrix} x \\ v' \end{bmatrix} \leq \begin{bmatrix} 0_m \\ 1 \end{bmatrix} \\ & x \geq 0 \\ & v' \text{ free} \end{aligned} \tag{5}$$

Note that dual problem (5) is exactly equivalent to min-max problem (3). This means we can rewrite max-min problem (2) and min-max problem (3) as a primal/dual optimization problems. The strong duality theorem tells us that the objective function of a primal problem equals the objective function of its dual problem. That is:

$$\begin{bmatrix} 0_m \\ 1 \end{bmatrix} \cdot \begin{bmatrix} y \\ v \end{bmatrix} = \begin{bmatrix} 0_n \\ 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ v' \end{bmatrix}$$

This can be simplified to $v = v'$. In terms of our min-max (3) and max-min (2) equations, this means that the *value* of a game is well-defined. That is, the value of a game remains the same when a game is optimized from both player I and player II's perspective.

$$\max_y \min_x y^T A x = \min_x \max_y y^T A x$$

2.2.4. Implementation

In this chapter we have seen that we can rewrite a game with simultaneous decision making into linear optimization problems (2) and (3). Solving linear optimization problems can be done in Python with the PuLP library. This toolkit requires a set of variables, constraints and an objective function. It will then use optimization techniques to calculate valued variables that satisfy the constraints such that the objective function is maximum. The python code that solves such problems can be found in Appendix A.1.

In our case, we wish to solve the NHG. In Section 2.1, we have seen how we can build a payoff matrix A given the parameters m, n, p. This matrix will be the input of our max-min and min-max problem. The output of our problem will be the value of the game and the optimal solutions y and x for player I and II respectively. The results of the linear optimization technique and the exact solution (see section 3.1) on several different NHGs can be found in Appendix A.3.

3. The Original Number Hides Game

In section 2.1, we have seen we can write down a payoff matrix A that corresponds to the NHG with parameters m, n and p, With linear optimization, we can find the value of the game and the optimal solutions for both players. However, solving such a linear optimization problem takes more and more time whenever the parameters grow in size. It would be more elegant if we came up with a direct mathematical expression for the value and the optimal strategies in terms of the parameters. The latter has been done by Baston, Bostock and Ferguson in [2]. Their solution contains a direct formula of the value and optimal strategies of the game. In the upcoming chapter, we will take a look at an algorithmic approach of how to find this direct formula in constant time. In Section 3.1, we will jump right into the exact solution and a general overview of the proof.

3.1. Exact solution

Let $G(m, n, p)$ be an arbitrary NHG. Note that we must have $m, n, p \geq 1$ and $m, n \leq p$, otherwise the rules of the game won't make any sense. Even more, without loss of generality, we may assume $p \geq m \geq n \geq 1$. This is because when can reduce a game with $m < n$ to a game with $m \geq n$. This reduction is not trivial and can be justified with Claim 1 in Section 3.2.

Theorem 1 (Exact Solution) *Let $p \geq m \geq n \geq 1$. Write $p = Mm + r$ with $M \geq 1$ and $0 \leq r < m$. Then the value of the game is:*

$$V(m,n,p) = \begin{cases} \frac{n}{M+1} & \text{for } n \leq r < m \\ \frac{n(M+1)-r}{M(M+1)} & \text{for } 0 \leq r < n \end{cases} \tag{6}$$

Moreover, $q(m,p)$ and $q(m,p+m-n)$ are optimal strategies for player I and II respectively, where $q(m,p) = (q_1(m,p), \ldots, q_{p-m}(m,p))^T$ is a stochastic vector.

If $r = 0$, the components of q(m, p) are:

$$q_i(m,p) = \begin{cases} \frac{1}{M} & \text{for } j = im, i = 0, 1, \ldots, (M-1) \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

If $r \neq 0$, the components of q(m, p) are:

$$q_i(m,p) = \begin{cases} \frac{M-i}{M(M+1)} & \text{for } j = im, i = 0, 1, \ldots, (M-1) \\ \frac{i+1}{M(M+1)} & \text{for } j = im + r, i = 0, 1, \ldots, (M-1) \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

At first glance, this theorem might look a little intimidating. Before we will look at the proof of the theorem, we should explore the nature of the expression. Equation (6) suggests we should use divide NHGs into two cases:

I. $\mathbf{n \leq r \leq m}$ [3]. In this case, equation (6) tells us the value of the game is $\frac{n}{M+1}$. The value $V_{m,n}(p)$ as a function of p is constant within such intervals.

II. $\mathbf{0 < r < n}$. In this case, equation (6) tells us the value equals $\frac{n(M+1)-r}{M(M+1)}$. This expression linearly depends on the parameter p on the intervals $0 < r < n$, because $r = p \bmod M$. In fact, on these intervals, $V_{m,n}(p)$ is exactly the linear interpolation between the constant intervals from case I.

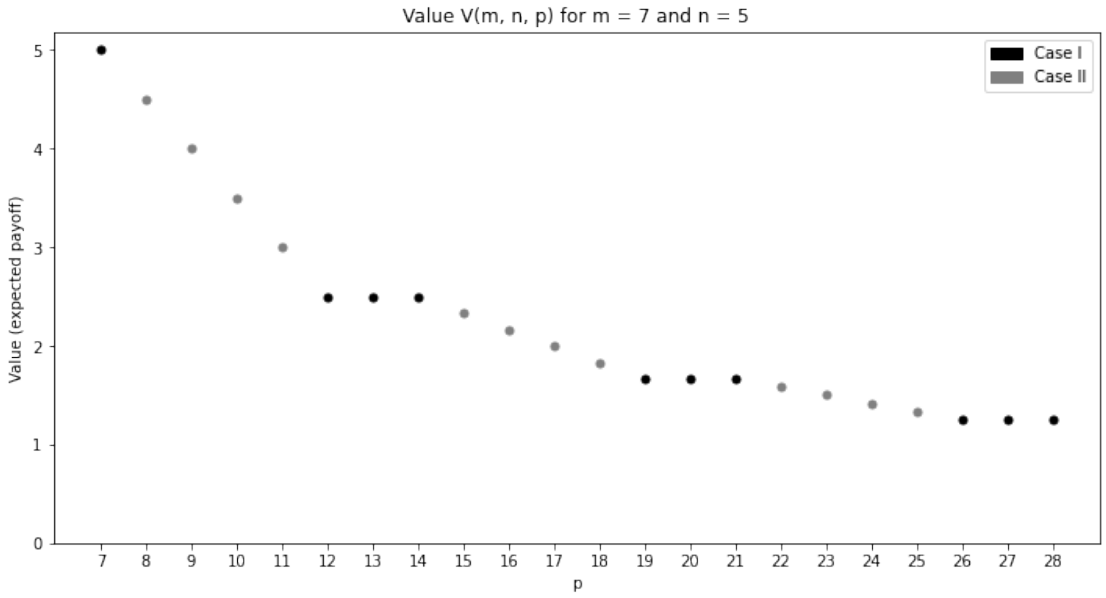

Figure 3: the value $V_{m,n}(p)$ of $G(m, n, p) = G(7, 5, p)$ as a function of p.

In Figure 3, it can be seen how the value (the expected payoff) of a NHG decreases as the boardsize increases. In the upcoming subsections, we will take a closer look at both the constant and the linear cases. First, we will look at some examples of some NHGs and the according optimal strategies $q(m, p)$ and $q(m, p + m - n)$. Then, we will see a proof of why these strategies lead to the value V in (6). The idea of this proof is similar to the min-max and max-min problems. We look at both players perspectives and conclude the value must be at least V from player I's perspective and the value can be at most V from player II's perspective. Hence, the value must be exactly V.

[3] The case where $r = m$ is not included in the theorem. In this case, one can rewrite $p = mM + r = mM' + r'$ with $M' = M + 1$ and $r' = 0$. Equation (6) then tells us $V = \frac{n'(M'+1)-r'}{M'(M'+1)} = \frac{n}{M+1}$.

For visual guidance of the proof, we will be considering $G(m,n,p) = G(7,5,15)$. We use Theorem 1 and collect the following results:

$$\begin{cases} p & = Mm + r = 2 \cdot 7 + 1 \\ M & = 2 \\ r & = 1 \\ V(m,n,p) & = \frac{n(M+1)-r}{M(M+1)} = 2\frac{1}{3} \\ q(m,n) & = \frac{1}{6}(2,1,0,0,0,0,0,1,2)^T \\ q(m,p+m-n) & = \frac{1}{6}(2,0,0,1,0,0,0,1,0,0,2)^T \end{cases}$$

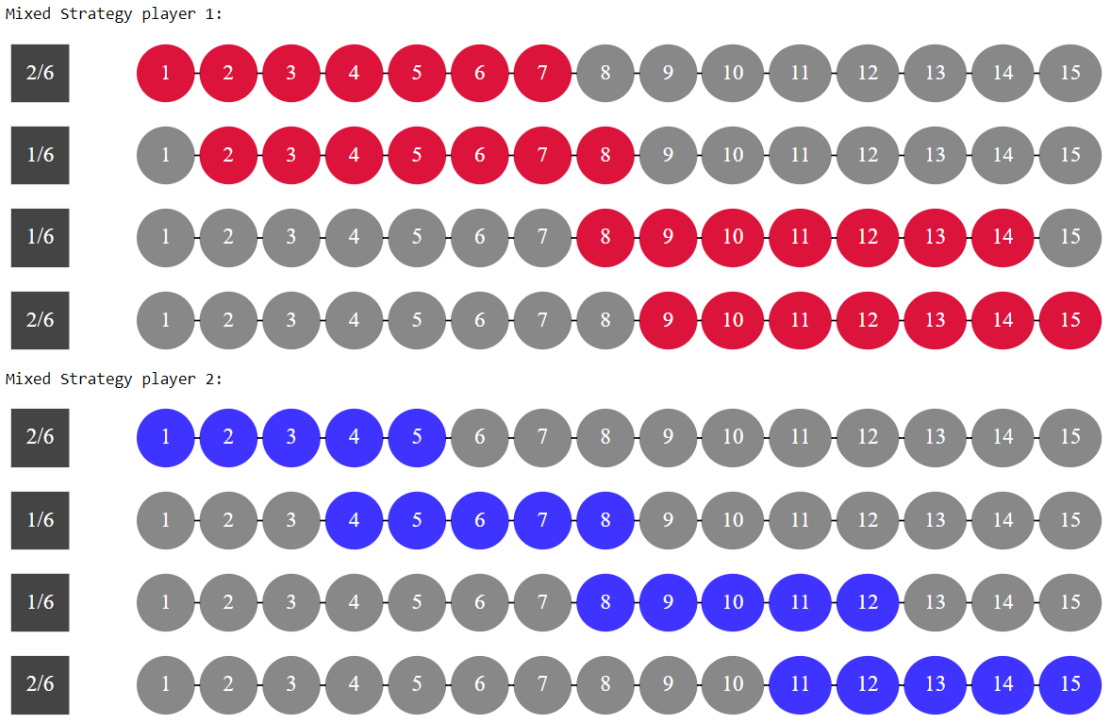

Figure 4: Optimal strategies $q(m,p)$ for Player I and $q(m,p+m-n)$ for Player II with weights > 0. These correspond to $G(m,n,p) = G(7,5,15)$.

In Figure 4[4], we see the optimal strategies for $G(7,5,15)$. If we set up a payoff matrix A (like we did in section 2.1), we can verify that $q(m,p)^T A q(m,p+m-n)$ indeed equals $V(m,n,p)$. This matrix and similar calculations can be found in Appendix A.3.

[4]The python code of this visualization can be found in Appendix A.4

3.1.1. Optimal strategy for player I

Just like in the max-min problem, player I needs to find a strategy y such that $\min_x y^T Ax$ is maximized. In other words, the strategy y must be able to get an expected payoff of at least $V(m,n,p)$ against all possible moves for player II. We will show the strategy $y = q(m,p)$ (like proposed in Theorem 1) is exactly such that:

$$\min_x y^T Ax = V(m,n,p) = \begin{cases} \frac{n}{M+1} & \text{for } n \leq r < m \\ \frac{n(M+1)-r}{M(M+1)} & \text{for } 0 \leq r < n \end{cases} \tag{9}$$

The matrix multiplication $y^T Ax$ describes the expected payoff of a game with strategies y and x. In an alternative method to calculate the expected payoff, we look at each coin individually. Recall the definition of the payoff in a NHG: *the number of coins in the overlap of the two chosen intervals.* To calculate the expected payoff, we must sum over all coins and add up the product of the probabilities that a specific coin c is selected by x and y. This is exactly the probability that coin c is part of the overlap and thus contributes to the expected payoff.

$$V(m,n,p) = y^T Ax = \sum_{c=1}^{p} \mathbb{P}(c \text{ covered by strategy } y)\mathbb{P}(c \text{ covered by strategy } x) \tag{10}$$

Let $y = q(m,p)$. We split up the coins in two groups:

1. Left-sided. $c = im + r_c$ with $i \geq 0$ and $1 \leq r_c \leq r$.
   Recall the definition of q in (8). Note that the only potentially non-zero weighted strategies that include coin c are im and $(i-1)m + r$. The probability that c is covered by strategy y is therefore $q_{im} + q_{(i-1)m+r} = \frac{M-i}{M(M+1)} + \frac{(i-1)+1}{M(M+1)} = \frac{1}{M+1}$.
2. Right-sided. $c = im + r_c$ with $i \geq 0$ and $r + 1 \leq r_c \leq m$
   If $r \neq 0$, the definition in (8) tells us the only non-zero weighted strategies that include coin c are im and $im + r$. Thus, probability that c is covered by strategy y is $q_{im} + q_{im+r} = \frac{M-i}{M(M+1)} + \frac{i+1}{M(M+1)} = \frac{1}{M}$.
   If $r = 0$, the definition in (7) tells us im is the only non-zero strategy that includes coin c. Thus, the probability that c is covered by strategy y is exactly $q_{im} = \frac{1}{M}$.

Combining these two cases gives us the following expression for the probability that coin c is covered by strategy y:

$$\mathbb{P}(c \text{ covered by strategy } y) = \begin{cases} \frac{1}{M+1} & \text{for } 1 \leq c \leq r \bmod m \\ \frac{1}{M} & \text{otherwise} \end{cases} \tag{11}$$

In Figure 5, the coin coverage is visualized with the color yellow. Observe that the probabilities are divided into the chunks $\{\{1,\ldots,m\},\{m+1,\ldots,2m\},\ldots\{Mm+1,\ldots,p\}\}$. Each of these chunks has a left-side (with low coverage) and a right-side (with high coverage).

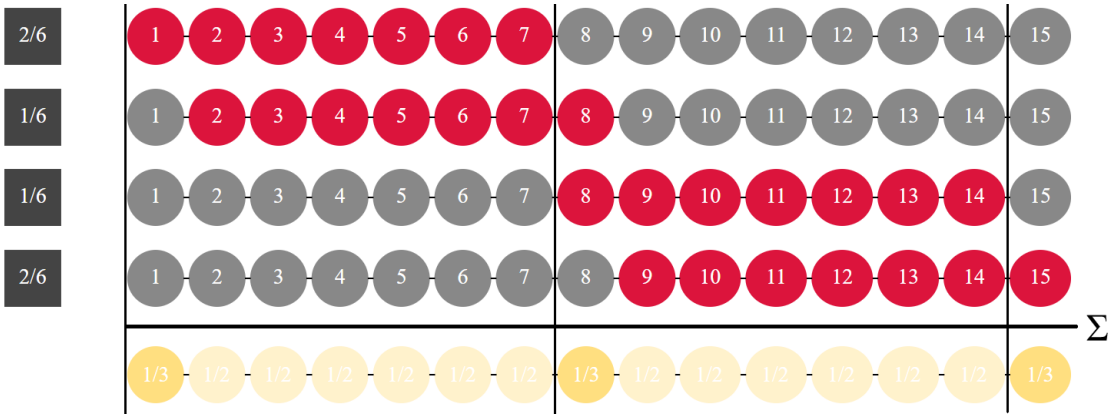

Figure 5: Example of the mixed strategy $y = q(m,p) = q(7,15)$ ($p = Mm+r = 2\cdot 7+1$). The probability that coin c is covered is either $\frac{1}{M+1}$ or $\frac{1}{M}$.

Now we have to reason what is the best strategy x for player II to counter the strategy y of player I. Recall that player II's goal is to minimize the payoff. Therefore, player II needs to choose a strategy that covers the coins with the lowest probability of being covered by x. Note that since $m \geq n$, player II can never choose an interval that is bigger than the chunks of size m. From equation (11) and Figure 5, it is clear that player II wishes to maximize the number of coins on the left-side of an interval. A strategy that satisfies this is $x = (1,0,\ldots,0)^T$. This means that player II chooses the first n coins with probability 1. The corresponding probability that coin c is covered by strategy x is:

$$\mathbb{P}(c \text{ covered by strategy } x) = \begin{cases} 1 & \text{for } c \leq n \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

Inserting (11) and (12) into (10) gives us:

$$\begin{aligned} V(m,n,p) = y^T A x &= \sum_{c=1}^{p} \mathbb{P}(c \text{ covered by strategy } y)\mathbb{P}(c \text{ covered by strategy } x) \\ &= \sum_{c=1}^{n} \begin{cases} \frac{1}{M+1} & \text{for } c \leq r \\ \frac{1}{M} & \text{otherwise} \end{cases} \\ &= \begin{cases} \frac{n}{M+1} & \text{for } r \geq n \\ \frac{r}{M+1} + \frac{n-r}{M} & \text{for } r < n \end{cases} \\ &= \begin{cases} \frac{n}{M+1} & \text{for } n \leq r < m \\ \frac{n(M+1)-r}{M(M+1)} & \text{for } 0 \leq r < n \end{cases} \end{aligned}$$

We have shown that player I can a choose strategy ($y = q(m,p)$) such that the value is at least V(m,n,p), like in Theorem 1. This finishes the first part of the proof.

3.1.2. Optimal strategy for player II

What remains to show is that player II can choose a strategy ($x = q(m, p+m-n)$) such that the value is at most $V(m, n, p)$. In the proof of this part, we separately consider 5 different cases based on the parameters m, n and $p = Mm + r$. For each case we will take the following steps to show player II assures a maximum payoff of $V(m, n, p)$:

1. Write down the optimal strategy for player II like proposed in the theorem.
2. For each coin, we calculate the probability it is covered by the optimal strategy. When we speak of the *coverage* of coin c, we mean $\mathbb{P}(c \text{ covered by strategy } x)$.
3. Show that strategy $0 = \{1, 2, \ldots, m\}$ for player I attains payoff $V(m, n, p)$
4. Show that no other strategy for player I attains a higher payoff.

In every of the 5 cases, we consider chunks of m coins like in Figure 6. Strategy $0 = \{1, 2, \ldots, m\}$ for player I contains exactly all the coins in the first chunk.

$$\text{Payoff} \;=\; \text{coin coverage in the first chunk} = \sum_{c=1}^{m} \mathbb{P}(c \text{ covered by strategy } x) \qquad (13)$$

Case (i): $r = n$.

