Bachelor Thesis

# Don't Worry!
# Mensch ärgere Dich nicht

August 30, 2019

*Ramil Sabirov*

First examiner: Prof. Dr. Peter Rossmanith
Second examiner: Prof. Dr. Gerhard Woeginger

## Declaration of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work and has been done without inadmissable help of others. I have not used any other sources or aids other than those mentioned in this work. This thesis has not been submitted for any degree or other purposes in this or similar form. I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

## Selbstständigkeitserklärung

Ich versichere hiermit an Eides Statt, dass ich nach bestem Wissen und Gewissen die vorliegende Arbeit selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Ich versichere, dass der geistige Inhalt dieses Werkes Produkt meiner eigenen Arbeit ist und dass jegliche Quellen, welche die Erarbeitung dieser Ergebnisse unterstützt haben, vollständig genannt wurden.

Ort, Datum

Unterschrift

**Abstract**

In this work we compute a pure strategy Nash-Equilibrium for a 2-player variant of the classical German board game "Mensch ärgere Dich nicht" as well as the corresponding winning probabilities for all positions assuming optimal play. Subsequently, some analysis of the computed data is conducted: among other thing the optimal player is tested against simple humanly understandable playing policies.

# Contents


**1 Introduction** **1**

**2 Preliminaries** **3**
2.1 Rules . . . 3
2.2 Mathematical Concepts . . . 6
2.2.1 Markov Chains . . . 6
2.2.2 Matrix Games . . . 7

**3 Mathematical Foundation** **11**
3.1 Modeling . . . 11
3.2 Finiteness of Madn Games . . . 12
3.3 Game Transition Matrix . . . 14
3.4 Winning Probabilities . . . 16
3.5 Optimal Strategies . . . 19

**4 Algorithm and Implementation** **23**
4.1 Mapping positions onto ID's . . . 26
4.1.1 Number of positions . . . 26
4.1.2 Encoding and Decoding . . . 28
4.2 Choosing good start values . . . 36
4.3 Compression of Results . . . 38

**5 Results** **39**
5.1 Convergence of Algorithm . . . 39
5.1.1 Computation Time . . . 40
5.2 Interesting probabilities . . . 41
5.3 Games against simple AIs . . . 41
5.3.1 Furthest Piece or Capture . . . 42
5.3.2 Furthest Piece no Passing or Capture . . . 43

**6 Conclusion** **45**

# Chapter 1

# Introduction

Games amaze people. For some they might serve as occasional amusement, helping in the change from the daily grind. Other might use games as social catalysts, a reason to come together and talk with each other at a games night. But they can be a lot more than just simple pastimes: they give people the possibility to compete with others within the realm of safe borders and thus allow people to go securely to their physical and/or mental limits and express themselves artistically through the decisions and actions they take.
The competitive factor of games has made the players yearn for further heights: becoming faster, becoming stronger or especially in the case of board games: becoming smarter. Mathematicians started to dedicate their studies to analyze board games. John von Neumann's proof for the Min-Max-Theorem on existing optimal strategies for finite two player zero-sum games [1] as well as John F. Nash's generalization of the theorem [2] marked the beginning of the research branch, called 'Game Theory'. With ever increasing computational power, artificial intelligences began to reach superhuman strength levels in one game after the other, climbing the ladder of complexity. The most prominent incident may be the defeat of the at that time reigning chess world champion Garry Kasparov by the computer Deep Blue in 1997 [3]. Back then, the victory was earth-shattering. How could a machine be able to play a deeply strategic and complex game at such a high level? Nowadays, no one is wondering. We have already accepted the fate that artificial intelligences can outperform us mortals. The task of overcoming the human player has shifted over the decades to reaching for perfection. We started to solve whatever the processing power permitted: reduced variants of games like Heads-Up limit hold'em poker [4] or even games in their classical form like Checkers [5]. This work wants to continue this list, tackling the German board game 'Mensch Ärgere Dich nicht'.
'Mensch Ärgere Dich nicht' (roughly translated: 'Don't Worry!') is very similar to the game Ludo, which might be better known outside German borders, and has been invented in 1907/1908 by Josef Friedrich Schmidt. Until today more than 90 million copies of this game have been sold making it well known in Germany.
This work will first make the reader familiar with the set of rules and needed mathematical concepts in the Preliminaries. After that we formalize our task in the Mathematical Foundation and introduce an algorithm which 'solves' the game in the Algorithm and Implementation section. The most interesting parts of the implementation will be explained in this chapter as well. At the end, in the Results chapter, we will take a brief look at the computed data, pointing out some interesting winning probabilities and comparing the optimal AI, which has been obtained by solving the game, against simple humanly understandable AIs.

# Chapter 2

# Preliminaries

This chapter will lay a fundament on which the rest of the work will be built on. It is subdivided in two sections, the first expanding on the set of rules of 'Mensch ärgere Dich nicht' and the second introduces necessary mathematical concepts.

## 2.1 Rules

Throughout the years of its existence a lot of modifications to the rules have been made. This work will depend on the official set of rules [6] which have been released with the invention of the game and still are attached to new-bought copies.

### Playing Field

In Figure 2.1 you can see the playing field of *'Mensch ärgere Dich nicht'*(Madn). To start with, we want to define certain special fields and give them names so that referring in the future will be easier and smoother.
The playing field consists of 40 *main squares* which form a cross shape: the main field of the game. The sequence of the squares is equidistantly interrupted by differently colored squares: the *A-squares*. They are the entry squares for the players' game pieces (*pieces*). Before the pieces join the game, they rest on their *B-squares* (or also called *box*) which form a square (in the geometrical sense) next to the corresponding A-square. The colored squares reaching from the cross-peak to the cross-center are called the *finish zone*. In general it is true that the color of a square encodes its belonging player. This means, for example, that the red A-square marks the entry point for the red pieces, which rest on the red B-squares and head in the direction of the red finish zone.

### Goal and starting position

The goal of the game, for every player, is to bring all their pieces into the finish zone. The first player to achieve this goal wins the game.
Before a piece can enter the finish zone it has to get out of the box and, starting from the A-square, make one lap along the main squares (see Figure 2.1). This means that one piece standing on the A-square has to advance 39 squares to finish the lap and then advance additional 1 to 4 squares to reach the finish zone. It is important to notice that a player's piece can never end up in the box or the finish zone of another player throughout the game.

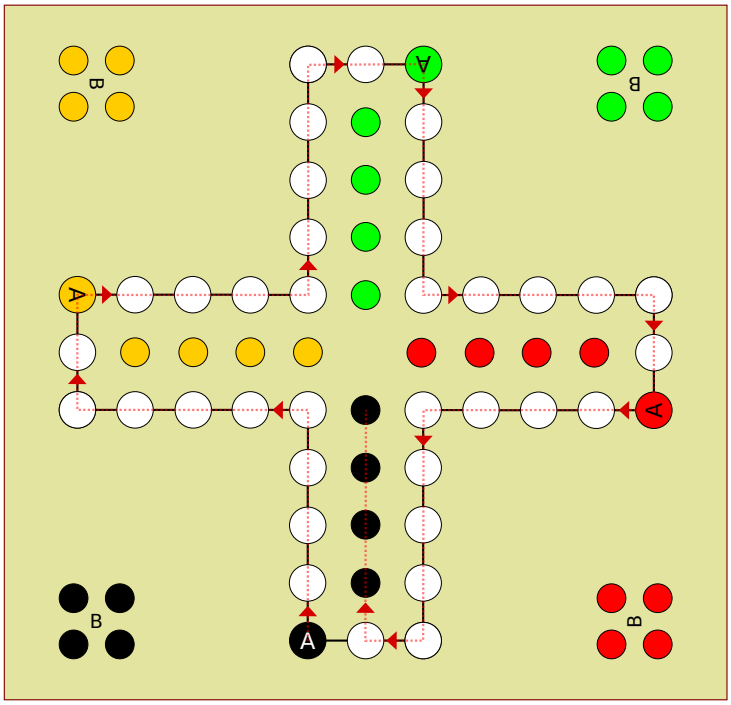



Figure 2.1: The playing field of Madn. The semi-transparent dotted red line together with the red arrows, are showing the path a piece of player *black* have to travel to reach its finish zone starting from the A-square (and are not part of the field design).

Before the game starts, every player puts one of his pieces on his A-square and the rest into the box. This is the *starting position*. From there on the players take turns in a fixed order. Usually clockwise.

### Turn

A turn starts with a single die roll. The rolled number can be assigned to any of the player's pieces in the game (not in the box) with some exceptions. The chosen piece advances by the assigned number of squares towards the finish zone. Note that the rolled number cannot be distributed to several pieces. Only one piece can move per turn.
If the advanced piece ends up on a square which is occupied by a opponent's piece, it captures the occupier, thus putting it back onto the opponent's B-squares and erasing every progress the piece made towards its finish zone.
A piece cannot be chosen if...

... the resulting square after advancing is out of the field. A piece can therefore never go further than the last field of the finish zone.

... the resulting square after advancing is already occupied by an own piece. Together with the 'capturing-rule' we have the invariant that there is at most one piece on square.

... there are (at least 1) pieces in the box and the A-square is occupied by an own piece. This means that a piece in the box could not be promoted into the game without violating the second condition. In this case the player is forced to move the piece on the A-square, as it has to be cleared.