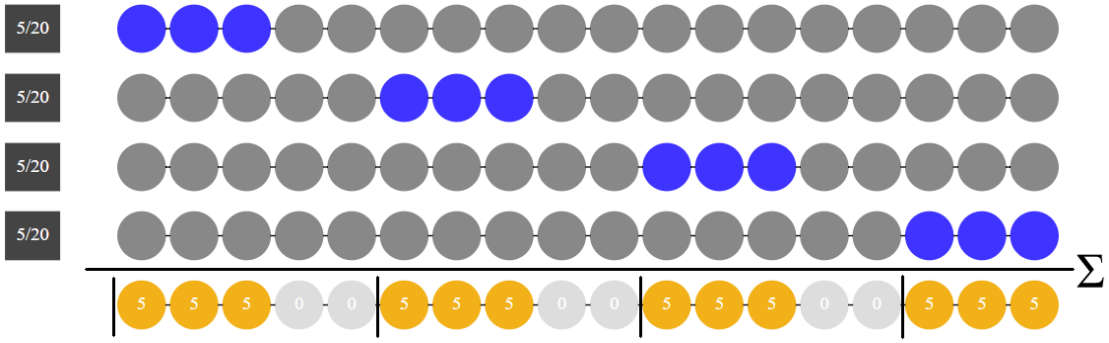

Figure 6: Example of case (i). $G(m, n, p) = G(5, 3, p = Mm + r = 5 \cdot 3 + 3)$. Strategy x followed by $20 \cdot \mathbb{P}(c \text{ covered by strategy } x)$.

Notice that $q(m, p + m - n)$ must be calculated like defined in (7). This is because we have $p + m - n = Mm + r + m - n = (M + 1)m = M'm + r'$ with $M' = M + 1$ and $r' = 0$. We find $q_i = \frac{1}{M'} = \frac{1}{M+1}$ whenever i is divisible by m. The chunks are divided into two intervals. The first interval contains $n = 3$ coins with coverage $\frac{1}{M+1} = \frac{1}{4} = \frac{5}{20}$. The second interval contains $m - n = 2$ coins with coverage 0. We can calculate the payoff like proposed in (13):

$$\text{Payoff} \;= \sum_{c=1}^{m} \mathbb{P}(c \text{ covered by strategy } x) = \sum_{c=1}^{n} \frac{1}{M+1} + \sum_{c=n+1}^{m} 0 = \frac{n}{M+1} = V(m, n, p)$$

Now imagine any another strategy for player I of m consecutive coins. Observe that any coin in this strategy corresponds with a coin in the first chunk with same coverage. Therefore, the payoff must be the same for all strategies for player I.

Case (ii): $r < n$ and $2n \leq m + r$

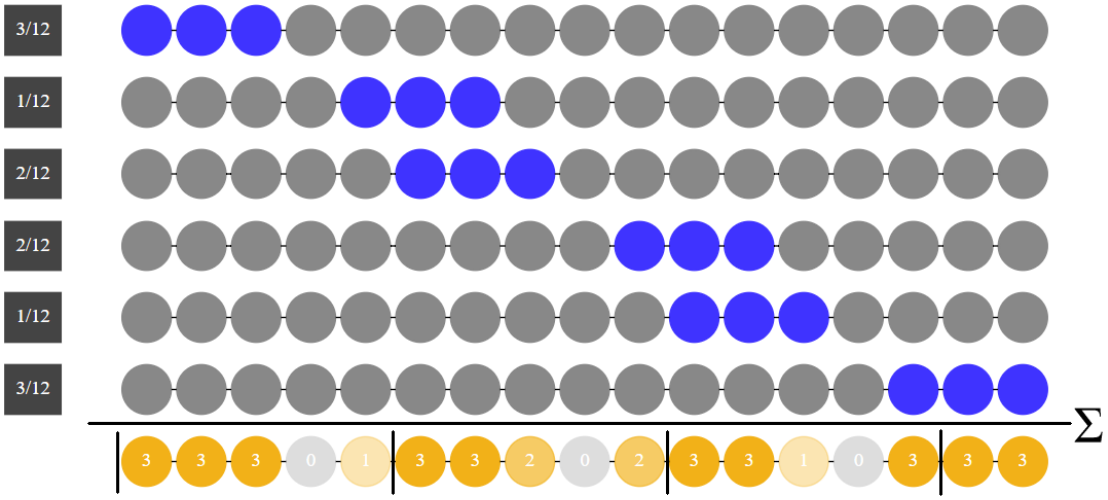

Figure 7: Example of case (ii). $G(m, n, p) = G(5, 3, p = Mm + r = 5 \cdot 3 + 2)$. Strategy x followed by $12 \cdot \mathbb{P}(c \text{ covered by strategy } x)$.

We will use the definition of q from (8) to find the mixed strategy $q(m, p + m - n)$. We calculate the coin coverage by taking a weighted sum like in Figure 7.

Let $i \in \{1, 2, \ldots, M+1\}$ denote the chunk index. Notice that each chunk is divided into 4 constant intervals with the following [coverage, length]

$$[\frac{M}{M(M+1)}, r], [\frac{M+1-i}{M(M+1)}, n-r], [0, m+r-2n], [\frac{i}{M(M+1)}, n-r]$$

$$= [\frac{3}{12}, 2], [\frac{4-i}{12}, 1], [\frac{0}{12}, 1], [\frac{i}{12}, 1]$$

We can calculate the payoff like proposed in (13) by summing over the coin coverage in the first chunk ($i = 1$). This means that for each interval, we add up the of the coin coverage within each interval times its length:

$$\begin{aligned} \text{Payoff} &= \frac{M}{M(M+1)} r + \frac{M}{M(M+1)}(n-r) + 0(m+r-2n) + \frac{1}{M(M+1)}(n-r) \\ &= \frac{n(M+1)-r}{M(M+1)} = V(m, n, p) \end{aligned}$$

Why is $\mathbf{0} = \{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{m}\}$ optimal for player I? [5]
Imagine any other strategy for player I of m consecutive coins. Observe that every coin c' in this new strategy corresponds[6] to a coin c in the first chunk on the same interval. We wish to show $\sum_{c'} \mathbb{P}(c' \text{ covered by strategy } y') \leq \sum_{c} \mathbb{P}(c \text{ covered by strategy } y)$.

Dependent parts. Two of the intervals depend on the chunk index. The dependent parts are $-\frac{i}{M(M+1)}$ and $\frac{i}{M(M+1)}$ respectively. Notice that in each chunk, the interval with the negative dependent part comes before the interval of the positive dependent part. Furthermore, these intervals are of equal length. Let us pair up the coins in the first chunk accordingly: (c_-, c_+) Where c_- and c_+ denote coins in the interval with negative and positive dependent parts respectively. Since $i = 1$, we find that the sum of the dependent parts in a pair is $-\frac{1}{M(M+1)} + \frac{1}{M(M+1)} = 0$. So the sum of all the dependent parts is 0.

$$\sum_{c'} \mathbb{P}(c' \text{ covered by strategy } x)_{dependent} \leq 0 = \sum_{c} \mathbb{P}(c \text{ covered by strategy } x)_{dependent}$$

We can also pair up the c' coins accordingly: (c'_-, c'_+). By the ordering of the intervals, the chunk index i of c'_- must be at least as high as the chunk index j of c'_+. Therefore, the sum of the dependent parts is $-\frac{i}{M(M+1)} + \frac{j}{M(M+1)} \leq 0$. So the sum of all the dependent parts never exceeds 0.

Independent parts. The other two intervals and the independent parts of the dependent intervals are independent of the chunk index. Therefore, the coin coverages of c and c' are the same on these parts.

$$\sum_{c'} \mathbb{P}(c' \text{ covered by strategy } x)_{independent} = \sum_{c} \mathbb{P}(c \text{ covered by strategy } x)_{independent}$$

When combining the dependent and independent parts, we find:

$$\begin{aligned}
&\sum_{c'} \mathbb{P}(c' \text{ covered by strategy } x) \\
&= \sum_{c'} \mathbb{P}(c' \text{ covered by strategy } x)_{dependent} + \sum_{c'} \mathbb{P}(c' \text{ covered by strategy } x)_{independent} \\
&\leq \sum_{c} \mathbb{P}(c \text{ covered by strategy } x)_{dependent} + \sum_{c} \mathbb{P}(c \text{ covered by strategy } x)_{independent} \\
&= \sum_{c} \mathbb{P}(c \text{ covered by strategy } x)
\end{aligned}$$

We have shown that no other strategy for player I attains a higher payoff. The same argument also holds in case (iii), (iv) and (v).

[5] Or rather, why is 0 included in an optimal strategy for player I?
[6] In this context, c' corresponds with c wherever $c' \equiv c$ modulo m

Case (iii): $r < n$ and $2n > m + r$

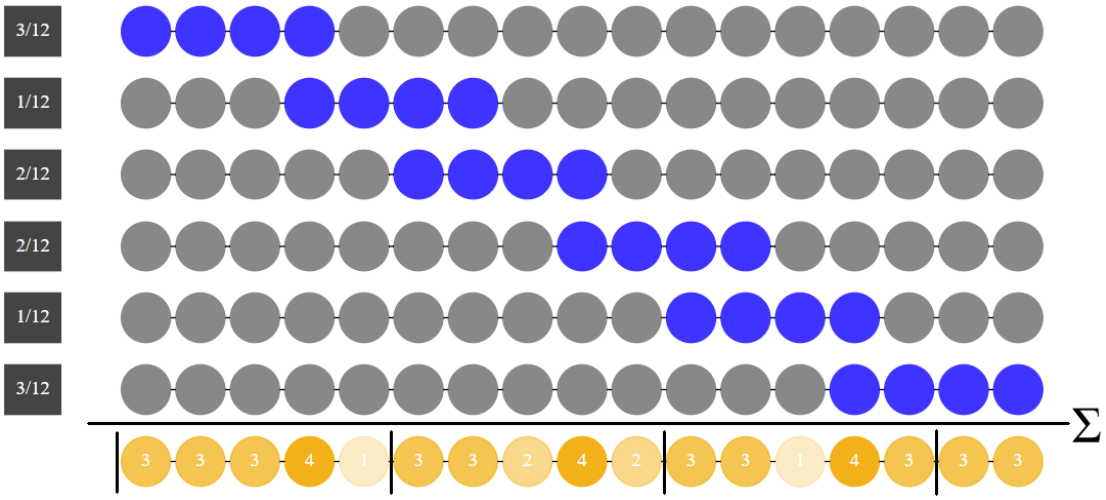

Figure 8: Example of case (iii). $G(m, n, p) = G(5, 4, p = Mm + r = 5 \cdot 3 + 2)$. Strategy x followed by $12 \cdot \mathbb{P}(c \text{ covered by strategy } x)$.

We will use the definition of q from (8) to find the mixed strategy $q(m, p + m - n)$. We calculate the coin coverage by taking a weighted sum like in Figure 7. Notice that each chunk is divided into 4 constant intervals with the following [coverage, length]:

$$[\frac{M}{M(M+1)}, r], [\frac{M+1-i}{M(M+1)}, m-n], [\frac{M+1}{M(M+1)}, 2n-m-r], [\frac{i}{M(M+1)}, m-n]$$

$$= [\frac{3}{12}, 2], [\frac{4-i}{12}, 1], [\frac{4}{12}, 1], [\frac{i}{12}, 1]$$

We can calculate the payoff like proposed in (13) by summing over the coin coverage in the first chunk ($i = 1$). This means that for each interval, we add up the of the coin coverage times its length:

$$\begin{aligned} \text{Payoff} &= \frac{M}{M(M+1)} r + \frac{M}{M(M+1)}(m-n) + \frac{M+1}{M(M+1)}(2n-m-r) + \frac{1}{M(M+1)}(m-n) \\ &= \frac{n(M+1)-r}{M(M+1)} = V(m, n, p) \end{aligned}$$

Case (iv): $r > n$ and $2n <= r$.
Again, each chunk is divided into 4 constant intervals with [coverage, length]:

$$[\frac{M+2-i}{(M+1)(M+2)}, n], [0, r-2n], [\frac{i}{(M+1)(M+2)}, n], [0, m-r]$$

We calculate the payoff like proposed in (13) using the product-sum:

$$\begin{aligned}\text{Payoff } &= \frac{M+2-1}{(M+1)(M+2)}n + 0(r-2n) + \frac{1}{(M+1)(M+2)}n + 0(m-r)\\ &= \frac{(M+2)n}{(M+1)(M+2)} = \frac{n}{M+1} = V(m,n,p)\end{aligned}$$

Case (v): $r > n$ and $2n > r$.
Again, each chunk is divided into 4 constant intervals with [coverage, length]:

$$[\frac{M+2-i}{(M+1)(M+2)}, r-n], [\frac{M+2}{(M+1)(M+2)}, 2n-r], [\frac{i}{(M+1)(M+2)}, r-n], [0, m-r]$$

We calculate the payoff like proposed in (13) using the product-sum:

$$\begin{aligned}\text{Payoff } &= \frac{M+2-1}{(M+1)(M+2)}(r-n) + \frac{M+2}{(M+1)(M+2)}(2n-r) + \frac{1}{(M+1)(M+2)}(r-n) + 0(m-r)\\ &= \frac{(M+2)n}{(M+1)(M+2)} = \frac{n}{M+1} = V(m,n,p)\end{aligned}$$

Conclusion.
We have considered the problem from the perspective of player II. We will combine the cases to conclude that in general, for any m, n, p, we have:

$$V(m,n,p) = \begin{cases} \frac{n}{M+1} & \text{for } n \le r < m \\ \frac{n(M+1)-r}{M(M+1)} & \text{for } 0 \le r < n \end{cases}$$

We have shown that player II can a choose strategy (x=q(m, p+m-n)) such that the value is at most $V(m,n,p)$, just like in Theorem 1. This finishes the second part of the proof.

In Section 3.1.1, we found that value is at least $V(m,n,p)$ when player I uses strategy $y = q(m,p)$. And in this section, we found that the value is at most $V(m,n,p)$ when player II uses strategy $x = q(m, p+m-n)$. Hence, Theorem 1 is correct.

3.2. Reduction if m < n

Claim 1 *We can reduce a game $G(m, n, p)$ with $m < n$ to a game $G'(m', n', p')$ with $m' \geq n'$ such that G and G' share the same value and share optimal strategies y and x up to a symmetrical extension of y with zeroes to match the dimensions of $G(m, n, p)$.*

Let us start off by considering an example.

$$G(3,5,10): \begin{pmatrix} 3 & 2 & 1 & 0 & 0 & 0 \\ 3 & 3 & 2 & 1 & 0 & 0 \\ 3 & 3 & 3 & 2 & 1 & 0 \\ 2 & 3 & 3 & 3 & 2 & 1 \\ 1 & 2 & 3 & 3 & 3 & 2 \\ 0 & 1 & 2 & 3 & 3 & 3 \\ 0 & 0 & 1 & 2 & 3 & 3 \\ 0 & 0 & 0 & 1 & 2 & 3 \end{pmatrix} \quad G(5,3,8): \begin{pmatrix} 3 & 3 & 3 & 2 & 1 & 0 \\ 2 & 3 & 3 & 3 & 2 & 1 \\ 1 & 2 & 3 & 3 & 3 & 2 \\ 0 & 1 & 2 & 3 & 3 & 3 \end{pmatrix} \tag{14}$$

Player I wishes to maximize the payoff by choosing rows with high values. Consider the game G(3, 5, 10) and its corresponding payoff matrix in (14). Note that the top two rows are strictly less than the third row. And the bottom two rows are strictly less than row 6.

We say that a row i **dominates** row i' if $\forall j : a(i, j) \geq a(i', j)$. In the context of game theory, this means that player I never has to choose a row i' that is dominated by row i. Let y be the stochastic vector of the optimal strategy for player with $y'_i > 0$. Then we can update this strategy without changing the optimality:

$$y_i \leftarrow y_i + y_{i'}$$
$$y_{i'} \leftarrow 0$$

Since row 1, 2, 7 and 8 are dominated, there exist optimal strategies $(0, 0, y, 0, 0)$ and x in $G(m, n, p)$. We remove the dominated rows 1, 2, 7 and 8 from the payoff matrix of $G(3, 5, 10)$ in (14). After this operation, we end up with a matrix that is exactly the same as the payoff matrix of game $G(5, 3, 8)$. The strategies y and x are also optimal in $G(5, 3, 8)$. In general, the reduction we applied here works when $m < n$ and $2n < p+m$. Then the top $n-m$ rows are dominated by the $(n-m+1)$th row and the bottom $n-m$ rows are dominated by the $(p-n+1)$th row. Therefore, we can eliminate these rows and reduce $G(m, n, p)$ to $G(m', n', p') = G(n, m, p-n+m)$.

$$G(3,7,10): \begin{pmatrix} 3 & 2 & 1 & 0 \\ 3 & 3 & 2 & 1 \\ 3 & 3 & 3 & 2 \\ 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 \\ 2 & 3 & 3 & 3 \\ 1 & 2 & 3 & 3 \\ 0 & 1 & 2 & 3 \end{pmatrix} \quad G(6,3,6): \begin{pmatrix} 3 & 3 & 3 & 3 \end{pmatrix} \tag{15}$$

In equation (15), we see that a lot of rows in the payoff matrix of $G(3, 7, 10)$ are dominated. We can eliminate all rows except the first center row, which dominates all other rows. We end up with the payoff matrix of a new game, $G(m', n', p') = G(p - n + m, m, p - n + m) = G(6, 3, 6)$.

In general, if $2n \geq p + m$, then the $\lfloor \frac{p-m+1}{2} \rfloor$th row (also known as the first center row) dominates all the other rows. Therefore, we can eliminate these rows. We end up with the simple game $G(m', n', p') = G(p - n + m, m, p - n + m)$ with $m' \geq n'$. The optimal strategy y for player I is just choosing the only row available. Any strategy x for player II is optimal. We can transform these strategies back into the original game $G(m, n, p)$. The optimal strategies are $(0, \ldots, 0, 1, 0, \ldots, 0)^T$ and x. Where the zeroes denote a symmetric extension until the strategy meets the required dimension of $p-m+1$.[7]

Let us summarize the details of the claim. We can reduce the game $G(m, n, p)$ with $m < n$ to a game $G(m', n', p') = G(\min\{n, p - n + m\}, m, p - n + m)$. The optimal solutions y and x in G' are also optimal solutions in G. The only difference is the fact that y has to be extended to $(0, \ldots, 0, y, 0, \ldots, 0)^T$ such that it has $p - m + 1$ elements.

3.3. Python implementation

In Section 2.2, we have looked at a numeric method to solve games with simultaneous decision making. The execution of this method will get more expensive as the size of payoff matrix A increases. That's why we looked at a more elegant way of solving the NHG specifically. The results of this way were described in Section 3.1. In Appendix A.2, a python script that utilizes the exact solution of Theorem 1 can be found. Furthermore, in Appendix A.3, several NHGs are tested on both the numerical and the exact solution.

[7] If $p - m + 1$ is even, we add one more 0 to the right than to the left.

4. The Number Hides Game on a tree

In Chapter 3, we have considered the easiest type of board to play the NHG on. Namely, a line with p coins. In this chapter, we will extend the complexity of the board to a tree of coins. A tree is a structure of c coins with exactly $c-1$ edges indicating whether coins are neighbouring or not. Furthermore, the edges have to be such that any two coins are connected though a path of neighbouring coins. We say a list of coins is *consecutive* whenever they form a path of neighbouring coins.