If the third condition holds but the advancement of the piece on the A-square would violate the second condition, the player is freed from the obligation to clear the A-square and can choose any other of his own pieces to move forward, respecting the first and second condition.
If a player has no legal move, that means that every piece choice would violate the conditions, he has to pass his turn.

### The die roll 6

The die roll 6 has two special properties:

1. If possible, the player has to bring one of his pieces in the box into the game, onto his A-square.

2. The player takes another turn directly after this one.

Again, if a piece promotes from the box and there is already a opponent's piece on the A-square it gets captured just like described above. If promotion is not possible, because the box is empty or the A-square is occupied by an own piece, the same rules hold as in the section 'Turn'. Nevertheless the player takes another turn.

### Additional rules

In this section we want to expand on rule modifications which have been established over the years in a way, that people might be surprised that these are not part of the official set of rules and therefore not part of this work.

#### Triple dice roll

In cases where rolling a 6 is the only way to make a move, the player is granted three tries to throw a 6. If he manages to hit the 6 with less throws, the other tries are dismissed.

#### Alternative starting position

Instead of putting one piece on the A-square, all pieces start from the box. Usually this rule modification is combined with the triple dice roll to accelerate the start of the game.

#### No Jumping over in finish zone

It is not allowed to jump over an own piece in the finish zone. This means that the first piece that enters the finish zone has to be eventually moved to the last field of the zone to win the game, as it cannot be passed. Therefore it gets harder to position all pieces in the finish zone.

#### Obligation to capture

If a piece can capture an opponent's piece by advancing the rolled number it has to do so. If this option is missed by the player it is considered a foul and the piece, which could capture, is set back in its box.

## 2.2 Mathematical Concepts

In this section we will cover some mathematical concepts which will be used later in the work.

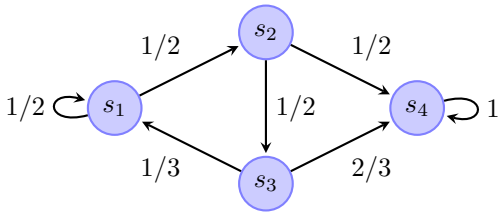
### 2.2.1 Markov Chains

A Markov Chain is describing a stochastic process. It is formally described as a set of states $S = \{s_0, \ldots, s_n\}$ and transition probabilities $p_{ij}$ for all $s_i, s_j \in S$. In step 0, a process starts in some state $s_0 \in S$. Being in a state $s_i$ it transitions to state $s_j$ with probability $p_{ij}$ in the next step. For a fixed $i$, $p_{ij}$ is describing a probability distribution over all states. Therefore it is $p_{ij} > 0$ for all $i, j$ and $\sum_{j=0}^{n} p_{ij} = 1$ for all $i$. Markov Chains can be visualized as a graph with $S$ being the nodes and $p_{ij}$ constituting the adjacency matrix. In the context of Markov Chains the matrix is usually called *Transition Matrix*.

**Example 2.2.1.** The Markov Chain for the state set $S = \{s_1, s_2, s_3, s_4\}$ and the Transition Matrix

$$P = \begin{pmatrix} 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 1/2 \\ 1/3 & 0 & 0 & 2/3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

can be visualized like this:



**Absorbing Markov Chains**

Given a Markov Chain, there might exist a state $s_i$ which is transitioning only to itself. This means $p_{ii} = 1$ and therefore $p_{ij} = 0$ for all $j \neq i$. Such a state is called an *absorbing state*. If the stochastic process would at some point transition into an absorbing state, it would stay there for all next steps, getting 'absorbed'. The absorbing states are sinks of the visualized graph. All states, which are not absorbing, are called *transient states*.
A Markov Chain is called *Absorbing Markov Chain* if it has at least one absorbing state and from every transient state there exists a path to an absorbing state (in the representing graph). In the Example 2.2.1 $s_4$ is the only absorbing state, thus $s_1, s_2, s_3$ are transient states. There are paths existing from the transient states to the absorbing state $s_4$: $s_4$ is 'reachable' from the states $s_1$, $s_2$ and $s_3$. Therefore, the Markov Chain in the example is an absorbing one.

**Canonical Form** The Transition Matrix of an Absorbing Markov Chain can be formed into a *canonical form*, which is helpful for further considerations. This is achieved renumbering the states so that the absorbing states come after the transient states. Specifically, given a Transition Matrix $P$ with $r$ absorbing states and $t$ transient states we transform $P$ to

$$P \rightsquigarrow \left(\begin{array}{c|c} Q & R \\ \hline 0 & I \end{array}\right),$$

where $Q$ is a $t \times t$ matrix, describing the transition probabilities from a transient state to another transient state, $R$ is a $t \times r$ matrix describing the transition probabilities from a transient state to an absorbing state, $I$ is a $r \times r$ identity matrix and 0 is a $r \times t$ matrix with all entries equal to 0.

**Absorption Probabilities** Having an Absorbing Markov Chain at hand, a process which starts in an arbitrary transient state will eventually be absorbed [7]. Moreover: Given the Transition Matrix $P$ in canonical form, let $b_{ij}$ denote the probability that a process starting in transient state $s_i$ is going to be absorbed in absorbing state $s_j$. Let further be $B$ the matrix with entries $b_{ij}$. Then $B$ is a $t \times r$ matrix and is equal to

$$B = (I - Q)^{-1} * R, \tag{2.1}$$

where $Q$ and $R$ are the sub-matrices as in the canonical form and $I$ a $t \times t$ identity matrix [7].

### 2.2.2 Matrix Games

Let player I and player II denote two different players. Both of them have a finite set $A_I$ and $A_{II}$ of possible actions. For every pair of actions $(a_i, a_j) \in A_I \times A_{II}$ we define a payoff for player I ($v_{ij}^I$) and player II ($v_{ij}^{II}$) through the pair $v_{ij} := (v_{ij}^I, v_{ij}^{II}) \in \mathbb{R}^2$.
The *Matrix Game* consists of both players choosing an action from their action set, without knowing the choice of the opponent. When both players have settled on their action, player I chooses $a_i$ and player II $a_j$, player I collects the payoff $v_{ij}^I$ and player II collects the payoff $v_{ij}^{II}$. This very simple game has its name because the payoffs can be illustrated in a matrix $V$ with entries $v_{ij}$. In this representation one player is settling the row and the other the column of the matrix, specifying one entry, which is leading to the respective payoffs.

**Example 2.2.2.** The game of *Rock, Paper, Scissors* can be understood as a Matrix Game. Both players have the same action set $A_I = A_{II} = \{\text{Rock}, \text{Paper}, \text{Scissors}\}$. If we say that winning

results in a payoff of 1, losing in a payoff of $-1$ and drawing is equal to the payoff of 0 we arrive at the payoff-matrix

$$V = \begin{array}{c} \begin{matrix} \textit{Rock} & \textit{Paper} & \textit{Scissors} \end{matrix} \\ \begin{bmatrix} (0,0) & (-1,1) & (1,-1) \\ (1,-1) & (0,0) & (-1,1) \\ (-1,1) & (1,-1) & (0,0) \end{bmatrix} \end{array} \begin{matrix} \\ \textit{Rock} \\ \textit{Paper} \\ \textit{Scissors} \end{matrix}$$

**Zero-Sum Games** Very often the players in games have diametrically opposed goals. A higher payoff for player I results in a lower payoff for player II and the other way around. A very important special case is where the payoffs of both players sum to zero, meaning $v_{ij}^I = -v_{ij}^{II}$ for all $i, j$. Those games are called *zero-sum games*. The "Rock, Paper, Scissors" game is one example for a zero-sum game. In zero sum games it is enough to note the payoff of one player. The 'zero-sum-variant' of the payoff matrix would therefore be given by

$$V = \begin{array}{c} \begin{matrix} \textit{Rock} & \textit{Paper} & \textit{Scissors} \end{matrix} \\ \begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} \\ \textit{Rock} \\ \textit{Paper} \\ \textit{Scissors} \end{matrix}$$

omitting the payoffs of player II (who chooses the row).

### Equilibrium and Optimal Strategies

The objective for both players of the matrix game is to maximize their own payoff. To do that, both players try to find an 'optimal' *game strategy*.

**Game Strategy** A game strategy $\sigma_I$ ($\sigma_{II}$) for player I (player II) is a probability distribution over all possible actions of his action set $A_I$ ($A_{II}$). The notion is, that the player specifies with which probability he will choose which action. If he chooses one action with the probability of 1 (and therefore all other with the probability of 0) his strategy is called *pure strategy*. Otherwise it is called a *mixed strategy*.

**(Nash-)Equilibrium** If both players settle on a game strategy $\sigma_I$ and $\sigma_{II}$, so that neither player can retrieve more payoff by altering his strategy while the other is staying put, the strategies are in a (Nash-)equilibrium[1]. Both players are not willing to change their strategy as this could only result in a decrease of their payoff. In a non-zero-sum game it might be that player II has an alternative strategy which is not changing the payoff for player II but reduces the payoff for player I. Because the only target is to maximize own payoff, player II will choose indifferently between his two options. In a zero-sum game this cannot be the case, as reducing the payoff for player I is invariably increasing player II's payoff. In this, more competitive variant, the optimal opponent will use any space to tie down player I's payoff. If player I assumes the worst case, that his opponent is this omnipotent optimal player, he has no reason to choose any other strategy than $\sigma_I$, as the opponent could always answer with $\sigma_{II}$. The same goes for player II assuming that player I is playing optimally. $\sigma_I$ and $\sigma_{II}$ are mutual best responses (*optimal strategies*), guaranteeing the maximal payoff against the toughest opponent's choice.

[1] Named after the mathematician John F. Nash

John von Neumann showed in his work 'Zur Theorie der Gesellschaftsspiele'[1], that for every two-player zero-sum matrix game such an equilibrium exists in mixed strategies.
In the Rock, Paper, Scissors game the optimal strategy for both players is to choose evenly distributed among the possible actions, meaning $\sigma_p(Rock) = \sigma_p(Paper) = \sigma_p(Scissors) = 1/3$ for both players $p \in \{I, II\}$, yielding to a expected payoff of 0.

**Saddlepoint** If we have a zero-sum Matrix Game then the position $(i, j)$ in the matrix $M$ is called a saddlepoint, if $m_{ij}$ is maximal in its column $j$ and minimal in its row $i$. Meaning that $j$ is the best pure strategy for the maximizing player and $i$ is the best pure strategy for the minimizing player. In this case the strategies $i$ and $j$ are even in an pure strategy equilbrium [8].

### Extended Form of a Game

Matrix Games seemingly have the strong restriction that the game itself consists only of one action. Most board games like chess or Go are not of this 'one-shot' nature. They consist of several moves, actions in a way, with alternating turns.
Let us assume we have a finite (board) game, with finitely many different positions (situations to take an action) and in every position we have a finite amount of possible actions. In this case we can enumerate all conceivable policies[2] a player can follow. Pitting the chosen policies of two players against each other, we arrive, again, at a Matrix Game, where the following of a policy is the equivalent of an action a player takes. The game in its natural, *extended form* has been reduced to a Matrix Game, its so called *strategic form*.

[2]The reader might like the term 'strategy' more, but it is already used in a different context

# Chapter 3

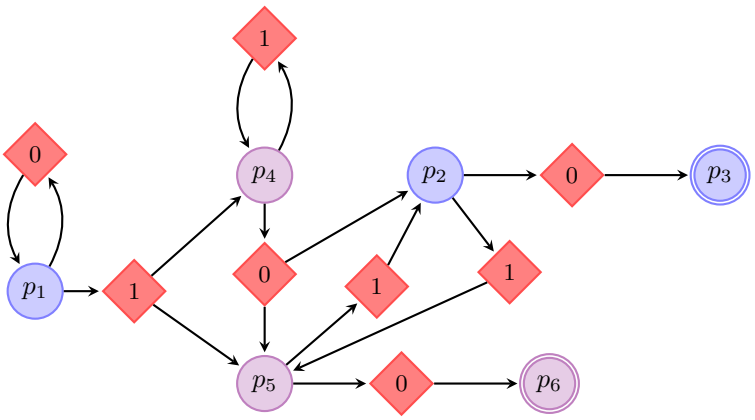
# Mathematical Foundation

## 3.1 Modeling

Even if 'Mensch ärgere Dich nicht'(Madn) can be played by four players (there is even a playing field which allows up to six players), this work will deal with 2-player Madn only.
On the classical playing field of Madn (as seen in Figure 2.1) there are four starting boxes. In the restriction of 2-player Madn, we let both players start from the boxes on opposing sides and ignore the other boxes (and finish zones). The starting position for two players is therefore symmetrical to the center of the playing field. This is the only arrangement which assures that both players' A-square has the same distance to the opponent's A-square. One could think of other arrangements of the player's starting boxes, which would lead to more unbalanced games but we stick to the symmetrical variant, like other classical board games such as chess, checkers and backgammon.
A *position* of the game is defined by a distribution of both player's pieces on the playing field and one player denoted as the next to move. A *legal position* is a position which can be reached from the starting position by taking legal turns according to the set of rules. Both player's having all their pieces in their finish zones, for example, is not a legal position as one player would have been the first one to fill his finish zone and by that he would have ended the game. Let $\mathbb{P}$ be the set containing all legal positions. We will omit the epithet 'legal' for convenience reasons. 'Position' will from here on always mean 'legal position'. There exists a subset of positions, where the game terminates due to one player winning. We call those positions *end-positions* and denote them as $\mathbb{P}_E \subsetneq \mathbb{P}$. As Madn is a game of alternating turns, there are positions with the first and with the second player to move. We denote one of the two players as the *Max-player* and the other as the *Min-player*. Whenever we talk about the winning probabilities we consider the view of the Max-player. This means that a winning probability of $P(p) = 1.0$ is equivalent to a certain win for the Max-player starting from position $p$. Obviously the Max-player is trying to maximize his winning probability, while the Min-player is trying to maximize the complementary probability, which is equivalent to minimizing the winning probability. We set $\mathbb{P}_{max}$ and $\mathbb{P}_{min}$ as the respective positions with the Max-player and Min-player to move. It is $\mathbb{P}_{max} \dot{\cup} \mathbb{P}_{min} = \mathbb{P}$. Let further $R$ be the set of possible random events. In the case of Madn we are dealing with a die roll, so in most cases it will be $R = \{1, \ldots, 6\}$. For comprehension purposes we will be looking at simple examples where $R$ describes a coin flip, being $R = \{0, 1\}$. The random events in $R$ will always be equally probable.
We can model all possible progressions of a game as a graph with the positions constituting the nodes of the graph and connect positions $p$ and $q$ by a directed edge $(p, q)$ if there exists a

transition from $p$ to $q$ by taking a turn. As a turn basically consists of two actions, the first one being the die roll and the second the subsequent move choice, we can refine our graph by an additional layer of chance nodes between two positions. To be more precise: For every non-ending position we introduce $|R|$ chance nodes, giving us the node set of $V = \mathbb{P} \cup ((\mathbb{P} \setminus \mathbb{P}_E) \times R)$. Given a position $p \in \mathbb{P} \setminus \mathbb{P}_E$, let $S_i(p)$ denote the set of possible successors (a successor a player could legally choose) for the random event $i \in R$. For the edge set $E$, we connect position node $p$ to its chance nodes $\{(q, r) : q = p, r \in R\}$ as well as the chance nodes $(q, r)$ to its possible successor position nodes $\{p : p \in S_r(q)\}$ by a directed edge. We call the directed graph $G = (V, E)$ the *game graph*.

**Example 3.1.1.** For $\mathbb{P} = \{p_1, p_2, p_3, p_4, p_5, p_6\}$, $\mathbb{P}_E = \{p_3, p_6\}$ and $R = \{0, 1\}$ the following graph depicts one possible game graph




The circles denote position nodes, the diamonds denote chance nodes. Blue circles denote positions with the Max-player to move and violet circles positions with the Min-player to move. In the circles with the doubled border, it does not matter which player's move it is, as those denote the end-positions. The color of the end-position nodes instead marks the winner of the game. Blue stands for 'Max-player wins' and violet for 'Min-player wins'. In this simplified example we are not rolling a die but flipping a coin, giving us only two chance nodes per non-ending position node. We are omitting the tuple notation for the chance nodes, as the edges clarify to which position node the chance nodes belong. Exemplary, for $p_4$ it holds that $S_0(p_4) = \{p_5, p_2\}$.

Rows and columns of matrices and vectors are usually addressed with numbers in $\mathbb{N}$. As we will often encounter the case, that we have one row (or column) for every position, we would like to address this row by the position itself rather than a natural number. Formally this can be done by a bijective labeling function $e : \mathbb{P} \to \{1, \ldots, |\mathbb{P}|\}$ and an inverse $d : \{1, \ldots, |\mathbb{P}|\} \to \mathbb{P}$ translating between position space and natural number space. Then for a vector $v$ the entry for the position $p$ can be retrieved by the evaluation of $v_{e(p)}$. For convenience reasons we will omit the explicit mapping between spaces and note $v_p$ instead. The same goes for matrices.

## 3.2 Finiteness of Madn Games

Assume we have given a game graph. For every position $p \in \mathbb{P}$ and for every die roll $r \in R$ the players can choose their desired successor out of $S_r(p)$. By doing so, the players determine their playing policies $\pi(p, r)$.

**Definition 3.2.1** (Playing Policy)**.** The playing policy for the Max-player is a function

$$\pi : (\mathbb{P}_{max} \setminus \mathbb{P}_E) \times R \to \mathbb{P}$$

and respectively for the Min-Player

$$\pi : (\mathbb{P}_{min} \setminus \mathbb{P}_E) \times R \to \mathbb{P},$$

so that $\pi(p, r) \in S_r(p)$ for all $r \in R$ and all $p \in \mathbb{P}_{max} \setminus \mathbb{P}_E$ (or for all $p \in \mathbb{P}_{min} \setminus \mathbb{P}_E$ respectively).