The rules of the extended game remain the same as in the original game. Both players simultaneously choose m and n consecutive coins respectively. Then, the payoff of the game is calculated by counting the number of coins in the overlap of both choices. This is the amount player II has to pay to player I. Player I acts as the hider and wishes to maximize the payoff, while player II acts as the searcher and wishes to minimize the payoff. In this chapter, we will look for optimal strategies[8].

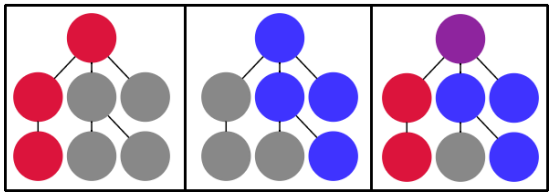

Figure 9: Player I chose $m = 3$ consecutive coins. Player II chose $n = 4$ consecutive coins. The payoff is 1.

4.1. Numerical Approach

Before we search for an exact solution for the NHG on a tree, we will solve the game numerically. For this, we will reuse the linear optimization program we used in Section 2.2. This will be a brute force way to find the value of the game and the corresponding optimal values. After we find these solution, we can look for conjectures to come up with an exact solution.

In this section, we will look at most of the elements from the python code about the NHG on a tree from Appendix A.5. Let us start by defining a general tree. If we can solve the NHG on a general tree, we can easily reduce it to games of the form G(m, n, B, b) later on. A tree consists of several objects of the class type node. Each node has a list of references to neighbouring nodes. When two nodes included each other in their 'neighbors' list, then the nodes are connected by an edge.

To solve the NHG on a tree, we need to enlist all the possible moves a player can pick m consecutive coins. In other words, we need to find all paths of length m within a tree. This can be done with a recursive search[9].

[8] Recall from Section 2 that a strategy is defined as a stochastic vector that assigns probabilities to all individual moves from the strategy set. An optimal strategy assures the best possible payoff against any move from the opponent.

[9] Algorithm 1 contains a simplified explanation of the code, the actual code is included under get_edges in Appendix A.5. This code keeps track of the visited nodes and throws away duplicate paths.

Algorithm 1: find_path(root, m)

```
if m = 1 then
    Trace back the path and append to the list;
else
    for neighbor in root.unvisited_neighbors do
        find_path(neighbor, m-1);
    end
end
```

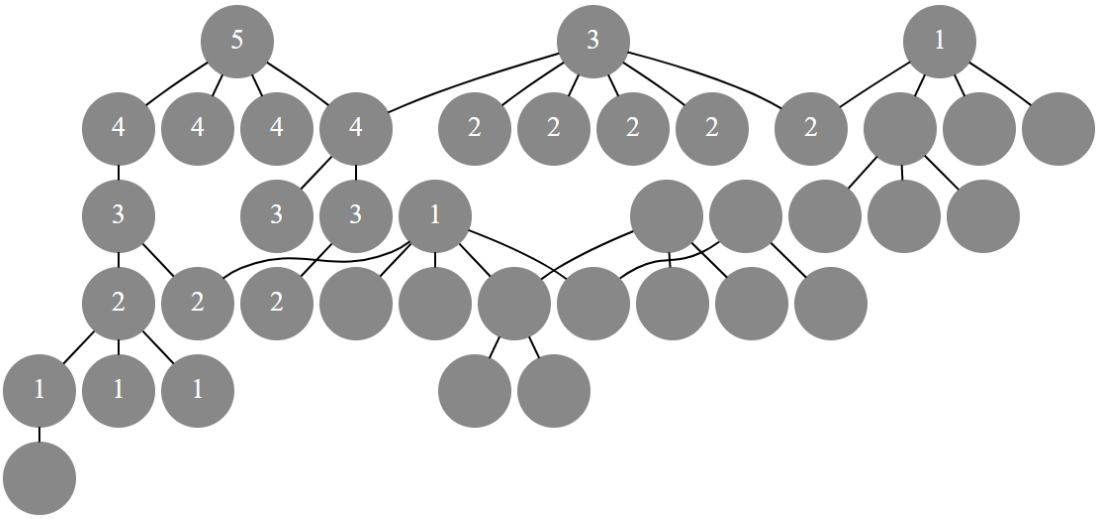

Figure 10: find_path(root, 5) applied on an arbitrary root on a tree. The labels on each coin represent the input value of m when that coin acts as the 'root'.

We run find_path(c, m) for each coin c to enlist all the possible paths of length m. Then, we also run find_path(c, n) for each coin c to enlist all the possible paths of length n for player II. To build the payoff matrix A, we consider every combination from these two lists and calculate the overlap (= the payoff). The final step is to apply the linear optimization program from Section 2.2 to calculate the optimal strategies y and x. These strategies give us the probability that a specific move will be chosen.

An example of non-zero weighted moves in the optimal strategies y and x on a tree with parameters $m = 5$ and $n = 4$ can be found in Figure 11. It is hard to see a pattern when looking at such an elaborated figure. That's why we will merge the optimal strategies into *coverage images*. Each coin c will have an opacity value equal to the weighted sum of strategies that include coin c. Opaque coins have a high coverage and transparent coins have a low coverage. Furthermore, gray coins have no coverage. An example of such a coverage image can be found in Figure 12.

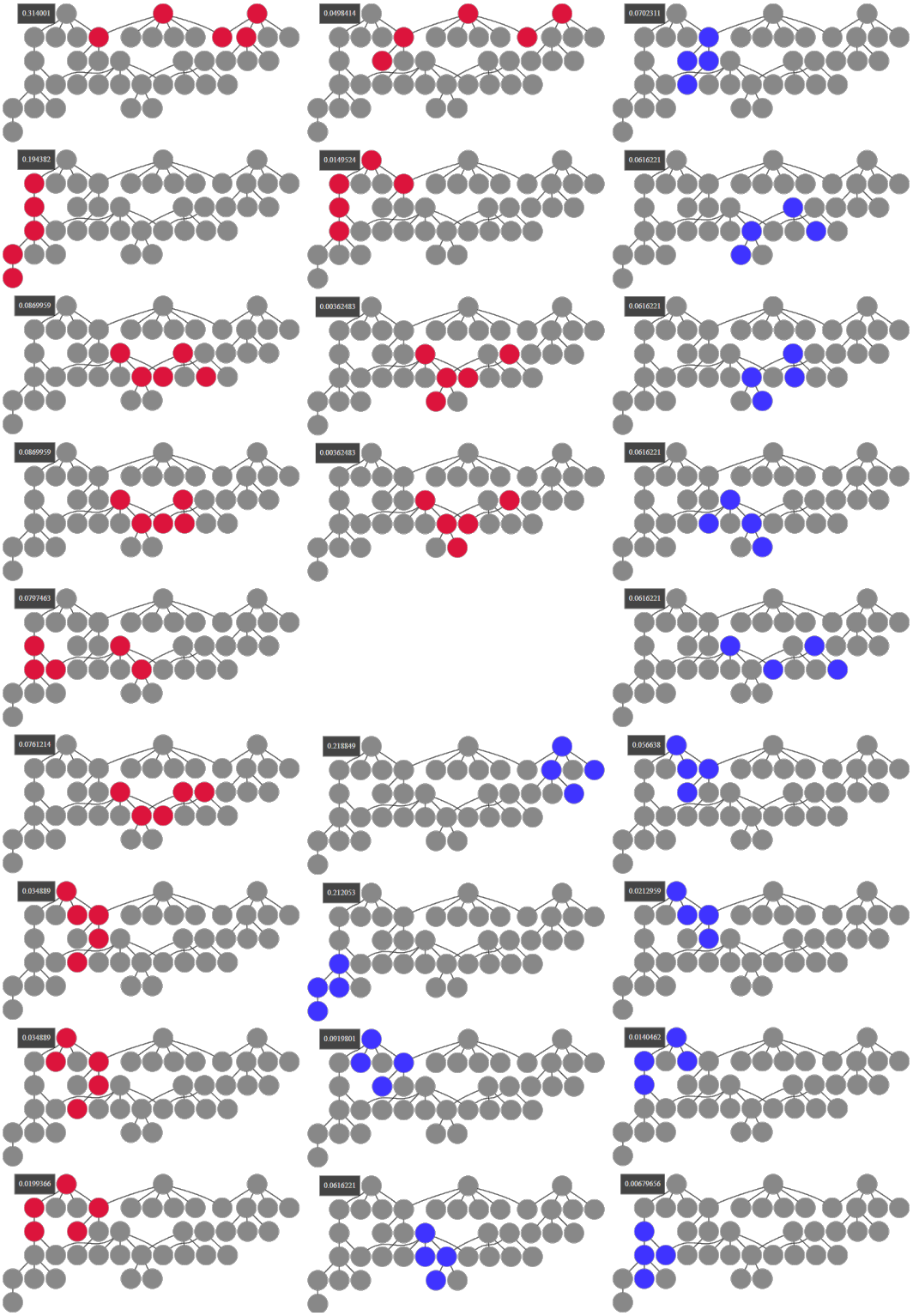

Figure 11: Moves in strategy y (13 nonzero moves / 89 total moves) and strategy x (13 nonzero moves / 87 total moves) after running the linear optimization program. The game was played with parameters $m = 5$ and $n = 4$ on a random tree.

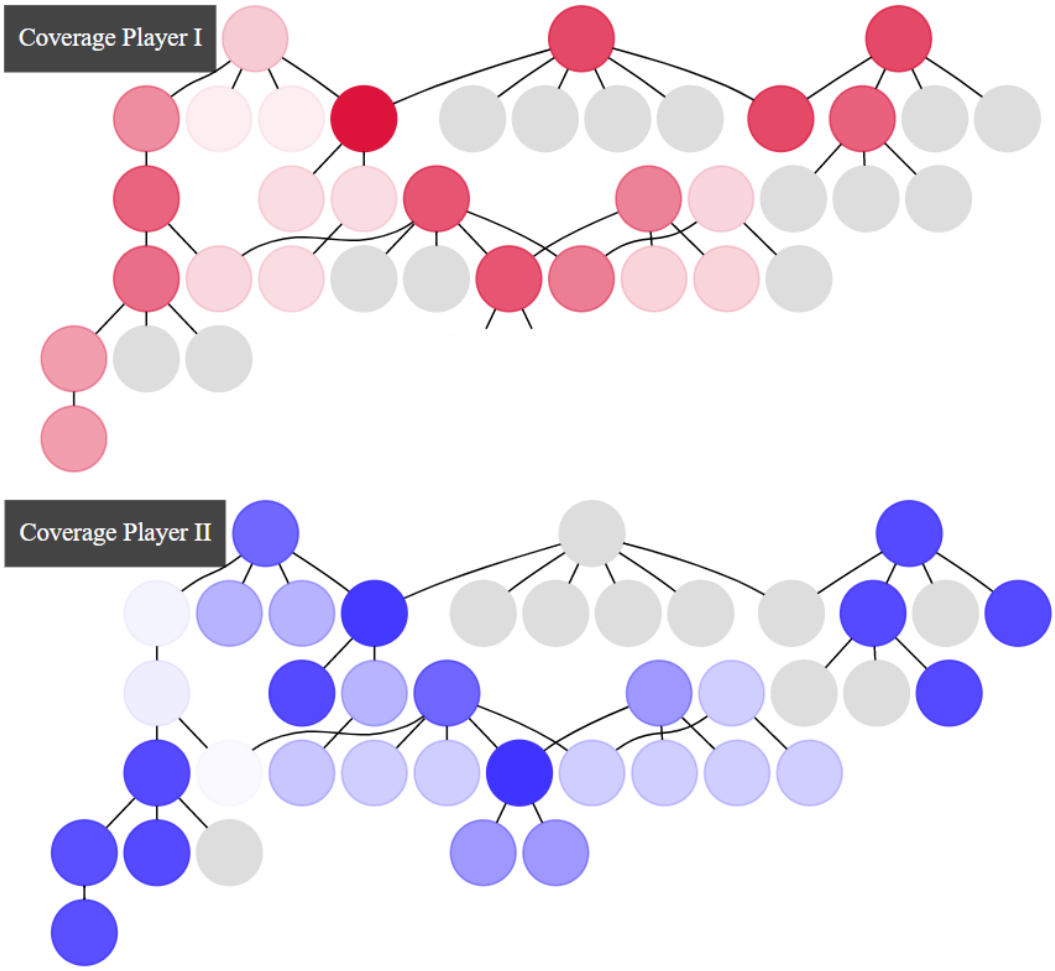

Figure 12: Coin coverage of strategy x and y from Figure 11.

We can retrieve the value of the game directly from the min-max or max-min linear program. Or we can calculate it using the formula $y^T Ax$ with the optimal strategies y and x. In the case of the tree from Figure 11 and Figure 12, the value is 0.677843. So on average, there is an overlap of 0.677843 coins between the move from player I and player II.

Our goal for this chapter is to use this brute force method to find patterns that can lead to a direct formula for the value of the game. To greatly reduce the complexity of this direct formula, we restrict our attention to a sub-type of a tree: spiders with an equal branchsize. Such a tree can be described with two parameters: B and b. We will discuss the details of this type of tree in the next subsection.

4.2. The NHG on a spider

Up until this point, we have looked at the Number Hides Games with parameters m, n and p, in which the coins lie on a single line. In this chapter we will extend the NHG by increasing the complexity of the board from a single line to multiple lines. We will introduce the concept of an origin. Coin 0 takes up the spot in the origin[10]. This root coin is connected to B branches with branchsize b. Several examples of this layout are provided in Figure 13. Such layouts are called spiders with an equal branchsize.

In this chapter, we consider NHGs of the form $G(m, n, B, b)$. Where m and n still denote the amount of consecutive coins that are to be chosen by player I and II respectively and B and b describe the layout of the board. Note that any NHG of the form $G(m, n, p)$ can be extended to a game $G(m, n, B, b)$ with $B = 1$ and $b = p - 1$ (or alternatively, $B = 2$ and $b = \frac{p-1}{2}$ if p is odd)

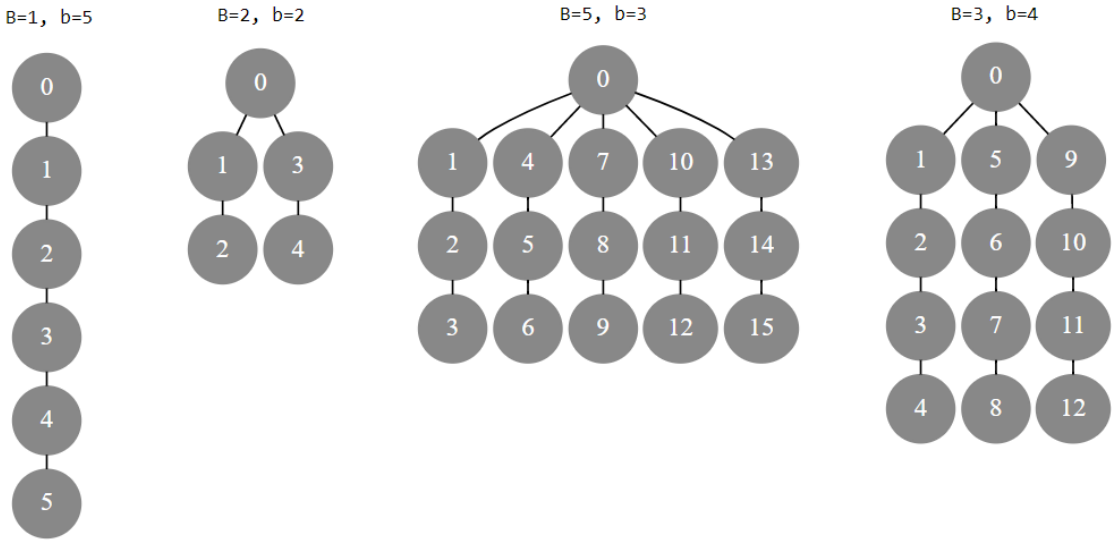

Figure 13: Several examples of the extended board of $G(m, n, B, b)$

Now, let us define a strategy set. Player I has to choose a consecutive sequence of m coins with exactly two endpoints. We will enlist all the moves into two categories:

[10]From this point onward, we denote this coin 0 by the *root coin*

- **Straight** moves. Straight moves only include coins from a single branch and/or the root coin. Figure 14 contains all the straight moves from the first branch. Note that this sub-strategy set is isomorphic to the full strategy set from the original NHG. In general, there are $b - m + 2$ straight moves in each of the B branches.

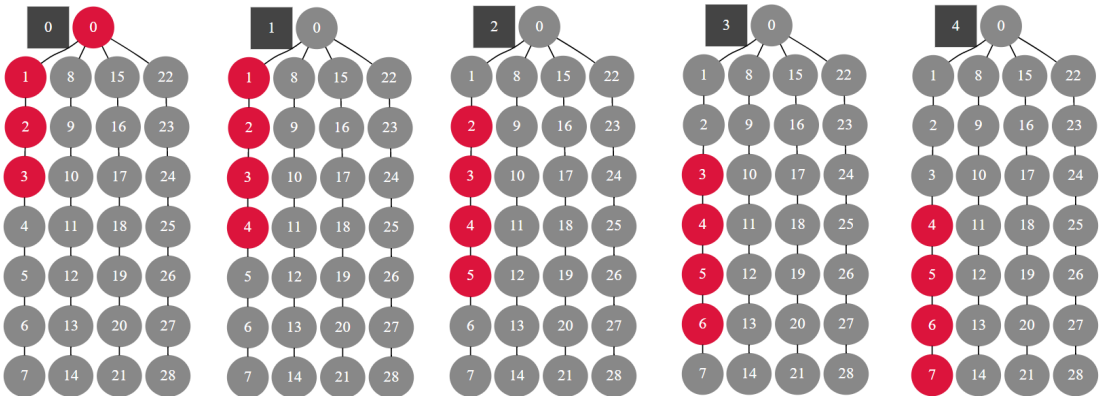

Figure 14: All straight moves in the first branch ($m = 4$, $B = 4$, $b = 7$)

- **Corner** moves. Corner moves always include the root coin and at least one coin from exactly two different branches. Figure 15 contains all the corner moves moving from the 1st branch to the 4th branch. We can enlist all corner moves by considering all combinations of 2 branches. In general, there are $m - 2$ corner moves in each of the $\frac{B(B-1)}{2}$ combinations of 2 branches.

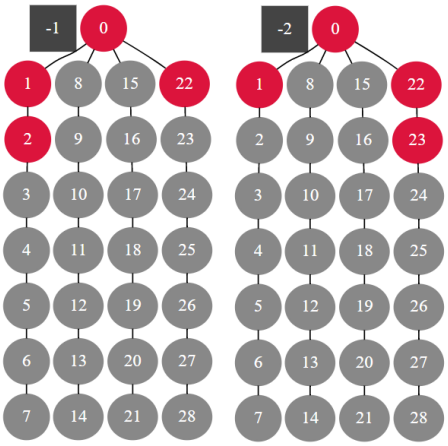

Figure 15: All corner moves from the 1st branch to the 4th branch ($m = 4$, $B = 4$, $b = 7$)

When combining Figure 14 and 15, we find a total of 7 moves: $(-2, -1, 0, 1, 2, 3, 4)$. The negative labels for the corner moves represent the number of coins in the second branch.