**Example 3.2.1.** In the game graph in Example 3.1.1 both players can choose between two different policies. The Max-player can choose between $\pi^1_{max}$ with

$$(p_1, 0) \mapsto p_1, (p_1, 1) \mapsto p_4, (p_2, 0) \mapsto p_3, (p_2, 1) \mapsto p_5$$

and $\pi^2_{max}$

$$(p_1, 0) \mapsto p_1, (p_1, 1) \mapsto p_5, (p_2, 0) \mapsto p_3, (p_2, 1) \mapsto p_5$$

and the Min-player can either choose $\pi^1_{min}$ defined by

$$(p_4, 0) \mapsto p_2, (p_4, 1) \mapsto p_4, (p_5, 0) \mapsto p_6, (p_5, 1) \mapsto p_2$$

or $\pi^2_{min}$ with mapping rules

$$(p_4, 0) \mapsto p_5, (p_4, 1) \mapsto p_4, (p_5, 0) \mapsto p_6, (p_5, 1) \mapsto p_2.$$

Note that the only *real* choice the Max-player has, is in the position $p_1$ with the die roll 1. The same is true for the Min-player in position $p_4$ with the die roll 0.

The playing policies do not take into account the previous progress of the game. There is no option for a player to specify that he would like to move to A, when he finds himself in a position for the first time and to B, otherwise. While this is a restriction for playing policies in general, it will not be for the search for the optimal policy. Because there is no exceptional game ending for position repetitions specified in the rule set of Madn(like a draw in chess after the repetition), every time the game transitions in a position it 'looks the same' to both players, no matter how often they have already seen it. If the optimal policy would require to deviate from previous play, than the previous play would have not be the optimal policy in the first place.
We now come to a theorem and a subsequent implication which argue that games in Madn cannot be played infinitely long.

**Theorem 3.2.1.** *For every position $p \in \mathbb{P}$ there exists a finite sequence of die rolls which forces one player to win the game, regardless of both players playing policies.*

*Proof.* For this proof we introduce the progress potential $Pr_i(p)$ of a position $p$ for the player $i \in \{0, 1\}$, which describes how many squares $i$'s pieces on the field can advance without the roll of a 6. For example, if one player has all his four pieces in the box, his progress potential equals zero, as no piece can move without preceding promotion to the A-square. Obviously it holds that $Pr_i(p) \geq 0$ is true for all positions $p$. Also, the piece positions of one player have no influence on the progress potential of the other. This means that once the pieces of one player are laid down on squares of the playing field, his progress potential is already determined, no matter where the opponent's pieces will be placed. It is also important to mention that not being able to move

after rolling a 1 is equivalent to the progression potential being equal to zero. Phrased in a more useful manner for this proof: If the progress potential is not zero, the player can make a move if he rolls a 1.
If we have a position $p_0$, so that $Pr_i(p_0) = 0$, it means player $i$ cannot make any progress towards the finish zone without rolling a 6. For this special case the die roll sequence is very easily constructed: If there is no progress possible, because player $i$ has finished all of his pieces, he has already won and we are done. If not, we pick player $i$ as the loser and $1-i$ as the winner. The winner will be granted the roll of the 6 whenever a position is reached where he cannot make any progress ($Pr_{1-i}(p) = 0$). Otherwise he rolls a 1. The loser will keep rolling the 1 until the end of the game. Following this scheme, the winner will be forced to move his pieces into his finish zone, while the loser cannot move and therefore cannot impede. Thus, the winner will eventually win and the loser will lose.
After we have provided a sequence for the special case let $p$ be an arbitrary position with $Pr_i(p) \neq 0$ for both players $i \in \{0, 1\}$. We now take a closer look, how the progress potential of a player is altered, when he rolls a 1. In that case one of his pieces has to advance one square and thus reduce his progress potential by one. The only side-effect which can happen is that an opponent's piece is captured. That would result in the loss of all progress potential the captured piece had, as it could not move anymore without preceding promotion. The opponent's progression potential would therefore decrease as well. Thus, the successive roll of the 1 strictly monotonically decreases the progress potential of both players, until we reach the special case with one potential being equal to zero. From there we already have a sequence of die rolls which force a player to win.
Because of the finite size of the playing field, the progression potential is finite as well. This means that the first part of the sequence (bringing one potential down to zero) is of finite length. The sequence for the subsequent special case has to be finite as well, as the winner is making progress towards his goal with every turn. □

**Corollary 3.2.1.1.** The probability that the game will continue for an infinite amount of moves is 0.

*Proof.* According to 3.2.1 there exists a finite sequence of die rolls which forces a player to win, starting in an arbitrary position $p \in \mathbb{P}$. Let $\epsilon_p$ be the probability that this sequence of die rolls will indeed happen, when in position $p$. Because the sequences are finite, it is $\epsilon_p > 0$. Let $\epsilon_{min}$ be the minimal probability out of all $\epsilon_p$. The probability that a win-forcing sequence of die rolls will *not* happen after the first $n$ moves is at most $(1-\epsilon_{min})^n$ and in an infinite game this probability becomes

$$\lim_{n\to\infty} (1-\epsilon_{min})^n = 0$$

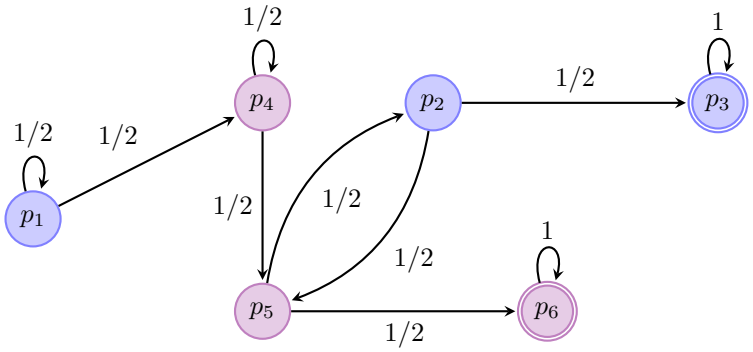
□

## 3.3 Game Transition Matrix

Once the playing policies for both players have been settled, let us call them $\pi_{min}$ for the Min-player and $\pi_{max}$ for the Max-player, the game can be regarded as a stochastic process: Starting in an arbitrary position $p$ it transitions for every random event $r$ with probability $1/|R|$ to the successor, which is determined by the responsible policy, either $\pi_{max}(p, r)$ or $\pi_{min}(p, r)$ dependent on whether in $p$ it is Max-player's or Min-player's move. This process ends, when it arrives at an end-position in $\mathbb{P}_E$. The stochastic process can be described as a Markov Chain with $\mathbb{P}$ being the states and transition probabilities like described above. End-positions are transitioning to themselves with the probability of 1, thus being absorbing states. Because of Theorem 3.2.1

there exists a path from every position in the state set of the Markov Chain to an end-position. Therefore we are dealing with an Absorbing Markov Chain. We call the Absorbing Markov Chain *Game Markov Chain* and its Transition Matrix *Game Transition Matrix* to the policies $\pi_{min}$ and $\pi_{max}$.

**Example 3.3.1.** According to Example 3.2.1 there are two playing policies for each player in the game graph in Example 3.1.1. If the Max-player chooses $\pi^1_{max}$ and the Min-player chooses $\pi^2_{min}$ we arrive at the Markov Chain




and its Game Transition Matrix

$$M = \begin{pmatrix} 1/2 & 0 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \\ 0 & 1/2 & 0 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

We can formalize the Game Transition Matrix with the following lemma.

**Lemma 3.3.1.** *The rows for non-ending positions $p \in \mathbb{P} \setminus \mathbb{P}_E$ of the Game Transition Matrix $M$ to the playing policies $\pi_{min}$ and $\pi_{max}$ are given by*

$$m_{pq} = \begin{cases} \frac{1}{|R|} * |\{r \in R : \pi_{max}(p,r) = q\}|, & \text{if Max-player to move in } p \\ \frac{1}{|R|} * |\{r \in R : \pi_{min}(p,r) = q\}|, & \text{if Min-player to move in } p \end{cases} \tag{3.1}$$

*and for end-position rows $p \in \mathbb{P}_E$ it is:*

$$m_{pq} = \begin{cases} 1, & \text{if } p = q \\ 0, & \text{if } p \neq q \end{cases} \tag{3.2}$$

*Proof.* The end-positions should transition to themselves with the probability of 1. This is obviously assured by Equation 3.2. Now let $p \in \mathbb{P} \setminus \mathbb{P}_E$ be a non-ending-position with the Max-player to move. The proof goes analogously for the Min-case.
For every random event $r \in R$ the player chooses $\pi_{max}(p,r)$ as the successor. Let $N$ be the

number of different die rolls which lead to a transition from position $p$ to $q$, i.e. $N = |\{r \in R : \pi_{max}(p,r) = q\}|$. As there are $|R|$ random events and each of them is equally likely, the probability that the game will transition from position $p$ to position $q$ by following the playing policy $\pi_{max}$ is given by

$$\frac{N}{|R|} = \frac{|\{r \in \{1,\dots,6\} : \pi_{max}(p,r) = q\}|}{|R|} = m_{pq}$$

according to Equation 3.1. □

## 3.4 Winning Probabilities

The winning probability $P(p, \pi_{max}, \pi_{min})$ (of the Max-player) starting from position $p$ is the absorption probability in an absorption state where the Max-player wins, of a process starting in state $p$ of the Game Markov Chain with the Game Transition Matrix $M$ to the settled playing policies $\pi_{min}$ and $\pi_{max}$. The winning probabilities for the end-positions $p_E \in \mathbb{P}_E$ can be easily determined as

$$P(p_E, \pi_{min}, \pi_{max}) = t(p_E) := \begin{cases} 1, & \text{if } \textit{Max-player} \text{ has won} \\ 0, & \text{if } \textit{Min-player} \text{ has won} \end{cases} \tag{3.3}$$

while the winning probabilities in the other positions are dependent on the chosen playing policies. The only thing we know for sure is that they are in the interval $[0,1]$. Noting the probabilities in a vector we can sum up all conceivable winning probabilities leading to the

**Definition 3.4.1.** Let $C$ be the set of *probability vector candidates* with

$$C := \{x \in [0,1]^{|\mathbb{P}|\times 1} : x_p = t(p_E) \text{ for all } p_E \in \mathbb{P}_E\}$$

If we have a probability vector candidate which is indeed describing the winning probabilities, i.e.: $x_p = P(p, \pi_{min}, \pi_{max})$, we call $x$ the *probability vector* to the playing policies $\pi_{max}$ and $\pi_{min}$. The next Lemma shows a useful property of the Game Transition Matrix in combination with a probability vector candidate.

**Lemma 3.4.1.** *Let $M$ be the Game Transition Matrix to the playing policies $\pi_{max}$ and $\pi_{min}$ for Max- and Min-player. Let further be $x \in C$ a probability vector candidate. Then the p-th entry of $M * x$ meets:*

$$(M*x)_p = \begin{cases} t(p), & \textit{if } p \in \mathbb{P}_E \\ \dfrac{1}{|R|}\displaystyle\sum_{i\in R} x_{\pi_{max}(p,r)}, & \textit{if } p \in \mathbb{P}_{min} \setminus \mathbb{P}_E \\ \dfrac{1}{|R|}\displaystyle\sum_{i\in R} x_{\pi_{min}(p,r)}, & \textit{if } p \in \mathbb{P}_{min} \setminus \mathbb{P}_E \end{cases}$$

*Proof.* First, let $p \in \mathbb{P}_E$ be an ending-position. Then it is:

$$(M*x)_p = \sum_{q\in\mathbb{P}} m_{pq} * x_q = m_{pp} * x_p = x_p = t(p)$$

according to Lemma 3.3.1 and Definition 3.4.1.
Let now be $p \in \mathbb{P} \setminus \mathbb{P}_E$ a non-ending position with the Max-player to move (the Min-player case is analogous). Let the random events be named, i.e: $R = \{r_1, \ldots, r_n\}$. We get:

$$\begin{aligned}
(M * x)_p =& \sum_{q \in \mathbb{P}} m_{pq} * x_q = \frac{1}{|R|} \sum_{q \in \mathbb{P}} |\{r \in R : \pi_{max}(p, r) = q\}| * x_q \\
=& \frac{1}{|R|} \sum_{q \in \mathbb{P}} |\{r = r_1 : \pi_{max}(p, r) = q\} \dot\cup \ldots \dot\cup \{r = r_6 : \pi_{max}(p, r) = q\}| * x_q \\
=& \frac{1}{|R|} \sum_{q \in \mathbb{P}} (|\{r = r_1 : \pi_{max}(p, r) = q\}| + \ldots + |\{r = r_6 : \pi_{max}(p, r) = q\}|) * x_q \\
=& \frac{1}{|R|} \sum_{q \in \mathbb{P}} \sum_{i \in R} |\{r = i : \pi_{max}(p, r) = q\}| * x_q \\
=& \frac{1}{|R|} \sum_{i \in R} \sum_{q \in \mathbb{P}} |\{r = i : \pi_{max}(p, r) = q\}| * x_q =: \frac{1}{|R|} \sum_{i \in R} \sum_{q \in \mathbb{P}} |Q_{i,q}| * x_q.
\end{aligned}$$

$Q_{i,q}$ contains at most one element. More precise: for an $i \in R$ it is (with Definition 3.5.2):

$$|Q_{i,q}| = |\{r = i : \pi_{max}(p, r) = q\}| = \begin{cases} 1, & \text{if } q = \pi_{max}(p, r) \\ 0, & \text{else} \end{cases}.$$

This means that the inner sum over all $q \in \mathbb{P}$ contains exactly one addend $\neq 0$ for $q = \pi_{max}(p, r)$. Therefore:

$$\frac{1}{|R|} \sum_{i \in R} \sum_{q \in \mathbb{P}} |Q_{i,q}| * x_q = \frac{1}{|R|} \sum_{i \in R} x_{\pi_{max}(p,r)}.$$

□

This Theorem implies that we do not need the explicit representation of the Game Transition Matrix $M$ for the matrix-vector product $M * x$ with a probability vector candidate. If we have access to the game graph, the multiplication scales down to simple linear combinations, where the factors are determined by the respective playing policies constituting $M$. Another implication of this Lemma is that the linear transformation of a probability vector candidate by a Game Transition Matrix $M$ results again in a probability vector candidate, meaning: $M * x \in C$, if $x \in C$.
We come to an integral theorem of this section, reducing the computation of the winning probabilities to an Eigenvector problem.

**Theorem 3.4.1.** *Let $\pi_{max}$ and $\pi_{min}$ be the chosen playing policies by the Max-player and Min-player and $M$ the corresponding Game Transition Matrix. Then there exists exactly one $x \in C$ so that $M * x = x$. Furthermore, $x$ is the probability vector to the playing policies $\pi_{max}$ and $\pi_{min}$.*

*Proof.* For precision reasons we will exceptionally use the bijective labeling function $e$ of the positions introduced in the modeling section in this proof. Without loss of generality let $e$ label the positions in a way so that the lowest $|\mathbb{P} \setminus \mathbb{P}_E|$ IDs are assigned to non-ending positions followed by the $|\mathbb{P}_E|$ IDs for end-positions. Let further be $e_p : \mathbb{P} \setminus \mathbb{P}_E \to \{1, \ldots, \mathbb{P} \setminus \mathbb{P}_E\}$ be a bijective function which is labeling the non-ending positions with $e_p(p) = e(p)$ and $e_{p_E} : \mathbb{P}_E \to \{1, \ldots, \mathbb{P}_E\}$

a bijective function which is labeling the end-positions in the same relative order as $e$ does, i.e. $e_{p_E}(p) = e(p) - |\mathbb{P} \setminus \mathbb{P}_E|$.
For this enumeration of positions the Game Transition Matrix $M$ is in the canonical form like defined in the preliminaries.

$$M = \left(\begin{array}{c|c} Q & R \\ \hline 0 & I \end{array}\right), \tag{3.4}$$

with $Q$ being a $|\mathbb{P}\setminus\mathbb{P}_E| \times |\mathbb{P}\setminus\mathbb{P}_E|$ matrix and $R$ being a $|\mathbb{P}\setminus\mathbb{P}_E| \times |\mathbb{P}_E|$ matrix. Let $\mathbb{P}_E^x \subset \mathbb{P}_E$ denote the end-positions, where the Max-player is winning. The winning probability $P(p, \pi_{max}, \pi_{min})$ in position $p$, when following playing policies $\pi_{max}$ and $\pi_{min}$, is the absorption probability in a state $q \in \mathbb{P}_E^x$ of a process in the Game Markov Chain which starts in the state $p$. According to 2.1 it is:

$$P(p, \pi_{max}, \pi_{min}) = \sum_{q \in \mathbb{P}_E^x} ((I - Q)^{-1} * R)_{e_p(p), e_{p_E}(q)} \tag{3.5}$$

Phrased in words: In the row $e_p(p)$ representing position $p$ we sum up all entries in the columns $e_{p_E}(q)$ representing end-positions $q$ where the Max-player wins. We rewrite the summation of relevant columns as a product with a vector $x^A \in \{0,1\}^{|\mathbb{P}_E| \times 1}$ defined by

$$x_i^A = \begin{cases} 1, & \text{if } e_{p_E}^{-1}(i) \in \mathbb{P}_E^x \\ 0, & \text{else} \end{cases} \tag{3.6}$$

for all $i \in \{1, \ldots, |\mathbb{P}_E|\}$. Then it is

$$x^T := (I - Q)^{-1} * R * x^A \tag{3.7}$$

whereby $x^T \in [0,1]^{|\mathbb{P}\setminus\mathbb{P}_\mathbb{E}| \times 1}$ describes the winning probabilities for non-ending positions, i.e.: $x^T_{e_p(p)} = P(p.\pi_{max}, \pi_{min})$. Note that $x_i^A = t(e_{p_E}^{-1}(i)) = P(e_{p_E}^{-1}(i), \pi_{max}, \pi_{min})$ is the counterpart of $x^T$, describing the winning probabilities for end-positions. Let $\begin{pmatrix} x^T \\ x^A \end{pmatrix}$ denote a vector with $x^T$ pasted on top of $x^A$. Its $i$-th entry describes the winning probability of position $e^{-1}(i)$. Furthermore it is in $C$ and it holds that

$$\begin{aligned}
M * \begin{pmatrix} x^T \\ x^A \end{pmatrix} &= \left(\begin{array}{c|c} Q & R \\ \hline 0 & I \end{array}\right) * \begin{pmatrix} x^T \\ x^A \end{pmatrix} \\
&= \begin{pmatrix} Q * x^T + R * x^A \\ x^A \end{pmatrix} \\
&\overset{3.7}{=} \begin{pmatrix} Q * (I - Q)^{-1} * R * x^A + R * x^A \\ x^A \end{pmatrix} \\
&= \begin{pmatrix} (Q * (I - Q)^{-1} + I) * R * x^A \\ x^A \end{pmatrix} \\
&= \begin{pmatrix} (Q * (I - Q)^{-1} + (I - Q)^{-1} * (I - Q)) * R * x^A \\ x^A \end{pmatrix} \\
&= \begin{pmatrix} (Q + I - Q) * (I - Q)^{-1} * R * x^A \\ x^A \end{pmatrix} \\
&= \begin{pmatrix} (I - Q)^{-1} * R * x^A \\ x^A \end{pmatrix} = \begin{pmatrix} x^T \\ x^A \end{pmatrix}
\end{aligned}$$

giving us the existence of an eigenvector to the eigenvalue of 1 to the Game Transition Matrix $M$, which is also a probability vector. It can be shown that every eigenvector to the eigenvalue of 1 of a Game Transition Matrix is indeed the probability vector, giving us the uniqueness and therefore concluding the proof. We forego proving this part, as it basically is following the proof above in the opposite direction. □