Note that each move has different variants depedending on which branch it is applied. For example, straight move 3 has $B = 4$ variants and corner move -2 has $\frac{B(B-1)}{2} = 6$ variants. (one for each combination of branches)

In Chapter 3 of part II of "A Course In Game Theory" by Ferguson (2020), it is mentioned that: *"If a finite game is invariant under a group, then there exist invariant optimal strategies for the players."* We can apply this principle on the extended NHG. Consider a group that contains all branch permutations. We can freely change the order of the branches without changing how the coins are connected. So by symmetry, the game is invariant under this group. By Ferguson's statement, this means that there exist invariant optimal strategies for both players.

In terms of our example, all variants of $(-2, -1, 0, 1, 2, 3, 4)$ have an equal probability of being chosen in the optimal strategy of player I. Let y denote the optimal strategy containing all the $\frac{B(B-1)}{2}(m-2) + B(b-m+2)$ different moves player 1 can make. Let $y_{-c[\alpha,\beta]} \in y$ denote the probability that player I will choose the a corner move from branch α to β with c coins in branch β and let $y_{s[\alpha]} \in y$ denote the probability that player I will choose straight move s in branch α. The invariant theorem tells us the probabilities are independent of the branch. In other words, we have:

$$\begin{aligned} &\forall c \in \{1, \dots, m-2\} : \forall \alpha, \alpha', \beta, \beta' \in \{1, \dots b\} : && y_{-c[\alpha,\beta]} = y_{-c[\alpha',\beta']} \\ &\forall s \in \{0, \dots, b-m+1\} : \forall \alpha, \alpha' \in \{1, \dots, b\} : && y_{s[\alpha]} = y_{s[\alpha']} \end{aligned}$$

From this point onward, we will speak of the *compact strategy* $\bar{y}$:

$$\bar{y} = (y_{-m-2}, y_{-m+3}, \dots, y_{-1}, y_0, \dots, y_{b-m+1})^T$$

Where each element y_{-c} in the first half represents all $\frac{B(B-1)}{2}$ corner strategies of type $-c$. And each element y_s in the second half represents all B straight strategies of type s. Keep in mind that the elements in a compact strategy $\bar{y}$ do not add up to 1. Since each element represents multiple strategies, the actual stochastic constraint is:

$$\sum_{c=1}^{m-2} \frac{B(B-1)}{2} y_{-c} + \sum_{s=0}^{b-m+2} B y_s = 1$$

Player II has a similar full strategy x and compact strategy $\bar{x}$:

$$\begin{aligned} &\bar{x} = (x_{-n-2}, x_{-n+3}, \dots, x_{-1}, x_0, \dots, x_{b-n+1})^T \\ &\sum_{c=1}^{n-2} \frac{B(B-1)}{2} x_{-c} + \sum_{s=0}^{b-n+2} B x_s = 1 \end{aligned}$$

4.3. Exact solution when m | b+1 and n < m

The exact solution of the NHG on a spider heavily depends on the parameters. Some combination of parameters force player I to choose lots of moves that include the root coin, while others actively avoid to. In this section we will focus on a specific subset of parameters. Namely, those under the conditions m | b+1 and n < m.

Theorem 2 (Exact Solution) *Let* $2b-1 \geq m > n \geq 1$ *and* $m \mid b+1$. *Then:*

$$V(m,n,B,b) = \frac{mn}{B(b+1)} \tag{16}$$

Moreover, a (compact) optimal strategy $\bar{y}$ *for player I is:*

$$\bar{y} = (0,0,\ldots,0,y_0,\ldots,y_{b-m+1})^T$$
$$y_i = \begin{cases} \frac{1}{B(b+1)} & \text{for } j = im, i = 0,1,\ldots,(M-1) \\ 0 & \text{otherwise} \end{cases}$$

Moreover, a (compact) optimal strategy $\bar{x}$ *for player II is:*

$$\bar{x} = (0,0,\ldots,0,x_0,\ldots,x_{b-n+1})^T$$
$$x_i = \begin{cases} \frac{m}{2B(b+1)} & \text{for } j = 1+im, i = 0,1,\ldots,(\frac{b+1}{m}-1) \\ \frac{m}{2B(b+1)} & \text{for } j = m-n+im, i = 0,1,\ldots,(\frac{b+1}{m}-1) \\ 0 & \text{otherwise} \end{cases}$$

4.3.1. Optimal strategy for player I

Player I needs to use a strategy that attains a payoff of at least $V(m,n,B,b)$ in all scenarios. We will show the strategy proposed Theorem 2 satisfies that condition. Notice that the compact strategy $\bar{y}$ from Theorem 2 doesn't contain any corner moves. The straight moves from this strategy start from the root coin and all multiples of m. Since $m \mid b+1$, those straight moves cover each coin from each branch exactly once. Moreover, since the root coin is contained in all branches, this coin is covered B times as much as the other coins. The coin coverage of this strategy is as in Figure 16. The best player II can do to counter this strategy is choosing a straight move that is fully contained in a branch. (without the root coin) The coverage of a branch coin is $\frac{m}{B(b+1)}$. In the worst case scenario, Player II will pick exactly n of these coins. Thus, the value of the game is at least $V(m,n,B,b) = \frac{mn}{B(b+1)}$.

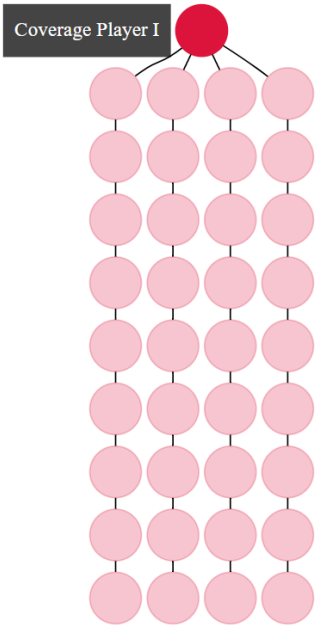

Figure 16: $G(m,n,B,b) = G(5,3,4,9)$

4.3.2. Optimal strategy for player II

To complete the proof, we need to show that player II has a strategy that attains a payoff of at most $V(m,n,B,b)$ in all possible match-ups. The strategy from Theorem 2 only contains moves without the branch coin. Moreover, the coin coverage of this strategies is divided into chunks of m, starting from the root coin. Each chunk is divided into 4 constant intervals with [coverage, length]:

$$[0,1],[\frac{m}{2B(b+1)},m-n-1],$$
$$[\frac{m}{B(b+1)},2n-m+1],[\frac{m}{2B(b+1)},m-n-1]$$

In other words, there are 3 types of coin coverages: 0, $\frac{m}{2B(b+1)}$ and $\frac{m}{B(b+1)}$. And they occur exactly $1, 2(m-n-1)$ and $2n-m+1$ times respectively in each chunk of m coins. Moreover, these occurrences remain the same for all possible moves of player I. Therefore, the expected payoff of the game equals the weighted sum of these coverage types:

$$\begin{aligned} V(m,n,B,b) &= 1\cdot 0 + 2(m-n+1)\frac{m}{2B(b+1)} \\ &\quad + (2n-m+1)\frac{m}{B(b+1)} \\ &= \frac{mn}{B(b+1)} \end{aligned}$$

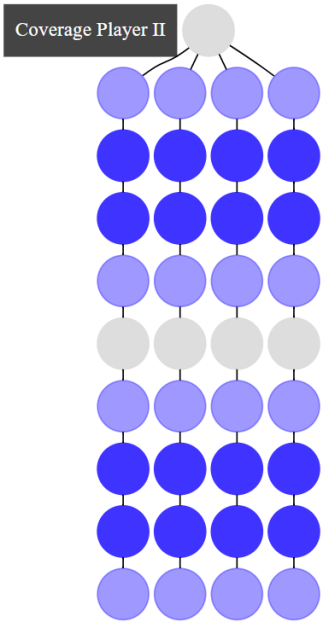

Figure 17: $G(m,n,B,b) = G(5,3,4,9)$

We have shown that player I reaches a minimum expected payoff of $\frac{mn}{B(b+1)}$ and player II reaches a maximum expected payoff of $\frac{mn}{B(b+1)}$. Therefore, the proposed optimal strategies from Theorem 2 are indeed optimal and the value equals $V(m,n,B,b) = \frac{mn}{B(b+1)}$.

4.4. Solution for large branchsizes

As mentioned in the previous section, a general solution heavily depends on the parameters and needs to be split up in many cases. That is why we are trying to find an estimate for the value instead of finding an exact solution for each individual case. We estimate the value by reducing a general game to a game with 2 branches and solving it as described in Theorem 1 from Section 3.1 with $p = 2b+1$. Then, we will scale the value of the game with 2 branches to the game with B branches. The idea behind this estimate is that the board size roughly increases with a factor $\frac{B}{2}$, so the value should scale accordingly. Let $p = 2b+1$ and let $V_o(m,n,p)$ denote the value as described in Theorem 1. Then, we use the following estimate for the NHG on a spider:

$$V(m,n,B,b) \approx \frac{2}{B}V_o(m,n,2b+1) \tag{17}$$

Another way to estimate a NHG on a spider is by completely disregarding the layout of the board. We can set $p = Bb + 1$ equal to the total number of coins on the board and calculate the value as if the coins were lined up like in the original NHG. This estimate works particularly well when b is a multiple of m. With this new approach, we can further improve our estimate to:

$$V(m, n, B, b) \approx \begin{cases} \frac{2}{B} V_o(m, n, 2b + 1) & \text{if } b \mid m \\ V_o(m, n, Bb + 1) & \text{otherwise} \end{cases} \tag{18}$$

We will now compare this estimate with the actual value calculated by the linear optimization program. Figure 18 shows the results of solving several games with different B and b and a fixed m and n. The dots indicate the actual value calculated with brute force. The lines indicate the estimate from (18). Moreover, the dots are painted blue whenever the actual value and the estimate differ by a truncation error of $< 10^{-6}$.

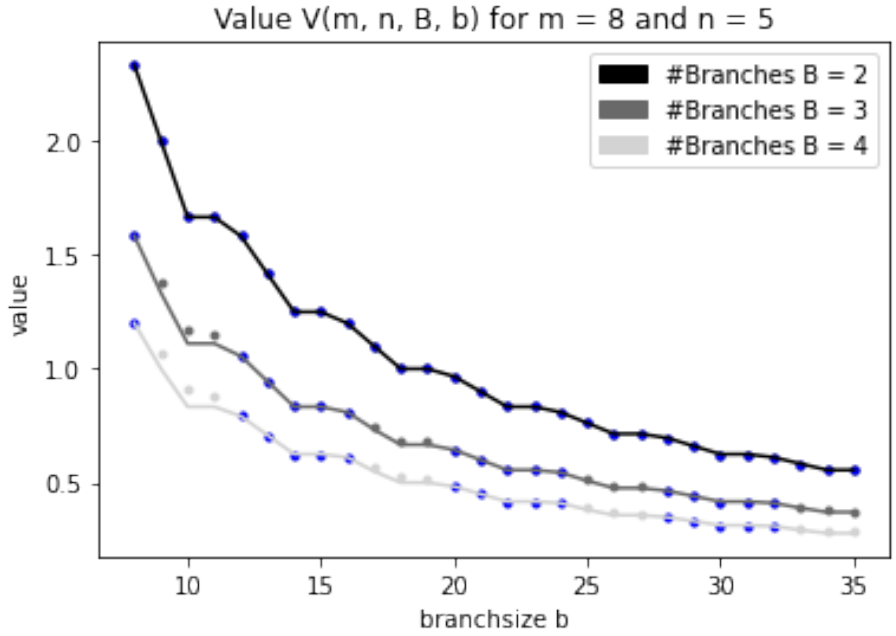

Figure 18: Plot of the value of several NHGs with fixed $m = 8$ and $n = 5$. Dots indicate the actual value calculated with linear optimization. Lines indicate the estimate from (18)

As the branchsize increase, the root coin becomes less significant and most of the game takes place within the branches. So it is to be expected that the ratio between the estimate and the actual value converges to 1.

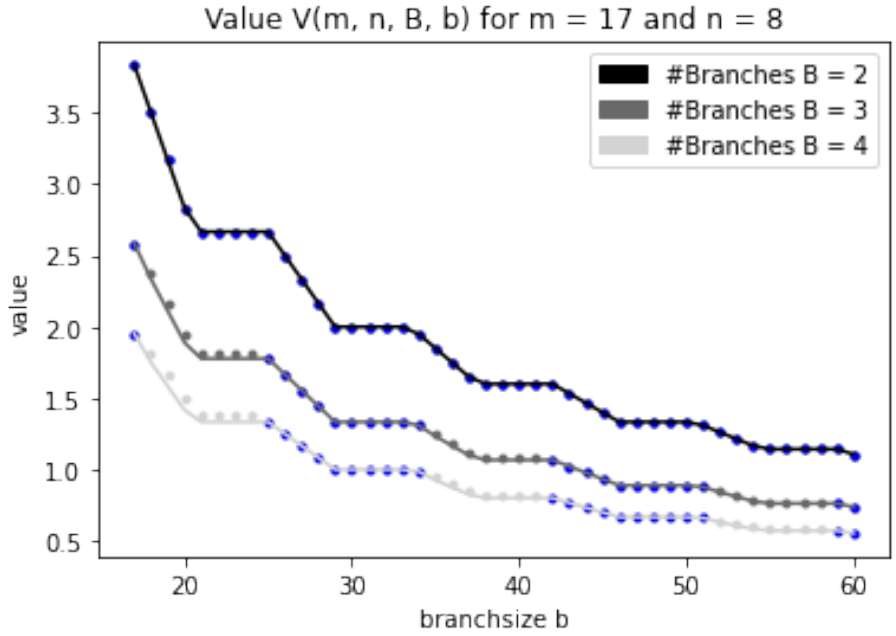

Figure 19: Plot of the value of several NHGs with fixed $m = 17$ and $n = 8$. Dots indicate the actual value calculated with linear optimization. Lines indicate the estimate from (18)

Numerical research suggests that the estimate from (18) equals the actual value of the game for at least $m - n + 1$ consecutive values of b in periods of length m. For some specific parameters, this may hold for an even larger interval. For example, in Figure 18, this interval has length $m - n + 2 = 5$ instead. But in general, it works for at least $m - n + 1$ consecutive values. For example, in Figure 19, the estimate is correct in intervals of length $m - n + 1 = 10$. In those cases, player II can play the game optimally without ever including the root coin in a move. Let us formally describe this conjecture.

Conjecture (The value of the NHG on a spider played within the branches) Let $B, m, n > 0$. Restrict b such that it can be written as $b = Mm + r$ with $M > 0$ and $n \leq r \leq m$. Then the estimate from (18) equals the value of the game.

$$V(m, n, B, b) = \begin{cases} \frac{2}{B} V_o(m, n, 2b + 1) & \text{if } b \mid m \\ V_o(m, n, Bb + 1) & \text{otherwise} \end{cases} \tag{19}$$

Finally, notice from Figure 18 and 19 that if the estimate from (18) doesn't meet the value, it is always an underestimate. That is because the estimate originates from the original NHG with the same amount of coins. In the extended game on a spider, we introduced a root coin with multiple edges. However, we didn't utilize these extra edges in the optimal strategies in the estimate. The actual value of the game can only be higher. Namely, when utilizing the special property of root coin to increase the payoff. Thus, our estimate is an underestimate.

4.5. Solution for small branchsizes

As the branchsize increases, the extended NHG becomes more and more like the original NHG. Estimate (18) has the greatest deviation from the original NHG whenever the branchsize is small. (around $m - 2 < b < m + n$) For small branchsizes, corner moves take up a considerable amount of the total strategy set. And since corner moves are not included in the original NHG, it makes sense that estimate (18) might deviate from the actual solution in those cases. That is why we are particularly interested in finding a better solution for small branchsizes.

Let us start by restricting the parameter space. Let $m, n, B, b \in \mathbb{N}$ be such that:

$$\begin{aligned} &m > 3 \\ &2 < n < m \\ &1 < B < 6 \\ &m < b < m + \frac{n}{2} \end{aligned} \tag{20}$$

We will search for a direct formula for the value of the game by looking at player II's perspective. Player II needs to hide the coins from player I to minimize the overlap. This can be done be spreading your moves across the board. We consider two types of moves for player II.

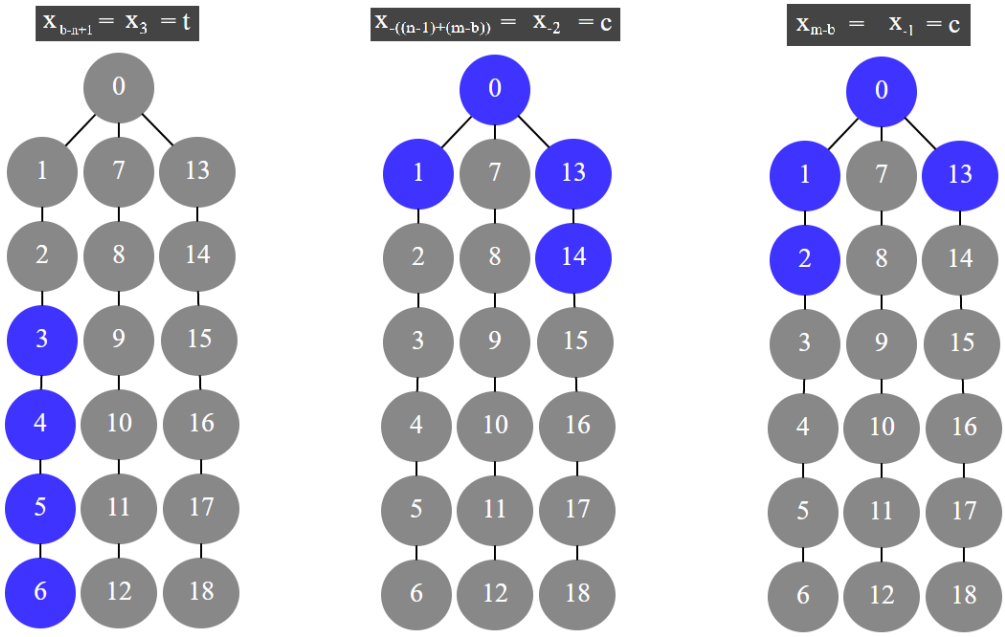

Figure 20: All non-zero moves in the strategy for player II. The left most image contains the tail move. (type 1) The other images represent the corner moves at maximum distance from the tail move. (type 2)

1. Move $b - n + 1$, the tail move.
   We say $x_{b-n+1} = t$. This is the probability that player II will pick a tail move.

2. Move $m - b$, the corner move.
   Note that $m - b < 0$, so this move is guaranteed to be a corner move. This corner move was chosen to be at maximum distance from the tail.
   We say $x_{m-b} = 2c$ whenever $m - b = -((n-1) + (m-b))$
   Otherwise, we have to share the probability with its symmetric counterpart.
   Then we say $x_{m-b} = x_{-((n-1)+(m-b))} = c$

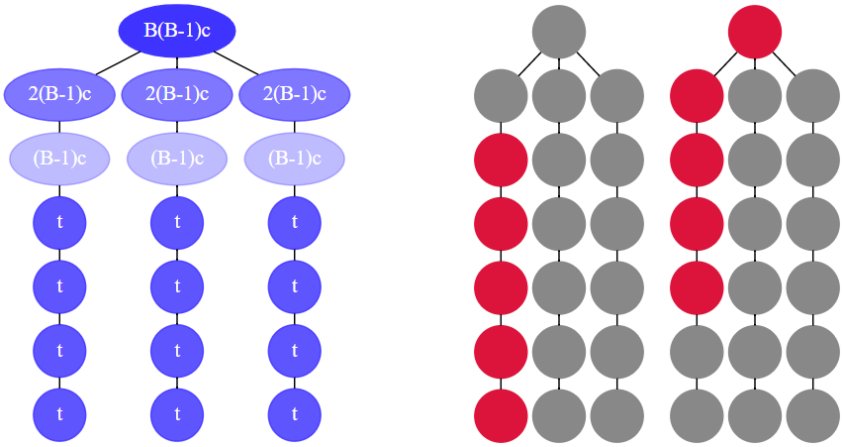

Figure 21: Coin coverage of the strategy for player II and two possible counter moves for player I. Later to be denoted as tail- and root-countermoves respectively.

We can combine the strategies from Figure 20 and calculate the coin coverage for each coin (see the leftmost image in Figure 21 for an example):

- The root coin is covered by all $B(B-1)$ variants of corner move $m - b$ and its symmetric counterpart. So its coverage is $B(B-1)c$.
- The 2nd layer of coins is covered by the corner moves $m-b$ and $-((n-1)+(m-b))$ that share a fixed branch. There are $(B-1)$ ways to pick a second branch in both cases. So their coverage is $2(B-1)c$.
- The 3rd layer of coins is covered by only one symmetric variant of the corner move. There are $(B-1)$ ways to pick a second branch respectively. So their coverage is $(B-1)c$.
- The 4th layer of coins is not covered by a move at all. These coins have 0 coverage. (Note that this layer has length 0 in the example from Figure 21, but this layer does exist for higher values of b.)
- The tail coins are only covered by a single tail move. So their coverage is t.
  (Beware: tail coins may overlap with the 3rd and 4th layer, however, this does not affect the upcoming calculations.)

We consider the 2 most extreme countermoves for player I. (see Figure 21)

1. The tail countermove. This move picks all coins from a branch in the 3rd, 4th and tail layer. The tail layer has length n. And the 3rd layer has length $(n-1-2(b-m))$. The payoff becomes: $V_{tail} = nt + (n-1-2(b-m)) \cdot (B-1)c$

2. The root countermove. This move picks the root coin (with coverage $B(B-1)c$) and the coins of the 2nd and 3rd layer in a single branch (they add up to a coverage of $(n-1) \cdot (B-1)c$). Moreover, whenever $m > b-n+1$, this move overlaps with $m-b+n-1$ tail coins for an additional coverage of $(m-b+n-1)t$.
The payoff becomes: $V_{root} = (B+n-1) \cdot (B-1)c + (m-b+n-1)t$

Recall from Section 4.2 that straight moves occur B times and corner moves occur $\frac{B(B-1)}{2}$ times. We can set up a stochastic constraint (probabilities must add up to 1) and rewrite it to express c in terms of t and B:

$$Bt + \frac{B(B-1)}{2}c + \frac{B(B-1)}{2}c = 1$$
$$c = \frac{1-tB}{B(B-1)}$$

We substitute this into our expressions of V_{tail} and V_{root}. Player II needs to minimize both counter strategies at the same time. (in fact, this is a small min-max problem) Therefore, the payoff lies at the intersection of $V_{tail}(t)$ and $V_{root}(t)$:

$$nt + \frac{(1-Bt)(n-1-2(b-m)}{B} = \frac{(1-Bt)(B+n-1)}{B} + (m+n-b-1)t$$

Rewriting this to isolate t gives us the weight player II assigns to a tail move.

$$t = \frac{2b-2m+B}{B(1+3b-3m+B)}$$

Finally, we insert this t into either $V_{tail}(t)$ or $V_{root}(t)$ and get the value of a game.

Conjecture (The value of the NHG on a spider for small branchsizes)
Let m, n, B, b be integers that satisfy the conditions from (20). Then:

$$V(m,n,B,b) = \frac{(4m+3n-3-2b)b + (-3n+3-2m)m + (B+1)n - 1}{(1+3b-3m+B)B} \tag{21}$$

We can compare this direct value with the value from the linear optimization program. For each value of $m > 3$ (the only unbounded constraint), we iterate through all values of n, B and b that satisfy (20) and calculate the value like proposed in (21). We find that the conjecture holds in at least 308512 cases (for all $3 < m < 99$). We won't check the validity of more cases due to the high time complexity of the optimization problem. But the formula is very likely to hold for all $m > 3$.

5. Final word

In this thesis, we checked the validity of the direct formula for the optimal strategies and the value of the original Number Hides Game like described in [2]. Moreover, we increased the complexity of the game by changing the layout of the board to a tree. With the help of the python scripts provided in Appendix A.5, we were able to numerically calculate optimal mixed strategies for both players and the corresponding value of the games. We are interested in direct formula's, since this numerical approach can be very time consuming for large parameters. We found direct solutions on a spider with an equal branchsize for several parameter spaces.

However, we were not able to find a general formula to solve a game on a general tree yet. Further research can be supported by the python code from Appendix A.5. This code generates figures like the ones that can be found in this thesis. Custom NHGs can easily be simulated by anyone with the following functions:

- `print_mixed_strategies(m, n, B, b)`
  Numerically solves $G(m, n, B, b)$. Displays all the non-zero weighted moves in the optimal strategies of both players as an image, prints the compact versions of the optimal strategies and prints the value of the game.

- `print_coverage(m, n, B, b)`
  Numerically solves $G(m, n, B, b)$. Creates a coverage image for both players, prints compact versions of the optimal strategies and prints the value of the game.

- `G = Graph(size)` Creates a graph G with `size` coins to play a NHG on.
  Repeatedly use `G.add_edge(i, j)` to customize the layout.
  Numerically calculate the payoff matrix, value and optimal strategies with
  `A, v, y, x = G.solve(m, n, edgeclasses_equal=False)`.
  Display strategies with `G.draw_edges(m, RED)` and `G.draw_edges(n, BLUE)`.

Although the code can be retrieved from the appendix, it is easier to download the code from `https://drive.google.com/file/d/1F2pHPoHiTI-UbVa3Vxe6zdfwhrOFbodK/view` instead.

References

[1] Ruckle, W. H. (1983), *Geometric games and their applications*, Research Notes in Mathematics No. 82, Pitman Advanced Publishing Program, Boston, London, Melbourne.

[2] Baston, V. J., Bostock F. A. and Ferguson T. S. (1989). *The number hides game.* Proceedings of the american mathematical society, 107(2), 437-447.

[3] Ruckle, W. H. (1982), *Technical Report #384*, Department of Mathematical Sciences, Clemson University.

[4] Ferguson, T. S. (2020), *A Course In Game Theory*, World Scientific.

Appendices

A. Python Code

A.1. Linear optimization

```
##### lp: definitions of the functions about linear programming (minmax and maxmin problems) #####

# A: (mxn) matrix
# m: positive integer
# n: positive integer
# x: probability COL-vector
# y: probability ROW-vector
# v: value = yAx

def int2str(i, minlength=8):
    '''converts int to a string with fixed minlength'''
    #this script is needed for alphabetical sorting
    s = str(i)
    if minlength > len(s):
        s = '0'*(minlength-len(s))+s
    return s

def lp_maxmin_solve(A):
    '''solves a linear problem of the form max_y(min_x(yAx)'''
    m, n = A.shape

    # from player I's perspective (player I maximizes y in yAx. y is a rowvector that picks the rows of A)
    maxmin_problem = pulp.LpProblem("MaxMin", pulp.LpMaximize)

    # v = value
    v = pulp.LpVariable("v")

    # y = probability vector
    y = [pulp.LpVariable("y"+int2str(i), lowBound=0) for i in range(m)]

    # INNERMIN-CONSTRAINT: (yA)_j >= v
    for j in range(n):
        maxmin_problem += (sum(A[i][j]*y[i] for i in range(m)) >= v)

    # SUM-CONSTRAINT: sum(y) = 1
    maxmin_problem += (sum(y) == 1)

    # OBJECTIVE: max v
    maxmin_problem += v

    #solve and return
    maxmin_problem.solve()
    v_result = maxmin_problem.variables()[0].varValue
    y_result = [maxmin_problem.variables()[i+1].varValue for i in range(m)]
    return v_result, y_result

def lp_minmax_solve(A):
    ''' solves a linear problem of the form min_x(max_y(yAx)) '''
    m, n = A.shape

    # from player II's perspective (player II minimizes x in yAx. x is a colvector that picks the columns of A)
    minmax_problem = pulp.LpProblem("MinMax", pulp.LpMinimize)

    # v = value
    v = pulp.LpVariable("v")

    # x = probability vector
    x = [pulp.LpVariable("x"+int2str(i), lowBound=0) for i in range(n)]

    # INNERMAX-CONSTRAINT: (Ax)_i <= v
    for i in range(m):
        minmax_problem += (sum(A[i][j]*x[j] for j in range(n)) <= v)

    # SUM-CONSTRAINT: sum(x) = 1
    minmax_problem += (sum(x) == 1)

    # OBJECTIVE: min v
    minmax_problem += v

    #solve and return
    minmax_problem.solve()
    v_result = minmax_problem.variables()[0].varValue
    x_result = [minmax_problem.variables()[i+1].varValue for i in range(n)]
```

```
    return v_result, x_result

def lp_solve(A, print_results=False):
    '''find the expected payoff and optimal strategies for a game with payoff matrix A'''
    v1, y = lp_maxmin_solve(A)
    v2, x = lp_minmax_solve(A)
    if v1 != v2:
        print("\nStrong duality doesn't hold. Hence, all of mathematics is inconsistent. 0=1. QED\n")
    elif print_results:
        print('### LP RESULTS ###\n')
        print(A, '\n')
        print('Value: ', str(v1), '\n')
        print('{:<36}{}'.format('Maximized strategy for the rows:', y))
        print('{:<36}{}\n'.format('Minimized strategy for the columns:', x))

    #return (value, [optimal strategy for P1, row, maximized], [optimal strategy for P2, col, minimized])
    return v1, y, x
```

A.2. Ferguson

```
##### FERGUSON: get exact solutions as described by BASTON, BOSTOCK and FERGUSON in [2] #####

def FERGUSON_q(m, p):
    '''get the exact mixed strategy for Player I, q(m, p) or Player II, q(m, p+m-n)'''

    q = (p-m+1)*[0]
    M = p//m
    x = p%m

    #case (3) in FERGUSON
    #equation (7) in this report
    if x == 0:
        for i in range(M):
            q[i*m] = M+1

    #case (4) in FERGUSON
    #equation (8) in this report
    else:
        #case (i)
        for i in range(M):
            q[i*m] = M-i

        #case (ii)
        for i in range(M):
            q[i*m+x] = i + 1

    return list(np.array(q)/M/(M+1))

def FERGUSON_get_payoff_value(i, m, j, n):
    ''' get the value of the number hides game G(m, n, p) at location (i, j)'''
    if m>=n:
        if i - n <= j <= i:
            return n - i + j
        if i <= j <= m + i - n:
            return n
        if m + i - n <= j <= m + i:
            return m + i - j
    else:
        if j - m <= i <= j:
            return m-j+i
        if j <= i <= n+j-m:
            return m
        if n + j - m <= i <= n + j:
            return n + j - i
    return 0

def FERGUSON_get_payoff_matrix(m, n, p):
    '''get the payoff matrix of the number hides game G(m, n, p)'''
    return np.array([[FERGUSON_get_payoff_value(i, m, j, n) for j in range(p-n+1)] for i in range(p-m+1)])

def FERGUSON_get_value(m, n, p):
    '''get the expected of the number hides game G(m, n, p), theorem 2'''
    if m>=n:
        M = p//m
        x = p%m
        # (!) why is 0<=x<=m in the article instead of 0<=x<m?
        # this makes the decomposition p = Mm+x not unique
        if 0<=x<n:
            # linear interpolation
            return (n*(M+1)-x)/M/(M+1)
        else:
            # proposition 3
```

```
            return n/(M+1)
    else:
        N = p//n
        x = p%n
        if 0<=x<=n-m:
            return m/N
        else:
            return (m*N+n-x)/N/(N+1)

def FERGUSON_solve(m, n, p, print_results=False):
    '''find the expected payoff and optimal strategies for the game G(m, n, p)'''
    if m>=n:
        #player I, chooses the rows of A
        y = FERGUSON_q(m, p)
        #player II, chooses the columns of A
        x = FERGUSON_q(m, p+m-n)
    else:
        # corollary 3
        y = (n-m)*[0] + FERGUSON_q(n, p-n+m) + (n-m)*[0]
        x = FERGUSON_q(n, p)

        # (!) what if p-n+m < n? then q(n, p-n+m) has length 0.
        # so y = (0, q(n, p-n+m), 0) = 0. but then its sum != 1.
        # this means y is not a valid strategy.
        # use y = (0,...0, 1, 0,... 0) of length (p-m+1) with a 1 at (p-m+1)//2.
        # this case was discussed in proposition 1 by the authors
        # unfortunately, they forgot to mention this case in corollary 3
        if p-n+m < n:
            # proposition 1
            y = (p-m+1)*[0]
            y[(p-m+1)//2] = 1

    #get the expected payoff, or value, of the game
    value = FERGUSON_get_value(m, n, p)

    if print_results:
        print('### FERGUSON RESULTS ###\n')
        #print(A, '\n')
        print('Value: ', str(value), '\n')
        print('{:<36}{}'.format('Maximized strategy for the rows:', y))
        print('{:<36}{}\n'.format('Minimized strategy for the columns:', x))

    return value, y, x

m, n, p = 6, 3, 6
A = FERGUSON_get_payoff_matrix(m, n, p)
value, y, x = FERGUSON_solve(m, n, p)
```