**Example 3.4.1.** In Example 3.3.1 the winning probabilities are described by the eigenvector $x \in C$ to the eigenvalue of 1 to $M$. It is

$$x = (1/3, 2/3, 1, 1/3, 1/3, 0)^{tr}.$$

Therefore, starting in position $p_1$ and following the playing policies $\pi^1_{max}$ and $\pi^2_{min}$ results in the Max-player winning the game with the probability $x_1 = 1/3$.

## 3.5 Optimal Strategies

Now that we have a way to calculate the winning probabilities we will define Madn as a Matrix Game in its strategic form.

**Definition 3.5.1** (Madn Matrix Game)**.**
Let $\Pi_{max} = \{\pi^1_{max}, \ldots, \pi^n_{max}\}$ and $\Pi_{min} = \{\pi^1_{min}, \ldots, \pi^m_{min}\}$ denote the set of all possible playing policies the Max-player and Min-player can follow.
We define $\Gamma_p$ as the *Madn Matrix Game* with $\Pi_{max}$ and $\Pi_{min}$ being the action sets of player I and player II. If player I chooses the action $\pi^i_{max}$ and player II chooses the action $\pi^j_{min}$ then the payoffs are the respective winning probabilities for the Madn game starting in position $p$: player I receives payoff $P(p, \pi^i_{max}, \pi^j_{min})$ and player II $1 - P(p, \pi_{max^i}, \pi^j_{min})$.

**Example 3.5.1.** According to Example 3.2.1, there are two playing policies for every player in the game graph in Example 3.1.1. We have already computed the winning probabilities for the confrontation of policies $\pi^1_{max}$ and $\pi^2_{min}$ in Example 3.4.1. The winning probabilities for the other three pairings are

$$x = (2/3, 2/3, 1, 2/3, 1/3, 0)^{tr}$$

for $\pi^1_{min}$ against $\pi^1_{max}$,

$$y = (1/3, 2/3, 1, 2/3, 1/3, 0)^{tr}$$

for $\pi^1_{min}$ against $\pi^2_{max}$ and

$$z = (1/3, 2/3, 1, 1/3, 1/3, 0)^{tr}$$

for $\pi^2_{min}$ against $\pi^2_{max}$. The $p_1$-Madn Matrix Game $\Gamma_{p_1}$ is:

$$\begin{matrix} \pi^1_{min} & \pi^2_{min} \\ \left[\begin{matrix} 2/3 & 1/3 \\ 1/3 & 1/3 \end{matrix}\right] & \begin{matrix} \pi^1_{max} \\ \pi^2_{max} \end{matrix} \end{matrix}.$$

The entries in the matrix denote the payoffs for the Max-player. The Min-player's payoffs are the complementary probabilities.

Note that the payoffs defined like this are not zero-sum but rather constant-sum. This is not integral, as we could restore the zero-sum property by subtracting 1 from all payoffs for the Min-player and by that not change the relative ordering of Min-player's playing policies by payoff, i.e. if the Min-player preferred playing policy $\pi_1$ over $\pi_2$ before the transformation, he

will still after.
We will now introduce a notion of optimal playing policies and their corresponding Game Transition Matrix. The definition might be confusing at first, as the playing policies are constructed in dependence of some probability vector candidate. At this point it would be best, to treat the names without any interpretation. The naming scheme will come full circle with the final theorem of the section.

**Definition 3.5.2** (Optimal Playing Policies)**.** Let $x \in C$ be a probability vector candidate. Then we define *optimal playing policies* $\pi_{min}^{opt(x)}$ for the Min-player and $\pi_{max}^{opt(x)}$ for the Max-player with

$$\pi_{max}^{opt(x)}(p,r) \in \underset{s \in S_r(p)}{\arg\max}(x_s)$$

and

$$\pi_{min}^{opt(x)}(p,r) \in \underset{s \in S_r(p)}{\arg\min}(x_s)$$

Note that if we have more than one maximal (or minimal in the Min-case) successor position, arg max consists of more than one element. In this case there are multiple optimal playing policies. In general this ambiguity can be resolved by assigning the successor with the lower ID.

**Definition 3.5.3** (Optimal Game Transition Matrix)**.** For a probability vector candidate $x \in C$ let $M(x)$ be the Game Transition Matrix to optimal playing policies $\pi_{max}^{opt(x)}$ and $\pi_{min}^{opt(x)}$. We call $M(x)$ an *Optimal Game Transition Matrix*.

Assuming that $x \in C$ is a probability vector, and is therefore describing the payoffs for every Madn Matrix Game $\Gamma_p$ for some strategies $\sigma_1$ and $\sigma_2$, the optimal playing policies choose the position as the successor, where they are getting the highest possible payoff. The Max-player policy chooses the maximum payoff and the Min-player is choosing the minimum payoff for the Max-player, which is the maximum payoff for himself. All other conceivable playing policies can only be inferior to the optimal ones.

**Theorem 3.5.1.** *Let $\pi_{max}^{opt(x)}$ and $\pi_{min}^{opt(x)}$ be optimal playing policies for Max- and Min-player to a probability vector candidate $x \in C$. Let further be $\pi'_{max}$ and $\pi'_{min}$ two (different) playing policies for Max- and Min-player. Then it is:*

$$x_{\pi_{max}^{opt(x)}(p,r)} \geq x_{\pi'_{max}(p,r)}$$

*for all $p \in \mathbb{P}_{max} \setminus \mathbb{P}_E$, $r \in R$ and*

$$x_{\pi_{min}^{opt(x)}(p,r)} \leq x_{\pi'_{min}(p,r)}$$

*for all $p \in \mathbb{P}_{min} \setminus \mathbb{P}_E$, $r \in R$.*

*Proof.* According to Definition 3.5.2 it is $\pi'_{max}(p,r) \in S_r(p)$. It follows

$$x_{\pi_{max}^{opt(x)}(p,r)} = x_{\underset{s \in S_r(p)}{\arg\max}(x_s)} \geq x_{\pi'_{max}}$$

for all $p \in \mathbb{P}_{max} \setminus \mathbb{P}_E$, $r \in R$. Note that the arg max-set in the index can be switched by out by any element of the set to ensure formal correctness. The proof for the Min-player's optimal playing policy is analogous. □

The construction of the optimal playing policies requires a probability vector $x$ to two playing policies. Now, it might turn out that these underlying playing policies are different from the optimal ones. In this case, the probability vector $x'$ to the optimal playing policies might differ from $x$. And again, the optimal playing policies constructed from $x'$ might differ from the ones constructed from $x$ and so on goes the sequence. It is clear that not all optimal playing policies, regardless of the input probability vector, can be optimal in the sense of a Nash equilibrium. However, the sequence above gets in balance, when $x$ is the probability vector to the optimal playing strategies constructed from $x$. In this case they justify their name and this is what our final theorem expresses:

**Theorem 3.5.2.** *Let $x \in C$ be a probability vector candidate and $\pi_{max}^{opt(x)}$, $\pi_{min}^{opt(x)}$ optimal playing policies for the Max- and Min-player, as well as $M(x)$ an Optimal Game Transition Matrix. If $M(x) * x = x$ then $\pi_{max}^{opt(x)}$, $\pi_{min}^{opt(x)}$ constitute an equilibrium in pure strategies for the Madn Matrix Game $\Gamma_p$ for all $p \in \mathbb{P}$.*