A.3. Unittests for linear optimization and ferguson (+manual test)

```
###############################################################################
###############################################################################
#########################      U N I T T E S T S      #########################
#########################   for the lp and ferguson   #########################
###############################################################################
###############################################################################

class Test_lp(unittest.TestCase):

    def test_rps(self):
        '''test a game of Rock, Paper, Scissors'''
        A = np.array([[0, -1, 1], [1, 0, -1], [-1, 1, 0]])
        value, y, x = lp_solve(A, print_results=False)
        self.assertAlmostEqual(value, y@A@x, places=4)
        self.assertEqual(value, 0)

    def test_fixed(self):
        '''predefined test with manually calculated value 4'''
        A = np.array([[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]])
        value, y, x = lp_solve(A, print_results=False)
        self.assertAlmostEqual(value, y@A@x, places=4)
        self.assertEqual(value, 4)

    def test_lp_with_int2str(self):
        '''test that failed in the past, this was fixed by replacing str() with int2str()'''
        A = np.array([[4, 4, 1, 4, 2, 4, 4, 3, 2, 4, 3]])
        value, y, x = lp_solve(A, print_results=False)
        self.assertAlmostEqual(value, y@A@x, places=4)

    def test_random(self):
        '''check if value corresponds with the yAx calculation with random A'''
        A = np.random.rand(9,5)
```

```
        value, y, x = lp_solve(A, print_results=False)
        self.assertAlmostEqual(value, y@A@x, places=4)

class Test_FERGUSON(unittest.TestCase):

    def test_examplechapter1(self):
        '''example from page 438 in fergusons article'''
        A = FERGUSON_get_payoff_matrix(5, 7, 17)
        value, y, x = FERGUSON_solve(5, 7, 17)
        self.assertAlmostEqual(value, 7/3)
        self.assertAlmostEqual(value, y@A@x)
        self.assertAlmostEqual(y[2], 1/3)  #{3, 4, 5, 6, 7}
        self.assertAlmostEqual(y[10], 1/3) #{11, 12, 13, 14, 15}
        self.assertAlmostEqual(y[3], 1/6)  #{4, 5, 6, 7, 8}
        self.assertAlmostEqual(y[9], 1/6)  #{10, 11, 12, 13, 14}
        self.assertAlmostEqual(x[0], 1/3)  #{1, 2, 3, 4, 5, 6, 7}
        self.assertAlmostEqual(x[10], 1/3) #{11, 12, 13, 14, 15, 16, 17}
        self.assertAlmostEqual(x[3], 1/6)  #{5, 6, 7, 8, 9, 10, 11}       (!)
        self.assertAlmostEqual(x[7], 1/6)  #{7, 8, 9, 10, 11, 12, 13}     (!)

        # (!) denotes a diviation of the vector q used on page 438 and 442
        # both lead to an optimal solution, but only the the vector from page 442
        # is derived from the definition of q proposed by the authors.
        # {5, 6, 7, 8, 9, 10, 11} does not correspond with x[3], but with x[4]
        # {7, 8, 9, 10, 11, 12, 13} does not correspond with x[7], but with x[6]

    def test_fixed1(self):
        '''test general case from the article on specific values of m, n, p'''
        m, n, p = 32, 12, 44
        A = FERGUSON_get_payoff_matrix(m, n, p)
        value, y, x = FERGUSON_solve(m, n, p)
        self.assertAlmostEqual(value, y@A@x)

    def test_fixed2(self):
        '''test general case from the article on specific values of m, n, p'''
        m, n, p = 11, 49, 49
        A = FERGUSON_get_payoff_matrix(m, n, p)
        value, y, x = FERGUSON_solve(m, n, p)
        self.assertAlmostEqual(value, y@A@x)

    def test_fixed3(self):
        '''test general case from the article on specific values of m, n, p'''
        m, n, p = 6, 3, 6
        A = FERGUSON_get_payoff_matrix(m, n, p)
        value, y, x = FERGUSON_solve(m, n, p)
        self.assertAlmostEqual(value, y@A@x)

    def test_random(self):
        '''test general case from the article on 100 random values of m, n, p'''
        for iteration in range(100):
            m, n= random.randint(1, 100), random.randint(1, 100)
            p = random.randint(max(m, n), 100)
            A = FERGUSON_get_payoff_matrix(m, n, p)
            value, y, x = FERGUSON_solve(m, n, p)
            self.assertAlmostEqual(value, y@A@x)

class Test_lp_and_FERGUSON(unittest.TestCase):

    def test_random_small(self):
        '''test if the lp and ferguson attain the same value on 5 random small games'''
        for iteration in range(5):
            m, n= random.randint(1, 15), random.randint(1, 15)
            p = random.randint(max(m, n), 20)
            A = FERGUSON_get_payoff_matrix(m, n, p)
            lp_value, lp_y, lp_x = lp_solve(A, print_results=False)
            exact_value, exact_y, exact_x = FERGUSON_solve(m, n, p)
            self.assertAlmostEqual(lp_value, exact_value, places=5)

    def test_fixed_big(self):
        '''test if the lp and ferguson attain the same value on 1 fixed big game'''
        m, n, p = 12, 31, 37
        A = FERGUSON_get_payoff_matrix(m, n, p)
        lp_value, lp_y, lp_x = lp_solve(A, print_results=False)
        exact_value, exact_y, exact_x = FERGUSON_solve(m, n, p)
        self.assertAlmostEqual(lp_value, exact_value, places=5)

    def test_random_big(self):
        '''test if the lp and ferguson attain the same value on 1 random big game'''
        m, n= random.randint(1, 50), random.randint(1, 50)
        p = random.randint(max(m, n), 50)
        A = FERGUSON_get_payoff_matrix(m, n, p)
        lp_value, lp_y, lp_x = lp_solve(A, print_results=False)
        exact_value, exact_y, exact_x = FERGUSON_solve(m, n, p)
        self.assertAlmostEqual(lp_value, exact_value, places=5)
```

```
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)

##############################################################################
##########################                          ##########################
##########################    U N I T T E S T S     ##########################
##########################    for the hypergraphs   ##########################
##########################                          ##########################
##############################################################################

class Test_graph(unittest.TestCase):
    def test_xtree(self):
        G=Graph(10, branches=3)
        lst = [[0, 1, 2], [0, 4, 5], [0, 7, 8], [1, 0, 4], [1, 0, 7], [1, 2, 3], [4, 0, 7], [4, 5, 6], [7, 8, 9]]
        self.assertListEqual(G.get_edges(3), lst)
    def test_path(self):
        G=Graph(10, branches=1)
        lst = [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7], [7, 8], [8, 9]]
        self.assertListEqual(G.get_edges(), lst)

class Test_lp_on_graph(unittest.TestCase):
    def test_sort_fixed_small(self):
        m, n = 6, 4
        G = Graph(11, branches=3)
        A, v, y, x = G.solve(m, n, print_results=False)
        self.assertAlmostEqual(y@A@x, v, places=4)

    def test_sort_fixed_big(self):
        m, n = 9, 5
        G = Graph(20, branches=5)
        A, v, y, x = G.solve(m, n, print_results=False)
        self.assertAlmostEqual(y@A@x, v, places=4)

    def test_compare_ferguson(self):
        m, n, p = 18, 15, 33
        ferguson_value = 7.5
        G = Graph(size=p, branches=1)
        value = G.get_value(m, n)
        self.assertAlmostEqual(value, ferguson_value, places=4)

class Test_representaties(unittest.TestCase):
    def test_get_edgeclass(self):
        self.assertEqual(get_edgeclass([1, 0, 69, 70], 68), -1)
        self.assertEqual(get_edgeclass([1, 2, 0, 69], 68), -1)
        self.assertEqual(get_edgeclass([0, 1, 2, 3, 4, 5], 112), 0)
        self.assertEqual(get_edgeclass([2, 1, 0], 112), 0)
        self.assertEqual(get_edgeclass([6, 5, 4, 3, 2, 1, 0, 12, 13, 14, 15, 16, 17], 11), -6)
        self.assertEqual(get_edgeclass([6, 5, 4, 3, 2, 1, 0, 12, 13, 14, 15], 11), -4)
        self.assertEqual(get_edgeclass([2, 1, 0, 12, 13, 14, 15], 11), -2)

class Test_compact_full(unittest.TestCase):
    def test_convertion_on_conjecture_1(self):
        for iteration in range(10):
            M = randint(2, 6)
            m = randint(6, 12)
            n = randint(2, m-1)
            B, b = 3, M*m-1
            full_y = get_exact_strategy(m, n, B, b, player=1)
            compact_y = full_to_compact(full_y, m, B, b)
            reconstructed_y = compact_to_full(compact_y, m, B, b)
            self.assertEqual(len(full_y), len(reconstructed_y))
            for i in range(len(full_y)):
                self.assertAlmostEqual(full_y[i], reconstructed_y[i], places=4)

            full_x = get_exact_strategy(m, n, B, b, player=2)
            compact_x = full_to_compact(full_x, n, B, b)
            reconstructed_x = compact_to_full(compact_x, n, B, b)
            self.assertEqual(len(full_x), len(reconstructed_x))
            for i in range(len(full_x)):
                self.assertAlmostEqual(full_x[i], reconstructed_x[i], places=4)

    def test_convertion_on_lp_fixed(self):
        m = 7
        n = 2
        B = 3
        b = 11
        G = Graph(size=1+B*b, branches=B)
        A, v, full_y, full_x = G.solve(m, n, edgeclasses_equal=True, print_results=False)

        compact_y = full_to_compact(full_y, m, B, b)
        compact_x = full_to_compact(full_x, n, B, b)
        reconstructed_y = compact_to_full(compact_y, m, B, b)
        reconstructed_x = compact_to_full(compact_x, n, B, b)
```

```
        self.assertEqual(len(full_y), len(reconstructed_y))
        for i in range(len(full_y)):
            self.assertAlmostEqual(full_y[i], reconstructed_y[i], places=4)

        self.assertEqual(len(full_x), len(reconstructed_x))
        for i in range(len(full_x)):
            self.assertAlmostEqual(full_x[i], reconstructed_x[i], places=4)

    def test_convertion_on_lp_random(self):
        m = randint(6, 10)
        n = randint(4, m-1)
        B = randint(1, 4)
        b = randint(10, 20)
        #print('(m, n, B, b)=({}, {}, {}, {})'.format(m, n, B, b))
        G = Graph(size=1+B*b, branches=B)
        A, v, full_y, full_x = G.solve(m, n, edgeclasses_equal=True, print_results=False)

        compact_y = full_to_compact(full_y, m, B, b)
        compact_x = full_to_compact(full_x, n, B, b)
        reconstructed_y = compact_to_full(compact_y, m, B, b)
        reconstructed_x = compact_to_full(compact_x, n, B, b)

        self.assertEqual(len(full_y), len(reconstructed_y))
        for i in range(len(full_y)):
            self.assertAlmostEqual(full_y[i], reconstructed_y[i], places=4)

        self.assertEqual(len(full_x), len(reconstructed_x))
        for i in range(len(full_x)):
            self.assertAlmostEqual(full_x[i], reconstructed_x[i], places=4)

class Test_lp_vs_exact(unittest.TestCase):
    def test_conjecture_1(self):
        for iteration in range(1):
            M = randint(2, 6)
            m = randint(6, 12)
            n = randint(2, m-1)
            B, b = 3, M*m-1
            G = Graph(size=1+B*b, branches=B)
            A, v, y, x = G.solve(m, n, edgeclasses_equal=True)
            new_y = get_exact_strategy(m, n, B, b, player=1)
            new_x = get_exact_strategy(m, n, B, b, player=2)
            #print('(M, m, n)=({},{},{})'.format(M, m, n))
            self.assertAlmostEqual(v, y@A@x, places=4)
            self.assertAlmostEqual(v, new_y@A@x, places=4)
            self.assertAlmostEqual(v, y@A@new_x, places=4)
            self.assertAlmostEqual(v, new_y@A@new_x, places=4)

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)

##############################################################################
##########################                          ##########################
##########################    MANUAL VISUALISATION  ##########################
##########################    for the code about lp ##########################
##########################    and ferguson          ##########################
##########################                          ##########################
##############################################################################

#some random other examples
for iteration in range(5):
    m, n= random.randint(1, 10), random.randint(1, 10)
    p = random.randint(max(m, n), 30)
    if iteration == 0:
        #overwrite the random m, n, p by a fixed set of parameters for the first iteration
        #this is the example from chapter 3.1 from this report
        m, n, p = 5, 7, 17
    header_str, symbol_offset, space_offset = 'G(m, n, p) = G({},{},{})'.format(m, n, p), 10, 3
    print('\n'+'/'*(len(header_str)+(symbol_offset+space_offset)*2))
    print('/'*(len(header_str)+(symbol_offset+space_offset)*2))
    print('/'*symbol_offset+' '*(space_offset*2+len(header_str)) + '/'*symbol_offset)
    print('/'*symbol_offset + ' '*space_offset + header_str + ' '*space_offset + '/'*symbol_offset)
    print('/'*symbol_offset+' '*(space_offset*2+len(header_str)) + '/'*symbol_offset)
    print('/'*(len(header_str)+(symbol_offset+space_offset)*2))
    print('/'*(len(header_str)+(symbol_offset+space_offset)*2)+'\n')
    A = FERGUSON_get_payoff_matrix(m, n, p)
    print('### PAYOFF MATRIX ###\n')
    print(A, '\n')
    lp_value, lp_y, lp_x = lp_solve(A, print_results=True)
    exact_value, exact_y, exact_x = FERGUSON_solve(m, n, p, print_results=True)

### output ###
////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
```

```
//////////                                                  //////////
//////////     This part corresponds to                     //////////
//////////     the example from section 3.1                 //////////
//////////                                                  //////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////
////////////////////////////////////////////
////////////////////////////////////////
//////////                              //////////
//////////    G(m, n, p) = G(7,5,15)    //////////
//////////                              //////////
////////////////////////////////////////
////////////////////////////////////////////
////////////////////////////////////////////////
```

PAYOFF MATRIX

```
[[5 5 5 4 3 2 1 0 0 0 0]
 [4 5 5 5 4 3 2 1 0 0 0]
 [3 4 5 5 5 4 3 2 1 0 0]
 [2 3 4 5 5 5 4 3 2 1 0]
 [1 2 3 4 5 5 5 4 3 2 1]
 [0 1 2 3 4 5 5 5 4 3 2]
 [0 0 1 2 3 4 5 5 5 4 3]
 [0 0 0 1 2 3 4 5 5 5 4]
 [0 0 0 0 1 2 3 4 5 5 5]]
```

LP RESULTS

```
Value:  2.33333

Maximized strategy for the rows:    [0.333333, 0.166667, 0.0, 0.0, 0.0, 0.0, 0.0, 0.166667, 0.333333]
Minimized strategy for the columns: [0.333333, 0.0, 0.333333, 0.0, 0.166667, 0.0, 0.0, 0.0, 0.166667, 0.0, 0.0]
```

FERGUSON RESULTS

```
Value:  2.3333333333333335

Maximized strategy for the rows:    [0.3333333333333333, 0.16666666666666666, 0.0, 0.0,
                                     0.0, 0.0, 0.0, 0.16666666666666666, 0.3333333333333333]
Minimized strategy for the columns: [0.3333333333333333, 0.0, 0.0, 0.16666666666666666, 0.0,
                                     0.0, 0.0, 0.16666666666666666, 0.0, 0.0, 0.3333333333333333]
```

```
////////////////////////////////////////////////
////////////////////////////////////////////////
//////////                            //////////
//////////   G(m, n, p) = G(5,7,17)   //////////
//////////                            //////////
////////////////////////////////////////////////
////////////////////////////////////////////////
```

PAYOFF MATRIX

```
[[5 4 3 2 1 0 0 0 0 0 0]
 [5 5 4 3 2 1 0 0 0 0 0]
 [5 5 5 4 3 2 1 0 0 0 0]
 [4 5 5 5 4 3 2 1 0 0 0]
 [3 4 5 5 5 4 3 2 1 0 0]
 [2 3 4 5 5 5 4 3 2 1 0]
 [1 2 3 4 5 5 5 4 3 2 1]
 [0 1 2 3 4 5 5 5 4 3 2]
 [0 0 1 2 3 4 5 5 5 4 3]
 [0 0 0 1 2 3 4 5 5 5 4]
 [0 0 0 0 1 2 3 4 5 5 5]
 [0 0 0 0 0 1 2 3 4 5 5]
 [0 0 0 0 0 0 1 2 3 4 5]]
```

LP RESULTS

```
Value:  2.33333

Maximized strategy for the rows:    [0.0, 0.0, 0.333333, 0.0, 0.0, 0.333333, 0.166667,
                                     0.0, 0.0, 0.0, 0.0, 0.0, 0.166667]
Minimized strategy for the columns: [0.333333, 0.0, 0.333333, 0.0, 0.166667, 0.0, 0.0,
                                     0.0, 0.166667, 0.0, 0.0]
```

FERGUSON RESULTS

```
Value:  2.3333333333333335

Maximized strategy for the rows:    [0, 0, 0.3333333333333333, 0.16666666666666666, 0.0, 0.0,
                                     0.0, 0.0, 0.16666666666666666, 0.3333333333333333, 0, 0]
Minimized strategy for the columns: [0.3333333333333333, 0.0, 0.0, 0.16666666666666666, 0.0, 0.0,
                                     0.0, 0.16666666666666666, 0.0, 0.0, 0.3333333333333333]
```

```
//////////////////////////////////////////////
//////////////////////////////////////////////
//////////                          //////////
//////////    G(m, n, p) = G(7,8,21)//////////
//////////                          //////////
//////////////////////////////////////////////
//////////////////////////////////////////////
```

PAYOFF MATRIX

```
[[7 6 5 4 3 2 1 0 0 0 0 0 0 0]
 [7 7 6 5 4 3 2 1 0 0 0 0 0 0]
 [6 7 7 6 5 4 3 2 1 0 0 0 0 0]
 [5 6 7 7 6 5 4 3 2 1 0 0 0 0]
 [4 5 6 7 7 6 5 4 3 2 1 0 0 0]
 [3 4 5 6 7 7 6 5 4 3 2 1 0 0]
 [2 3 4 5 6 7 7 6 5 4 3 2 1 0]
 [1 2 3 4 5 6 7 7 6 5 4 3 2 1]
 [0 1 2 3 4 5 6 7 7 6 5 4 3 2]
 [0 0 1 2 3 4 5 6 7 7 6 5 4 3]
 [0 0 0 1 2 3 4 5 6 7 7 6 5 4]
 [0 0 0 0 1 2 3 4 5 6 7 7 6 5]
 [0 0 0 0 0 1 2 3 4 5 6 7 7 6]
 [0 0 0 0 0 0 1 2 3 4 5 6 7 7]
 [0 0 0 0 0 0 0 1 2 3 4 5 6 7]]
```

LP RESULTS

```
Value:  2.83333

Maximized strategy for the rows:    [0.0, 0.0, 0.222222, 0.0, 0.0, 0.277778, 0.0,
                                     0.444444, 0.0, 0.0, 0.0555556, 0.0, 0.0, 0.0, 0.0]
Minimized strategy for the columns: [0.333333, 0.0, 0.0, 0.0, 0.0, 0.333333, 0.0,
                                     0.0, 0.0, 0.166667, 0.0, 0.0, 0.166667, 0.0]
```

FERGUSON RESULTS

```
Value:  2.8333333333333335

Maximized strategy for the rows:    [0, 0.3333333333333333, 0.0, 0.0, 0.0, 0.16666666666666666, 0.0,
                                     0.0, 0.0, 0.16666666666666666, 0.0, 0.0, 0.0, 0.3333333333333333, 0]
Minimized strategy for the columns: [0.3333333333333333, 0.0, 0.0, 0.0, 0.0, 0.16666666666666666,
                                    0.0, 0.0, 0.16666666666666666, 0.0, 0.0, 0.0, 0.0, 0.3333333333333333]
```

```
//////////////////////////////////////////////
//////////////////////////////////////////////
//////////                          //////////
//////////    G(m, n, p) = G(10,1,10)//////////
//////////                          //////////
//////////////////////////////////////////////
//////////////////////////////////////////////
```

PAYOFF MATRIX

```
[[1 1 1 1 1 1 1 1 1 1]]
```

LP RESULTS

```
Value:  1.0

Maximized strategy for the rows:    [1.0]
Minimized strategy for the columns: [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

FERGUSON RESULTS

```
Value:  1.0

Maximized strategy for the rows:    [1.0]
Minimized strategy for the columns: [0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.5]
```

```
//////////////////////////////////////////////
//////////////////////////////////////////////
//////////                          //////////
//////////    G(m, n, p) = G(5,9,12)//////////
//////////                          //////////
//////////////////////////////////////////////
//////////////////////////////////////////////
```

PAYOFF MATRIX

```
[[5 4 3 2]
```

```
 [5 5 4 3]
 [5 5 5 4]
 [5 5 5 5]
 [5 5 5 5]
 [4 5 5 5]
 [3 4 5 5]
 [2 3 4 5]]

### LP RESULTS ###

Value:  5.0

Maximized strategy for the rows:    [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]
Minimized strategy for the columns: [1.0, 0.0, 0.0, 0.0]

### FERGUSON RESULTS ###

Value:  5.0

Maximized strategy for the rows:    [0, 0, 0, 0, 1, 0, 0, 0]
Minimized strategy for the columns: [0.5, 0.0, 0.0, 0.5]


//////////////////////////////////////////////////
//////////////////////////////////////////////////
//////////                              //////////
//////////   G(m, n, p) = G(10,4,28)    //////////
//////////                              //////////
//////////////////////////////////////////////////
//////////////////////////////////////////////////

### PAYOFF MATRIX ###

[[4 4 4 4 4 4 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [3 4 4 4 4 4 4 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [2 3 4 4 4 4 4 4 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 2 3 4 4 4 4 4 4 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 2 3 4 4 4 4 4 4 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 2 3 4 4 4 4 4 4 4 3 2 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 2 3 4 4 4 4 4 4 4 3 2 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 2 3 4 4 4 4 4 4 4 3 2 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 2 3 4 4 4 4 4 4 4 3 2 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 2 3 4 4 4 4 4 4 4 3 2 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 2 3 4 4 4 4 4 4 4 3 2 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 2 3 4 4 4 4 4 4 4 3 2 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 2 3 4 4 4 4 4 4 4 3 2 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 2 3 4 4 4 4 4 4 4 3 2 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 4 4 4 4 4 4 3 2 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 4 4 4 4 4 4 3 2 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 4 4 4 4 4 4 3 2]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 4 4 4 4 4 4 3]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 4 4 4 4 4 4]]

### LP RESULTS ###

Value:  1.33333

Maximized strategy for the rows:    [0.333333, 0.0, 0.333333, 0.0, 0.0, 0.0,
                                     0.0, 0.0, 0.0, 0.0, 0.333333, 0.0, 0.0,
                                     0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
Minimized strategy for the columns: [0.333333, 0.0, 0.333333, 0.0, 0.0, 0.0, 0.0, 0.0,
                                     0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.333333,
                                     0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

### FERGUSON RESULTS ###

Value:  1.3333333333333333

Maximized strategy for the rows:    [0.3333333333333333, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                                     0.16666666666666666, 0.0, 0.16666666666666666, 0.0, 0.0,
                                     0.0, 0.0, 0.0, 0.0, 0.0, 0.3333333333333333]
Minimized strategy for the columns: [0.25, 0.0, 0.0, 0.0, 0.08333333333333333, 0.0, 0.0, 0.0,
                                     0.0, 0.0, 0.16666666666666666, 0.0, 0.0, 0.0, 0.16666666666666666,
                                     0.0, 0.0, 0.0, 0.0, 0.0, 0.08333333333333333, 0.0, 0.0, 0.0, 0.25]
```

A.4. Visualization of the NHG

```
#######################################################################
# import libraries to do calculations, draw graphs and print tables #
#######################################################################

from IPython.display import display
```

```
import graphviz
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import numpy as np
import math as math
import re

def get_mixed_strategy(m, p):
    ''' return q(m, p) * M * (M+1) from FERGUSON '''
    # note that the mixed strategy for player 2 is the same, except p <- p+m-n
    q = (p-m+1)*[0]
    M = p//m
    x = p%m
    #case (3)
    if x == 0:
        for i in range(M):
            q[i*m] = M+1
    #case (4)
    else:
        #case (i)
        for i in range(M):
            q[i*m] = M-i
        #case (ii)
        for i in range(M):
            q[i*m+x] = i + 1
    return q


##########################################################
# Used to display mixed strategies in the original NHG #
##########################################################

def print_strategy(index, m, p, weight=-1, totalweight=-1, display_zero_weight = False, select_color="#DC143C"):
    '''prints the strategy #index'''
    if display_zero_weight == True or weight!=0:
        #define the graph
        G = graphviz.Graph(name='my_graph.gv')
        G.graph_attr.update(nodesep="0.05")
        G.node_attr.update(width="0.5")
        G.node_attr.update(fontcolor='white')

        #(optional) display the mixed strategy weights
        gray = '#888888'
        darkgray = '#444444'
        if weight>=0:
            G.node_attr.update(color="#e6bbb2")
            with G.subgraph() as S:
                S.attr(rank='same')
                nodename = str(weight)
                if totalweight >= 0:
                    nodename += '/'+str(totalweight)
                else:
                    nodename += '_'
                S.node(nodename, shape='box', style='filled', color=darkgray)
                S.node('', color='white', style='filled')

        #add all the nodes to the graph
        for i in range(1, p+1):
            with G.subgraph() as S:
                #make the path horizontal
                S.attr(rank='same')

                #create nodes and color according to parameters index, m
                c=gray
                if index<i<=index+m:
                    c=select_color
                S.node(str(i), color=c, style='filled')

                #add simple edges
                if i>1:
                    S.edge(str(i-1), str(i))
        display(G)

def print_strategy_set(m, n, p, player=1):
    '''prints the complete strategy set'''
    if player == 1:
        select_color = "#DC143C"
        weights = get_mixed_strategy(m, p)
    else:
        select_color = "#3F33FF"
        weights = get_mixed_strategy(m, p+m-n)
        m=n

    #plot the strategy set
```

```
    for index in range(p-m+1):
        print_strategy(index, m, p, weight=weights[index], totalweight=sum(weights), select_color=select_color)

def print_mixed_strategies(m, n, p):
    '''prints the mixed strategies for G(m, n, p) for player 1 and 2'''
    print('Mixed Strategy player 1:')
    print_strategy_set(m, n, p, player=1)
    print('Mixed Strategy player 2:')
    print_strategy_set(m, n, p, player=2)


####################################
# EXAMPLE (Figure 4 in the report) #
####################################

print_mixed_strategies(7,5,15)
```

A.5. The NHG on a Tree

```
#############################################################################################
#############################################################################################
#################                                                  ##########################
#################    Definition of a general (hyper)graph          ##########################
#################                                                  ##########################
#############################################################################################
#############################################################################################

LIGHTGRAY = '#dddddd'
DARKGRAY = '#444444'
GRAY = '#888888'
RED = "#DC143C"
BLUE = "#3F33FF"

def float2hex(x: float, lowerbound=0, upperbound=1):
    '''maps float x in [lowerbound, upperbound] to a hex value in ['00', 'ff']'''
    x = sorted([lowerbound, x, upperbound])[1]
    x -= lowerbound
    x *= 255/(upperbound-lowerbound)
    return hex(int(x))[2:]

def reorder(lst, weights):
    '''return lst in descending order based on the weights'''
    order = [i for _, i in sorted(zip(weights, range(len(lst))), reverse=True)]
    return [lst[i] for i in order]

class Node:
    def __init__(self, ID: int):
        '''set up a node with given ID'''
        self.ID = ID
        self.neighbors = []

class Graph:
    #specific order to print the edges based on weigths y and x. empty = no specific order.
    y = []
    x = []
    m = 0
    n = 0

    def __init__(self, size: int, branches=0):
        '''set up a graph with given size and layout'''
        #set nodes
        self.size = size
        self.branches = branches
        self.nodes = [Node(i) for i in range(size)]

        #set edges
        if branches > 0:
            branchsize = (size-1)//branches
            remainder = (size-1)%branches
            index = 1
            for branch in range(branches):
                self.add_edge(0, index)
                index += 1
                for i in range(1, branchsize + (branch<remainder)):
                    self.add_edge(index-1, index)
                    index += 1

    def add_edge(self, i, j):
        '''add an edge between node i and j'''
        self.nodes[j].neighbors.append(self.nodes[i])
        self.nodes[i].neighbors.append(self.nodes[j])
```

```
def remove_edge(self, i, j):
    '''remove an edge between node i and j'''
    if self.nodes[i] in self.nodes[j].neighbors:
        self.nodes[j].neighbors.remove(self.nodes[i])
    if self.nodes[j] in self.nodes[i].neighbors:
        self.nodes[i].neighbors.remove(self.nodes[j])

def get_edges(self, h=2, current_edge=[]):
    '''get all the edges containing h consecutive nodes'''
    #NOTE: IGNORES ordering self.x and self.y
    if len(current_edge) == h:
        #terminate the recursive process and return the ID's of the node in the edge
        edge = [node.ID for node in current_edge]
        #dont save duplicates. for example: [1, 2, 3, 4] and [4, 3, 2, 1].
        if edge[0] > edge[-1]:
            return []
        return [edge]
    if len(current_edge) == 0:
        #no starting point: start searching from from every node
        neighbors = self.nodes
        prev_ID = -1
    else:
        #continue seaching from current edge
        neighbors = current_edge[-1].neighbors
        prev_ID = -1
        if len(current_edge) >= 2:
            prev_ID = current_edge[-2].ID

    #build an edge using increasing neighbors
    edges = []
    for neighbor in neighbors:
        #make sure you don't traverse backwards
        if prev_ID != neighbor.ID:
            edges += self.get_edges(h, current_edge+[neighbor])
    return edges

def get_edgeclasses(self, h=2):
    '''get a list of the edgeclass of each edge containing h consecutive nodes'''
    #get branchsize
    if (self.size-1)%self.branches != 0:
        print('WARNING: the layout of the graph is not symmetrical!')
        print("Calculating edgeclasses may not make sense")
    branchsize = (self.size-1)//self.branches

    #get the edges in unsorted order
    edges = self.get_edges(h)
    S1 = len(edges)

    #calculate the edgeclass for all edges
    edgeclasses = S1*[0]
    for index, edge in enumerate(edges):
        for node in edge:
            edgeclasses[index] = get_edgeclass(edge, branchsize)
            #if node > 0:
                #add some ID in [1, 2, ... branchsize]
                #edgeclasses[index] += get_edgeclass()(node-1)%branchsize+1
    return edgeclasses

def get_edges_sorted(self, h=2, highlightcolor=RED):
    #get the edges in unsorted order
    edges = self.get_edges(h)
    if highlightcolor==RED and len(self.y) == len(edges):
        #sort the edges by strategy y
        edges = reorder(edges, self.y)
    elif highlightcolor==BLUE and len(self.x) == len(edges):
        #sort the edges by strategy x
        edges = reorder(edges, self.x)
    return edges

def draw(self, highlight=[], highlightcolor=None, basecolor=GRAY, alpha=[], label=''):
    '''draw self and highlight some nodes with the highlightcolor'''
    #set the setting of the graph G
    G = graphviz.Graph(name='my_graph.gv')
    G.graph_attr.update(nodesep="0.05")
    G.graph_attr.update(ranksep="0.05")
    G.node_attr.update(width="0.5")
    G.node_attr.update(heig="0.5")
    G.node_attr.update(fontcolor='white')

    if label!='':
        #display the label
        with G.subgraph() as S:
            S.attr(rank='same')
            S.node('label', shape='box', style='filled', color=DARKGRAY, label=label)
```

```
        #add all the nodes to the graph
        for i, node in enumerate(self.nodes):
            color = basecolor
            #draw highlighted colors
            if node.ID in highlight:
                color = highlightcolor
            #draw alpa if defined for every node
            if len(alpha) == len(self.nodes):
                if alpha[i] == 0:
                    color = LIGHTGRAY
                else:
                    color += float2hex(alpha[i], upperbound=max(alpha))
            G.node(str(node.ID), color=color, style='filled')

        #add all the edges to the graph
        for edge in self.get_edges():
            G.edge(str(edge[0]), str(edge[1]))

        #draw
        display(G)

    def draw_edges(self, h, highlightcolor=RED, draw_zero_weighted_edges=False, \
                           sort=True, representatives_only=False):
        '''draw self for each edge of length h'''

        #get edges (un)sorted:
        if sort:
            edges = self.get_edges_sorted(h, highlightcolor)
            y = sorted(self.y, reverse=True)
            x = sorted(self.x, reverse=True)
        else:
            edges = self.get_edges(h)
            y = self.y
            x = self.x

        if representatives_only:
            edgeclasses = self.get_edgeclasses(h)
            if sort:
                if highlightcolor == RED:
                    edgeclasses = reorder(edgeclasses, self.y)
                if highlightcolor == BLUE:
                    edgeclasses = reorder(edgeclasses, self.x)


        #draw the graph with edges of length h:
        for index, edge in enumerate(edges):
            #set the label according to the latest self.solve:
            if highlightcolor==RED and len(y) == len(edges):
                if not draw_zero_weighted_edges and y[index] == 0:
                    continue
                label = str(y[index])
            elif highlightcolor==BLUE and len(x) == len(edges):
                if not draw_zero_weighted_edges and x[index] == 0:
                    continue
                label = str(x[index])
            else:
                label = ''

            if representatives_only:
                #skip if edge is not a representative
                if index != get_representative(index, edgeclasses):
                    continue
                else:
                    label += '_(x{})'.format(edgeclasses.count(edgeclasses[index]))

            self.draw(edge, highlightcolor, label=label)

    def get_payoff_matrix(self, m, n):
        '''get the payoff matrix of a NHG on this graph with parameters m and n'''
        edges_P1 = self.get_edges(m)
        edges_P2 = self.get_edges(n)
        S1 = len(edges_P1)
        S2 = len(edges_P2)
        A = np.zeros((S1, S2))

        for i, edge1 in enumerate(edges_P1):
            for j, edge2 in enumerate(edges_P2):
                #get the payoff for all combinations of edges
                A[i, j] = len([intersect for intersect in edge1 if intersect in edge2])
        return A

    def solve(self, m, n, edgeclasses_equal=False, print_results=False):
        '''get the results of a NHG on this graph with parameters m and n using lp'''
```

```
        #calculate the payoff matrix
        A = self.get_payoff_matrix(m, n)

        #find the solution using linear optimization
        if edgeclasses_equal:
            #demand that edges from the SAME EDGECLASS also have THE SAME PROBABILITY
            edgeclasses1 = self.get_edgeclasses(h=m)
            edgeclasses2 = self.get_edgeclasses(h=n)
            v, y, x = lp_solve(A, print_results=print_results, edgeclasses1=edgeclasses1, \
                                                               edgeclasses2=edgeclasses2)
        else:
            v, y, x = lp_solve(A, print_results=print_results)

        #save the strategies for "get_edges_sorted" and "draw_edges"
        self.y = y
        self.x = x
        #return the results
        return A, v, y, x

    def get_value(self, m, n):
        '''solves the system and retrieves only the value'''
        _, v, _, _ = self.solve(m, n)
        return v

    def get_coverage(self, m, n, highlightcolor=RED):
        '''get a list of the coverage for each node for a NHG for parameters (m, n) on this graph'''
        #which strategy to draw the coverage of: RED or BLUE?
        if highlightcolor == RED:
            edges = self.get_edges(m)
            if len(edges) != len(self.y):
                print('WARNING: system has not been solved for (m, n) = ({}, {})'.format(m, n))
                print('Now running self.solve(m,n)...')
                self.solve(m, n)
            weights = self.y
        if highlightcolor == BLUE:
            edges = self.get_edges(n)
            if len(edges) != len(self.x):
                print('WARNING: system has not been solved for (m, n) = ({}, {})'.format(m, n))
                print('Now running self.solve(m, n)...')
                self.solve(m, n)
            weights = self.x

        #calculate the coverage for each node:
        coverage = self.size*[0]
        for i in range(len(edges)):
            for node in edges[i]:
                coverage[node] += weights[i]
        return coverage

def print_header(m, n, B, b):
    '''prints a header for G(m, n, B, b)'''
    #m: amount of coins to choose by player I
    #n: amount of coins to choose by player II
    #B: amount of branches in the tree
    #b: length of a branch

    header_str, symbol_offset, space_offset = 'G(m, n, B, b) = G({}, {}, {}, {})'.format(m, n, B, b), 10, 3
    print('\n'+'/'*(len(header_str)+(symbol_offset+space_offset)*2))
    print('/'*(len(header_str)+(symbol_offset+space_offset)*2))
    print('/'*symbol_offset+' '*(space_offset*2+len(header_str)) + '/'*symbol_offset)
    print('/'*symbol_offset + ' '*space_offset + header_str + ' '*space_offset + '/'*symbol_offset)
    print('/'*symbol_offset+' '*(space_offset*2+len(header_str)) + '/'*symbol_offset)
    print('/'*(len(header_str)+(symbol_offset+space_offset)*2))
    print('/'*(len(header_str)+(symbol_offset+space_offset)*2)+'\n')

def print_graph(B, b, header=True):
    '''prints the graph for B, b'''
    #Print header
    if header:
        print_header('-', '-', B, b)
    #Set up and solve the graph
    G = Graph(size=1+B*b, branches=B)
    G.draw()

##########################################################################
##########################################################################
#################                                  #######################
#################    LP SOLVER                     #######################
#################    (modified for the symmetries  #######################
#################    of a NHG on a graph)          #######################
#################                                  #######################
##########################################################################
##########################################################################
```

```
##### lp: definitions of the functions about linear programming (minmax and maxmin problems) #####

# A: (mxn) matrix
# m: positive integer
# n: positive integer
# x: probability COL-vector
# y: probability ROW-vector
# v: value = yAx

def int2str(i, minlength=8):
    '''converts int to a string with fixed minlength'''
    #this script is needed for alphabetical sorting
    s = str(i)
    if minlength > len(s):
        s = '0'*(minlength-len(s))+s
    return s

def get_edgeclass(edge, b):
    '''returns the edgeclass of an edge like describe in the introduction of chapter 4'''
    m = len(edge)
    if 0 in edge:
        #root strategy (= straight strategy starting on the root)
        if edge[0] == 0 or edge[-1] == 0:
            return 0
        #corner strategy
        c = edge.index(0)
        #additional symmetry of corner strategies
        if c >= m/2:
            c = m-c-1
        return -c
    #straight strategy
    return (min(edge)-1)%b + 1

def get_representative(edge, edgeclasses=[]):
    '''returns the index of the first edge in the same edgeclasses'''
    if len(edgeclasses)==0:
        return edge
    return edgeclasses.index(edgeclasses[edge])

def lp_maxmin_solve(A, edgeclasses=[]):
    '''solves a linear problem of the form max_y(min_x(yAx)'''
    m, n = A.shape

    # from player I's perspective (player I maximizes y in yAx. y is a rowvector that picks the rows of A)
    maxmin_problem = pulp.LpProblem("MaxMin", pulp.LpMaximize)

    # v = value
    v = pulp.LpVariable("v")

    # y = probability vector
    y = [pulp.LpVariable("y"+int2str(i), lowBound=0) for i in range(m)]

    # INNERMIN-CONSTRAINT: (yA)_j >= v
    for j in range(n):
        maxmin_problem += (sum(A[i][j]*y[i] for i in range(m)) >= v)

    # SUM-CONSTRAINT: sum(y) = 1
    maxmin_problem += (sum(y) == 1)

    # EDGECLASS EQUAL CONSTRAINT: PROBABILITIES FOR THE SAME EDGECLASS MUST BE EQUAL:
    for i in range(m):
        maxmin_problem += y[i] == y[get_representative(i, edgeclasses)]

    # OBJECTIVE: max v
    maxmin_problem += v

    #solve
    maxmin_problem.solve()
    v_result = maxmin_problem.variables()[0].varValue
    y_result = [maxmin_problem.variables()[i+1].varValue for i in range(m)]

    #check if pulp was able to solve the system with the EDGECLASS EQUAL CONSTRAINT
    if maxmin_problem.status != 1:
        print("WARNING: symmetric solution doesn't exists for player I")
        print("Now retrying without the edgeclass constraint...")
        v_result, y_result = lp_maxmin_solve(A)

    return v_result, y_result

def lp_minmax_solve(A, edgeclasses=[]):
    ''' solves a linear problem of the form min_x(max_y(yAx)) '''
    m, n = A.shape

    # from player II's perspective (player II minimizes x in yAx. x is a colvector that picks the columns of A)
```

```
    minmax_problem = pulp.LpProblem("MinMax", pulp.LpMinimize)

    # v = value
    v = pulp.LpVariable("v")

    # x = probability vector
    x = [pulp.LpVariable("x"+int2str(i), lowBound=0) for i in range(n)]

    # INNERMAX-CONSTRAINT: (Ax)_i <= v
    for i in range(m):
        minmax_problem += (sum(A[i][j]*x[j] for j in range(n)) <= v)

    # SUM-CONSTRAINT: sum(x) = 1
    minmax_problem += (sum(x) == 1)

    # EDGECLASS EQUAL CONSTRAINT: PROBABILITIES FOR THE SAME EDGECLASS MUST BE EQUAL:
    for i in range(n):
        minmax_problem += x[i] == x[get_representative(i, edgeclasses)]

    # OBJECTIVE: min v
    minmax_problem += v

    #solve
    minmax_problem.solve()
    v_result = minmax_problem.variables()[0].varValue
    x_result = [minmax_problem.variables()[i+1].varValue for i in range(n)]

    #check if pulp was able to solve the system with the EDGECLASS EQUAL CONSTRAINT
    if minmax_problem.status != 1:
        print("WARNING: symmetric solution doesn't exists for player II")
        print("Now retrying without the edgeclass constraint...")
        v_result, y_result = lp_minmax_solve(A)

    return v_result, x_result

def lp_solve(A, print_results=False, edgeclasses1=[], edgeclasses2=[]):
    '''find the expected payoff and optimal strategies for a game with payoff matrix A'''
    v1, y = lp_maxmin_solve(A, edgeclasses=edgeclasses1)
    v2, x = lp_minmax_solve(A, edgeclasses=edgeclasses2)
    if v1 != v2:
        if len(edgeclasses1) == 0 and len(edgeclasses2) == 0:
            print("\nStrong duality doesn't hold. Hence, all of mathematics is inconsistent. 0=1. QED\n")
        else:
            print("WARNING: symmetric solution doesn't exists. Primal and Dual differ.")
            print("Value player I   (maximized):", v1)
            print("Value player II: (minimized):", v2)
            print("Now retrying both optimization problems without the edgeclass constraint...")
            v1, y, x = lp_solve(A, print_results=print_results)
    elif print_results:
        print('### LP RESULTS ###\n')
        print(A, '\n')
        print('Value: ', str(v1), '\n')
        print('{:<36}{}'.format('Maximized strategy for the rows:', y))
        print('{:<36}{}\n'.format('Minimized strategy for the columns:', x))

    #return (value, [optimal strategy for P1, row, maximized], [optimal strategy for P2, col, minimized])
    return v1, y, x