*Proof.* Let $x \in C$ be a probability vector candidate, so that $M(x) * x = x$. We will show that the optimal playing policies for Max- and Min-player form the equilibrium in pure strategies. For this cause, let $\pi'_{max}$ be an arbitrary alternative playing policy for the Max-player and let $M_I$ be the Game Transition Matrix to the playing policies $\pi'_{max}$ and $\pi_{min}^{opt(x)}$. Because $x \in C$ it is according to Lemma 3.4.1 and with Theorem 3.5.1

$$(M_I * x)_p = \begin{cases} t(p), & \text{if } p \in \mathbb{P}_E \\ \dfrac{1}{|R|} \displaystyle\sum_{i \in R} x_{\pi'_{max}(p,r)}, & \text{if } p \in \mathbb{P}_{max} \setminus \mathbb{P}_E \\ \dfrac{1}{|R|} \displaystyle\sum_{i \in R} x_{\pi_{min}^{opt(x)}(p,r)}, & \text{if } p \in \mathbb{P}_{min} \setminus \mathbb{P}_E \end{cases}$$

$$\leq \begin{cases} t(p), & \text{if } p \in \mathbb{P}_E \\ \dfrac{1}{|R|} \displaystyle\sum_{i \in R} x_{\pi_{max}^{opt(x)}(p,r)}, & \text{if } p \in \mathbb{P}_{max} \setminus \mathbb{P}_E \\ \dfrac{1}{|R|} \displaystyle\sum_{i \in R} x_{\pi_{min}^{opt(x)}(p,r)}, & \text{if } p \in \mathbb{P}_{min} \setminus \mathbb{P}_E \end{cases} = (M(x) * x)_p = x_p$$

for all $p \in \mathbb{P}$. If $(M_I^n * x)_p \leq x_p$ for all $p \in \mathbb{P}$ is true for one $n \in \mathbb{N}$ then because $M_I^n * x \in C$ it follows again with Lemma 3.4.1 and Theorem 3.5.1 for $n+1$

$$(M_I^{n+1} * x)_p = (M_I * (M_I^n * x))_p = \begin{cases} t(p), & \text{if } p \in \mathbb{P}_E \\ \dfrac{1}{|R|} \displaystyle\sum_{i \in R} (M_I^n * x)_{\pi'_{max}(p,r)}, & \text{if } p \in \mathbb{P}_{max} \setminus \mathbb{P}_E \\ \dfrac{1}{|R|} \displaystyle\sum_{i \in R} (M_I^n * x)_{\pi_{min}^{opt(x)}(p,r)}, & \text{if } p \in \mathbb{P}_{min} \setminus \mathbb{P}_E \end{cases}$$

$$\leq \begin{cases} t(p), & \text{if } p \in \mathbb{P}_E \\ \dfrac{1}{|R|} \displaystyle\sum_{i \in R} x_{\pi'_{max}(p,r)}, & \text{if } p \in \mathbb{P}_{max} \setminus \mathbb{P}_E \\ \dfrac{1}{|R|} \displaystyle\sum_{i \in R} x_{\pi_{min}^{opt(x)}(p,r)}, & \text{if } p \in \mathbb{P}_{min} \setminus \mathbb{P}_E \end{cases}$$

$$\leq \begin{cases} t(p), & \text{if } p \in \mathbb{P}_E \\ \dfrac{1}{|R|} \displaystyle\sum_{i \in R} x_{\pi_{max}^{opt(x)}(p,r)}, & \text{if } p \in \mathbb{P}_{max} \setminus \mathbb{P}_E \\ \dfrac{1}{|R|} \displaystyle\sum_{i \in R} x_{\pi_{min}^{opt(x)}(p,r)}, & \text{if } p \in \mathbb{P}_{min} \setminus \mathbb{P}_E \end{cases} = (M(x) * x)_p = x_p$$

for all $p \in \mathbb{P}$. By means of induction we have shown that

$$(M_I^n * x)_p \leq x_p \tag{3.8}$$

for all $p \in \mathbb{P}$ and all $n \in \mathbb{N}$. $M_I^n * x$ converges to an eigenvector $x'$ to the eigenvalue of 1 if $n$ goes to infinity. As $M_I^n * x \in C$ for all $n$ it is $\lim_{n\to\infty} M_I^n * x = x' \in C$. According to Theorem 3.4.1 $x'$ is the probability vector to the playing policies $\pi'_{max}$ and $\pi_{min}^{opt(x)}$ and because of Equation 3.8 it is $x'_p \leq x_p$ for all $p \in \mathbb{P}$. This means that if the Max-player deviates in any way from the optimal playing policy he can only worsen his winning probability, and therefore his payoff in every Matrix Game $\Gamma_p$, $p \in \mathbb{P}$, if the Min-player stays put. We get a similar result, if the Min-player is the only player changing his playing policy. Instead of Equation 3.8 we can follow completely analogously (with $M_I$ now being the Game Transition Matrix to the alternative playing policy $\pi'_{min}$ and $\pi_{max}^{opt(x)}$)

$$(M_I^n * x)_p \geq x_p. \tag{3.9}$$

for all $p \in \mathbb{P}$ and all $n \in \mathbb{N}$. Entailing that every deviation from the optimal playing policy of the Min-player increases the winning probabilities for the Max-player and is therefore not feasible either. Summing up, the optimal policies constitute the saddle point for every Madn Matrix Game $\Gamma_p$ for all $p \in \mathbb{P}$ and are therefore optimal pure strategies forming an equilibrium. □

**Example 3.5.2.** We have seen in Example 3.4.1 that if the Max-player chooses the playing policy $\pi_{max}^1$ and the Min-player chooses playing policy $\pi_{min}^2$ in the game graph depicted in Example 3.1.1 we arrive at the winning probabilities

$$x = (1/3, 2/3, 1, 1/3, 1/3, 0)^{tr}.$$

It can further be verified that

$$\pi_{max}^1(p, r) \in \arg\max_{s \in S_r(p)}(x_s)$$

for $p \in \{p_1, p_2\}$ and $r \in \{0, 1\}$ and

$$\pi_{min}^2(p, r) \in \arg\min_{s \in S_r(p)}(x_s)$$

for $p \in \{p_4, p_5\}$ and $r \in \{0, 1\}$. Entailing that $\pi_{max}^1$ and $\pi_{min}^2$ are optimal playing policies. It follows that the Game Transition Matrix to the playing policies is an Optimal Transition Matrix $M(x)$ to the probability vector $x$ with $M(x) * x = x$. According to Theorem 3.5.2 $\pi_{max}^1$ and $\pi_{min}^2$ constitute an equilibrium in pure strategies for every Matrix Game $\Gamma_p$ for all $p \in \mathbb{P}$.

# Chapter 4

# Algorithm and Implementation

Having laid down the mathematical foundation, it is quite clear how the algorithm should operate to find the equilibrium and the associating winning probabilities: we search for the fixed point $x \in C$ so that $M(x) * x = x$. With this fixed point we have the winning probabilities for the Max-player and can compute the winning probabilities of the Min-player as well (1 - winning probability). Furthermore we can read off the optimal strategies for both players from the probability vector $x$. A very simple approach is a fixed point iteration:

**Algorithm 1** Naive Fixed Point Iteration

```
procedure NaiveIteration
    initialize x ∈ C
    repeat
        x' ← M(x) * x
        x ← x'
    until ||x − x'||_∞ ≤ ε
return x
```

A more general form of this algorithm, in general known as value iteration, can be used as a solving method for Markov Games [9], also known as stochastic games [10]. We went through the struggle of introducing this algorithm through a formal mathematical foundation, as MadN, in its natural form, is not a stochastic game and it was not quite clear how to transform it conveniently.
If we would follow the naive interpretation we would have to generate the $|\mathbb{P}| \times |\mathbb{P}|$ transition matrix. As the matrix is very sparse, this would be a waste of time and resources.
According to the remark after Lemma 3.4.1 we can instead simplify the calculation to a linear combination of the probabilities of the maximal successors. Applying this tweak we arrive at the improved Algorithm 2. As a quick reminder: The functions $d$ and $e$ are the encode and decode functions and $S_r(p)$ generates all possible successor positions of $p$ under the condition that the die rolled an $r$.

**Correctness** The algorithm iterates through all entries of the probability candidate vector $x$ and determines its value of the next iteration by applying the referenced Lemma. Note the case distinction between end-positions(lines 8 to 9), Max-player positions (lines 10 to 17) and Min-player positions (lines 18 to 25), just like in the Lemma. Entries of end-positions are not altered, as they were already initialized with $t(p)$. The Max-player and Min-player case are symmetrical,

**Algorithm 2** Fixed Point Iteration

```
1:  procedure ITERATION
2:      initialize x ∈ C
3:      int changes ← 0;
4:      repeat
5:          for i in 0, ..., |ℙ| − 1 do
6:              Position p ← d(i)
7:              float newVal;
8:              if p ∈ ℙ_E then
9:                  continue
10:             else if p ∈ ℙ_max then                    ▷ The Max-player case
11:                 for r in 1, ..., 6 do
12:                     S ← S_r(p)
13:                     float maxVal ← −∞
14:                     for s in S do
15:                         if x_e(s) > maxVal then
16:                             maxVal ← x_e(s)
17:                     newVal ← newVal + maxVal/6
18:             else if p ∈ ℙ_min then                    ▷ The Min-player case
19:                 for r in 1, ..., 6 do
20:                     S ← S_r(p)
21:                     float minVal ← +∞
22:                     for s in S do
23:                         if x_e(s) < maxVal then
24:                             minVal ← x_e(s)
25:                     newVal ← newVal + minVal/6
26:             if |newVal − x_i| > ϵ then
27:                 changes ← changes + 1
28:             x_i ← newVal
29:     until changes = 0
30: return x
```

so it is enough to argue the correctness of the Max-player case. It can be seen that lines 13 to 16 determine the maximal value of all successors of p for every die roll $r$, this is $\max_{s \in S_r(p)} x_s$. After the $i$-th iteration over the die rolls it holds that

$$newVal = \frac{1}{6} \max_{s \in S_1(p)} \{x_s\} + \ldots + \frac{1}{6} \max_{s \in S_i(p)} \{x_s\}$$

entailing that after the sixth iteration *newVal* is indeed the value of the position p after the iteration.