######################################################################
######################################################################
#################                            #########################
#################   Compact <==> Full        #########################
#################                            #########################
######################################################################
######################################################################

def normalize(lst):
    '''returns the lst such that sum(lst) == 1'''
    total = sum(lst)
    return [ele/total for ele in lst]

def compact_to_full(compact_strategy, m, B, b, normalized=True):
    '''converts a strategy in compact form to full form'''
    #a compact strategy is based on general symmetries and is of the form:
    #[0, 1, 2, ..., b-m+1, -m+2, -m+3, ..., -1]
    #
    #the first part (0, 1, 2, ..., b-m+1) denotes the strategies from the orignal NHG
    #they occur B times (for each branch 1 time)
    # B * (b-m+1+1) of the strategies are of this type
    #
    #the second part (-m+2, -m+3, ..., -1) denotes corner moves that take part in AT LEAST 2 BRANCHES
    #they occur B(B-1)/2 times. ("B choose 2" times)
    # B(B-1)/2 * (m-2) of the strategies are of this type
    #
```

```
    #sidenote: corner moves are symmetric around (-m+2) // 2
    #since -m+2 could be either odd or even, I decided to define the whole interval [-m+2, ..., -1]
    #instead of the shorter version with the same information: [(-m+2)//2, ..., -1]
    #BE CAREFUL: half of this symmetric interval is disregarded during the conversion
    #so make sure the probabilities are actually symmetric


    #check if compact_strategy is actually in the desired form
    compact_length = (b-m+2) + max(0, m-2)
    if len(compact_strategy) != compact_length:
        print("WARNING: compact_strategy doesn't have the expected length!")
        print("(actual, expected=(b-m+2)+(m-2))=({}, {})".format(len(compact_strategy), compact_length))
    for i in range(0, (m-2)//2):
        if compact_strategy[-m+2+i] != compact_strategy[-1-i]:
            #the tails are not symmetric. Now setting both to the average to fix this.
            average = (compact_strategy[-m+2+i] + compact_strategy[-1-i])/2
            compact_strategy[-m+2+i], compact_strategy[-1-i] = average, average
            #print("WARNING: the tails of compact_strategy are not symmetric!")
            #print('P[{}] != P[{}]'.format(-m+2+i, -1-i))
            #print('h = {}, so there should be h-2={} corner moves'.format(m, m-2))
            #print('and b-h+1+1={} straight moves\n'.format(b-m+2))

    #set the compact edges like described above
    compact_edges = compact_length * [None]
    #first part
    for i in range(b-m+2):
        compact_edges[i] = [i+j for j in range(m)]
    #second part
    for i in range(-m+2, 0):
        compact_edges[i] = [abs(i+j) for j in range(m)]
    #print('compact_edges:', compact_edges)

    #calculate the according edgeclasses:
    compact_edgeclasses = compact_length* [0]
    for index, edge in enumerate(compact_edges):
        for node in edge:
            compact_edgeclasses[index] = get_edgeclass(edge, b)
            #if node > 0:
                #add some ID in [1, 2, ... branchsize]
                #compact_edgeclasses[index] += (node-1)%b+1

    #set the full edges and full edgeclasses
    G = Graph(size=1+B*b, branches=B)
    full_edges = G.get_edges(m)
    full_edgeclasses = G.get_edgeclasses(m)
    length = B * (b-m+2) + B*(B-1)//2 * (m-2)
    if m == 1:
        length = B * (b-m+2) - (B-1)
    if len(full_edges) != length:
        print("ERROR in 'compact_to_full': len(full_edges) != length")
        print('len(full_edges) =',len(full_edges))
        print('length =',length)

    #set the full strategy
    full_strategy = length*[0]
    for index in range(length):
        compact_index = compact_edgeclasses.index(full_edgeclasses[index])
        full_strategy[index] = compact_strategy[compact_index]
        #print('(compact_index, full_edge) = ({}, {})'.format(compact_index, full_edges[index]))
    if normalized:
        full_strategy = normalize(full_strategy)
    return full_strategy

def full_to_compact(full_strategy, m, B, b, normalized=False):
    '''converts a strategy in full form to compact form'''
    # warning #1: this method requires a symmetric full strategy
    # warning #2: this method returns compact strategy out of proportion. remember:
    #             straight moves have a weight of B
    #             corner moves have a weight of B(B-1)//2

    #calculate compact edges
    compact_length = (b-m+2) + max(0, m-2)
    compact_edges = compact_length * [None]
    #first part
    for i in range(b-m+2):
        compact_edges[i] = [i+j for j in range(m)]
    #second part
    for i in range(-m+2, 0):
        compact_edges[i] = [abs(i+j) for j in range(m)]

    #calculate the compact edgeclasses:
    compact_edgeclasses = compact_length*[0]
    for index, edge in enumerate(compact_edges):
        for node in edge:
```

```
            compact_edgeclasses[index] = get_edgeclass(edge, b)
            #if node > 0:
                #add some ID in [1, 2, ... branchsize]
                #compact_edgeclasses[index] += (node-1)%b+1

    #set the full edges and full edgeclasses
    G = Graph(size=1+B*b, branches=B)
    full_edges = G.get_edges(m)
    full_edgeclasses = G.get_edgeclasses(m)
    length = B * (b-m+2) + B*(B-1)//2 * (m-2)
    if m == 1:
        length = B * (b-m+2) - (B-1)
    if len(full_edges) != length:
        print("ERROR in 'full_to_compact': len(full_edges) != length")
        print('len(full_edges) =',len(full_edges))
        print('length =',length)

    #set the compact strategy
    compact_strategy = compact_length*[0]
    #first part
    for index in range(b-m+2):
        full_index = full_edgeclasses.index(compact_edgeclasses[index])
        #this type occurs B times
        compact_strategy[index] = full_strategy[full_index]
        #print('(index, full_index)=({},{})'.format(index, full_index))
        #print('adding', compact_strategy[index])

    #second part
    if B>1:
        #corner moves only exists if B>1
        for index in range(-m+2, 0):
            full_index = full_edgeclasses.index(compact_edgeclasses[index])
            #this type occurs B(B-1)/2 times
            compact_strategy[index] = full_strategy[full_index]
            #print('(index, full_index)=({},{})'.format(index, full_index))
            #print('adding', compact_strategy[index])

    #quick check if the full strategy was symmetric
    full_sum = sum(full_strategy)
    if m>2:
        compact_sum = sum(compact_strategy[0:b-m+2])*B + sum(compact_strategy[-m+2::])*B*(B-1)/2
    else:
        #no corner moves
        compact_sum = sum(compact_strategy[0:b-m+2])*B
    if round(full_sum, 5) != round(compact_sum, 5):
        print('WARNING: the full strategy is not symmetric')
        print('full_sum = {}'.format(full_sum))
        print('compact_sum = {}'.format(compact_sum))
        print('\nfull_strategy:')
        print(full_strategy)
        print('\ncompact_strategy:')
        print(compact_strategy)
        print('\nDEBUG THE FULL STRATEGY FROM THE WARNING:')
        Gdebug = Graph(size=1+B*b, branches=B)
        Gdebug.x = full_strategy
        Gdebug.y = full_strategy
        Gdebug.draw_edges(m, highlightcolor=RED, representatives_only=True)

    #normalize the strategy (doesn't make much sense, since this is only for 1 branch)
    if normalized:
        compact_strategy = normalize(compact_strategy)
    return compact_strategy

def get_exact_strategy(m, n, B, b, player=1, compact=False, normalized=True, print_strategy=False):
    '''create an exact strategy for G(m, n, B, b) in full form'''

    #set edgesize according to the player
    if player==1:
        h=m
    else:
        h=n

    #start off by create weights for the strategy in compact form
    compact_strategy = ( (b-h+2) + (h-2) ) * [0]

    #get the exact solution
    if (b+1) % m == 0:
        #CONJECTURE #1: 'branchsize + origin is a multiple of m'
        M = (b+1)//m
        if player == 1:
            #strategies im have equal probabilities
            for index in range(M):
                compact_strategy[index*m] = 1
        if player == 2:
```

```
            #strategies 1+index*m and (m-n)+index*m have equal probabilities
            for index in range(M):
                compact_strategy[1+index*m] = 1
                compact_strategy[(m-n)+index*m] = 1
    else:
        print('ERROR: no exact solution found for G(m, n, B, b) = G({}, {}, {}, {})!'.format(m, n, B, b))

    if compact:
        return compact_strategy
    '''finally convert the strategy from compact to full'''
    #convert to a full strategy. BE CAREFUL, after this step:
    #each straight move weighs      B          times as much
    #each corner move weighs  B(B-1)/2  times as much
    full_strategy = compact_to_full(compact_strategy, h, B, b, normalized=normalized)
    if print_strategy:
        G = Graph(size=1+B*b, branches=B)
        if player == 1:
            G.y = new_y
            G.draw_edges(m, representatives_only=True)
        else:
            G.x = new_x
            G.draw_edges(n, representatives_only=True)
    return full_strategy

def get_exact_strategies(m, n, B, b, compact=False, normalized=True, print_strategy=False, print_list=False):
    y = get_exact_strategy(m, n, B, b, player=1, compact=compact, \
                                       normalized=normalized, print_strategy=print_strategy)
    x = get_exact_strategy(m, n, B, b, player=2, compact=compact, \
                                       normalized=normalized, print_strategy=print_strategy)
    if print_list:
        print_strategies(m, n, y, x, compact=compact)
    return y, x

def get_lp_strategies(m, n, B, b, print_strategy=False, print_list=False, compact=False):
    G = Graph(size=1+B*b, branches=B)
    A, v, y, x = G.solve(m, n, edgeclasses_equal=True)
    if print_strategy:
        print_mixed_strategies(m, n, B, b, edgeclasses_equal=True)
    if compact:
        y = full_to_compact(y, m, B, b)
        x = full_to_compact(x, n, B, b)
    if print_list:
        print_strategies(m, n, y, x, compact=compact)
    return y, x

##########################################################################
##########################################################################
#################                                  #######################
#################   Plotting the experiments       #######################
#################                                  #######################
##########################################################################
##########################################################################


#plot with m, n constant and B, b variable (color, x-axis respectively)

def plot_expected_payoff(m, n, B_list, max_b, print_details=True, retval=False):
    colors = ['black', 'dimgray', 'gray', 'darkgray', 'lightgray']
    if type(B_list) == int:
        B_list = [B_list]
    min_b = max(m, n) #max(m, n)//min(2, B)
    for B in B_list:
        #set the lists for the plot (for each B)
        b_list, v_list, v_list_exact = [b for b in range(min_b, max_b+1)], [], []
        for b in b_list:
            G = Graph(size=1+B*b, branches=B)
            A, v, y, x = G.solve(m, n, edgeclasses_equal=True)
            v_list.append(v)
            v_list_exact.append(get_exact_value(m, n, B, b))
            if round(v_list[-1], 5) == round(v_list_exact[-1], 5):
                pass #print('b+1=={} -> v == v_exact '.format(b+1))
            #print('b+1 = {}, v = {}, v_exact = {}\n'.format(b+1, v_list[-1], v_list_exact[-1]))
        plt.plot(b_list, v_list, '.', color=colors[B-B_list[0]])
        plt.plot(b_list, v_list_exact, '-', color=colors[B-B_list[0]])

        #some data print commands
        if print_details:
            print('########################################')
            print('################ B = {} ################'.format(B))
            print('########################################\n')
            marker = 0
            for index in range(1, len(v_list)):
                if v_list[index-1] != v_list[index]:
                    if index-marker>1:
```

```
                        print('b = {:<2}...{:<2} : '.format(b_list[marker], b_list[index-1]), end='')
                    else:
                        print('b = {:<7} : '.format(b_list[marker]), end='')
                    print('v_______= {}'.format(str(v_list[marker:index])))
                    print('____________: v_exact = {}\n'.format(str(v_list_exact[marker:index])))
                    marker = index

    #plot the plot
    if len(B_list) > 1:
        plt.legend(handles=[mpatches.Patch(label='#Branches_B = '+str(B), \
                            color=colors[B-B_list[0]]) for B in B_list])
    plt.title('Value_V(m, n, B, b) for m = {} and n = {}'.format(m, n))
    plt.xlabel('branchsize b')
    plt.ylabel('value')
    plt.show()

    if retval:
        return v_list, v_list_exact

def get_exact_value(m, n, B, b):
    '''returns the exact value for the game G(m, n, B, b)'''
    if B <= 2:
        #standard ferguson
        p = B*b+1
        return FERGUSON_get_value(m, n, p)
    else:
        #scaled ferguson
        p = 2*b+1
        return 2/B * FERGUSON_get_value(m, n, p)

def evaluate_lp_vs_exact(m, n, B, b):
    '''returns a report for the sake of finding a conjecture to improve the direct solution'''
    exact_v = get_exact_value(m, n, B, b)
    G = Graph(size=1+B*b, branches=B)
    lp_v = G.get_value(m, n)
    print('Evaluation_report_for_(m, n, B, b)=({}, {}, {}, {}):'.format(m, n, B, b))
    if round(exact_v, 4) == round(lp_v, 4):
        if abs(exact_v-lp_v)==0:
            truncation_error = '-19'
        else:
            truncation_error = int(np.log(abs(exact_v-lp_v))/np.log(10))
        print('EXACT_V == LP_V (up to a truncation error of order 10^{})\n'.format(truncation_error))
    else:
        print('EXACT_V / LP_V == {}\n'.format(exact_v/lp_v))

######################################################################
######################################################################
##################                                  ##################
##################   High level functions           ##################
##################                                  ##################
######################################################################
######################################################################

def print_strategies(m, n, y, x, compact=True):
    '''prints the compact strategies as readable txt'''
    print('')
    if compact:
        cut = 2-m
        if cut < 0:
            #print('compact y =                    ', y)
            print('compact_y = (straight_moves)', y[0:cut])
            print('____________(corner_moves)__', y[cut::])
        else:
            #print('compact y =                    ', y)
            print('compact_y = (straight_moves)', y)
        print('')
        cut = 2-n
        if cut < 0:
            #print('compact x =                    ', x)
            print('compact_x = (straight_moves)', x[0:cut])
            print('____________(corner_moves)__', x[cut::])
        else:
            #print('compact x =                    ', x)
            print('compact_x = (straight_moves)', x)
    else:
        print('full_y = ',y)
        print('')
        print('full_x = ',x)
    print('')

def print_mixed_strategies(m, n, B, b, draw_zero_weighted_edges=False, sort=True, header=True, \
                           edgeclasses_equal=True, exact=False, custom_y=[], custom_x=[]):
    '''prints the mixed strategies for G(m, n, B, b) for player 1 and 2'''
    #Print header
```

```
    if header:
        print_header(m, n, B, b)
    #Set up and solve the graph
    G = Graph(size=1+B*b, branches=B)
    A, v, y, x = G.solve(m, n, edgeclasses_equal=edgeclasses_equal)

    #set exact strategies
    if exact:
        G.y = get_exact_strategy(m, n, B, b, player=1)
        G.x = get_exact_strategy(m, n, B, b, player=2)

    #set custom y IF PROVIDED
    if len(custom_y) > 0:
        if len(custom_y) != len(y):
            #print('{} = len(custom_y) != len(y) = {}'.format(len(custom_y), len(y)))
            #print('using a compact form? length =',len(custom_y))
            custom_y = compact_to_full(custom_y, m, B, b)
            #print('I transformed your compact y to length =',len(custom_y))
        if len(custom_y) == len(y):
            #update y to be custom_y
            print('\n%%% using custom_y %%%\n')
            G.y = custom_y
        else:
            print('ERROR: custom_y was provided, but neither in compact nor full form')
    #set custom x IF PROVIDED
    if len(custom_x) > 0:
        if len(custom_x) != len(x):
            custom_x = compact_to_full(custom_x, n, B, b)
        if len(custom_x) == len(x):
            #update x to be custom_x
            print('\n%%% using custom_x %%%\n')
            G.x = custom_x
        else:
            print('ERROR: custom_y was provided, but neither in compact nor full form')

    #print the mixed strategies
    print('Mixed Strategy player 1'+ exact*' (exact)'+':')
    if not draw_zero_weighted_edges:
        print('(Displaying {}/{} total strategies)'.format(np.count_nonzero(G.y), len(G.y)))
    G.draw_edges(m, RED, draw_zero_weighted_edges=draw_zero_weighted_edges, sort=sort, \
                         representatives_only=edgeclasses_equal)
    print('Mixed Strategy player 2'+ exact*' (exact)'+':')
    if not draw_zero_weighted_edges:
        print('(Displaying {}/{} total strategies)'.format(np.count_nonzero(G.x), len(G.x)))
    G.draw_edges(n, BLUE, draw_zero_weighted_edges=draw_zero_weighted_edges, sort=sort, \
                          representatives_only=edgeclasses_equal)

    #print value
    print('Value (lp):', v)
    print('y@x@x_____:', G.y@A@G.x)

    #print compact strategies
    print_strategies(m, n, full_to_compact(G.y, m, B, b), full_to_compact(G.x, n, B, b), compact=True)

def print_coverage(m, n, B, b, header=True, print_list=False, edgeclasses_equal=True, \
                               exact=False, custom_y=[], custom_x=[]):
    '''prints the coverage of the strategies for G(m, n, B, b) for player 1 and 2'''
    #Print header
    if header:
        print_header(m, n, B, b)

    #Set up and solve the graph with the lp
    G = Graph(size=1+B*b, branches=B)
    A, v, y, x = G.solve(m, n, edgeclasses_equal=edgeclasses_equal)

    #set exact strategies
    if exact:
        G.y = get_exact_strategy(m, n, B, b, player=1)
        G.x = get_exact_strategy(m, n, B, b, player=2)

    #set custom y IF PROVIDED
    if len(custom_y) > 0:
        if len(custom_y) != len(y):
            #print('{} = len(custom_y) != len(y) = {}'.format(len(custom_y), len(y)))
            #print('using a compact form? length =',len(custom_y))
            custom_y = compact_to_full(custom_y, m, B, b)
            #print('I transformed your compact y to length =',len(custom_y))
        if len(custom_y) == len(y):
            #update y to be custom_y
            print('\n%%% using custom_y %%%\n')
            G.y = custom_y
        else:
            print('ERROR: custom_y was provided, but neither in compact nor full form')
    #set custom x IF PROVIDED
```

```
if len(custom_x) > 0:
    if len(custom_x) != len(x):
        custom_x = compact_to_full(custom_x, n, B, b)
    if len(custom_x) == len(x):
        #update x to be custom_x
        print('\n%%% using custom_x %%%\n')
        G.x = custom_x
    else:
        print('ERROR: custom_y was provided, but neither in compact nor full form')

#Calculate and draw coverage for both players
label = 'Coverage Player I' + exact*' (exact)'
coverage = G.get_coverage(m, n, highlightcolor=RED)
if print_list:
    print(coverage)
G.draw(label=label, basecolor=RED, alpha=coverage)
label = 'Coverage Player II' + exact*' (exact)'
coverage = G.get_coverage(m, n, highlightcolor=BLUE)
if print_list:
    print(coverage)
G.draw(label=label, basecolor=BLUE, alpha=coverage)

#print value
print('Value (lp):', v)
print('y@A@x :', G.y@A@G.x)

#print compact strategies
print_strategies(m, n, full_to_compact(G.y, m, B, b), full_to_compact(G.x, n, B, b), compact=True)
```