**Precision** Due to the fact that we are running this algorithm on a machine, with restricted precision, we cannot hope to find the *exact* fixed point. We have to be content with an approximation, so it is no restriction that the algorithm terminates when no entry of $x$ changed significantly, meaning that the change was smaller than some predefined $\epsilon$. However, the choice of the $\epsilon$ is a payoff between precision and runtime. For our cause we chose $\epsilon = 10^{-6}$.

| | Naive method | | Improved method | |
|---|---|---|---|---|
| Iteration ($k$) | $x^k$ | $\|x^k - x^*\|_\infty$ | $x^k$ | $\|x^k - x^*\|_\infty$ |
| 1 | $(1, 1, 1, 1, \frac{1}{2}, 0)^{tr}$ | 0.6666 | $(\frac{7}{8}, \frac{3}{4}, 1, \frac{3}{4}, \frac{3}{4}, 0)^{tr}$ | 0.5416 |
| 2 | $(1, \frac{3}{4}, 1, \frac{3}{4}, \frac{1}{2}, 0)^{tr}$ | 0.6666 | $(\frac{23}{32}, \frac{11}{16}, 1, \frac{9}{16}, \frac{3}{8}, 0)^{tr}$ | 0.3854 |
| 3 | $(\frac{7}{8}, \frac{3}{4}, 1, \frac{5}{8}, \frac{3}{8}, 0)^{tr}$ | 0.5416 | $(\frac{75}{128}, \frac{43}{64}, 1, \frac{29}{64}, \frac{11}{32}, 0)^{tr}$ | 0.2526 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 10 | * | 2.995e−2 | * | 5.371e−3 |

Table 4.1: A table pitting the simple iteration approach seen in Algorithm 1 against the improved iteration approach in Algorithm 2 with iteration order $6 \to 3 \to 5 \to 2 \to 4 \to 1$ in the Example 3.1.1. Starting with the vector $x^0 = (1, 1, 1, 1, 1, 0)^{tr}$ and converging to $x^* = (1/3, 2/3, 1, 1/3, 1/3, 0)^{tr}$. Decimal numbers have been cut off after the 4th decimal place. The first iteration of the improved variant reaches the same precision as the naive iteration after the third iteration.

**Serial Examination** Another point that needs to be addressed is the serial processing of the positions. The values of the next iteration are directly written into $x$ by overwriting the previous ones. Now imagine the following scenario: The first $n-1$ positions were examined and we have computed the next iteration value for all of them. We continue with the $n$-th entry and realize that one of its maximal successors (say for the die roll 1) is one of the $n-1$ positions, which were already examined in this iteration. The value of position $p$ therefore does not solely depend on the last iteration but also on values calculated in this iteration. This fact will not violate the convergence of the algorithm. On the contrary, it can be usefully exploited to speed it up: the value of a position calculated during an iteration is depending on the value of its optimal successors. For the optimal successors the same is true and so on. This recursion goes on until an end-position is found, as their winning probabilities are set in stone. Thus the information of the end-positions, in form of win or lose, are pillars for the winning probabilities of all other positions and have to be propagated through the graph. Retrospect to the game graph in the Example 3.1.1. Starting with the probability vector candidate $x^0 = (1, 1, 1, 1, 1, 0)^{tr}$ you can see the outcomes for the simple Algorithm 1 and its improved variant 2 for the first few iterations in Table 4.1. Thereby, the improved method iterates through the positions in the order $6 \to 3 \to 5 \to 2 \to 4 \to 1$ which is the order of shortest distances to end-positions (obtained by ignoring the chance nodes). Because in the naive method the value of position $p$ with shortest distance $d$ to end-positions can only influence the values of positions with distances $< d+1$ in one iteration and $p_1$ has the distance 3 to an end-position, it takes three iterations to alter the first entry of the vector. On the other hand the improved method is passing the values through in one iteration. Of course, with increasing diameter of the graph, the benefit of the improved method gets even bigger.

**Parallelizing** The execution of one iteration can be excellently parallelized, as the positions can be processed completely independently from each other. However, the resulting non-deterministic procession order, makes it impossible to examine positions strictly in the order of shortest distances to end-positions. Thus, the exploitabilty of the serial examination shrinks, but this drawback will be widely outweighed by the amount of possible speed-up, which is only bounded by the number of processing cores one computing node is able to provide.

**Simplifying the Iteration** By computing the matrix-vector product over linear combinations we result in a faster convergence rate and faster calculation per iteration. Now we will see that we can cut the computing time down further by using a symmetry of the game.
In Madn every legal piece distribution on the field has two positions associated with it. One with the Max-player and the other with the Min-player to move. This has one important implication: Let $p$ be an arbitrary position with the Min-player to move and let $P(p)$ be the optimal winning probability (from Max-player's view) for the game starting in position $p$. Now we turn the tables. This means we 'recolor' all Min-player's pieces to Max-player's pieces and the other way around and set the Max-player as the next player to move. What we have formally done is transform position $p$ with the Min-player to move, to a (legal) position $p'$ with the Max-player to move. Now, the Max-player finds himself in the exactly same situation as the Min-player before the transformation and the other way around. Because both players are playing optimally, the winning probability just gets inverted by the transformation meaning $P(p) = 1 - P(p')$. Having a fixed probability vector for the Optimal Transition Matrix restricted to positions with the Max-player to move, we can construct a fixed probability vector $x$ for $M(x)$, by assigning to positions with the Min-player to move the complementary winning probability of their counterpart obtained through the transformation above. We conclude, that it is enough to consider positions with the Max-player to move only, which leads us to the final form of our Algorithm 3.

## 4.1 Mapping positions onto ID's

In contract to the mathematical section, we used the explicit encoding and decoding of positions in the pseudo-code of the algorithms. This was to point out, that this is still a task at hand. The quality of the implementation of these functions turns out to be crucial regarding computation time and should be done diligently. Indeed, this might be the hardest part of the implementation. As we have explained in the foregoing paragraph about the simplification of the iteration, we do not need to work with all possible positions. It is sufficient to consider positions with the Max-player to move. So instead of the functions $e, d$ which were introduced in the mathematical section and were mapping all positions $\mathbb{P}$, we are interested in a bijective encode function

$$e_m : \mathbb{P}_{max} \to \{0, \ldots, |\mathbb{P}_{max}| - 1\}$$

and its inverse decode function

$$d_m : \{0, \ldots, |\mathbb{P}_{max}| - 1\} \to \mathbb{P}_{max}.$$

For this matter it is helpful to know how many positions there are to map.

### 4.1.1 Number of positions

Let $P$ be the number of pieces each player controls. For both players the playing field can be divided in three disjoint 'sections': the box, the main field and the finish zone. Thereby the main field is the only section which is shared between both players. Let $S$ be the number of squares contained in the field. In the standard variant of Madn we have $P = 4$ and $S = 40$.
Given a position we can count the number of pieces the Max-player has out of his box $p_1$($p_2$ for the Min-player respectively) and how many of them are already in the finish zone $f_1 \leq p_1$ ( $f_2 \leq p_2$). We call the 4-tuple $(p_1, f_1, p_2, f_2)$ the *abstract profile* of the position.
We split up the set of all Max-player positions into *Abstract Positions* by grouping positions

**Algorithm 3** Simplified Fixed Point Iteration

```
1: procedure ITERATION
2:     initialize x ∈ C
3:     int changes ← 0;
4:     repeat
5:         for i in 0, ..., |P_max| − 1 do
6:             Position p ← d_m(i)
7:             float newVal;
8:             if p ∈ P_E then
9:                 continue
10:            else
11:                for r in 1, ..., 6 do
12:                    S ← S_r(p)
13:                    float maxVal ← −∞
14:                    if r = 6 then                          ▷ Turn does not change
15:                        for s in S do
16:                            if x_{e_m(s)} > maxVal then
17:                                maxVal ← x_{e_m(s)}
18:                    else                                   ▷ Turn does change
19:                        for s in S do
20:                            if 1 − x_{e_m(s)} > maxVal then
21:                                maxVal ← 1 − x_{e_m(s)}
22:                    newVal ← newVal + maxVal/6
23:            if |newVal − x_i| > ε then
24:                changes ← changes + 1
25:            x_i ← newVal
26:    until changes = 0
27: return x
```

with the same abstract profile together:

$$A(p_1, f_1, p_2, f_2) := \{p \in \mathbb{P}_{max} : \text{Max-player has } p_1 \text{ pieces out of his box and } f_1 \text{ pieces in his finish zone,}$$
$$\text{Min-player has } p_2 \text{ pieces out of his box and } f_2 \text{ pieces in his finish zone}\}$$

The Abstract Positions are a partition of $\mathbb{P}_{max}$, meaning that the disjoint union of all Abstract Positions is

$$\bigcup_{p_1=0}^{P} \bigcup_{f_1=0}^{p_1} \bigcup_{p_2=0}^{P} \bigcup_{f_2=0}^{p_2} A(p_1, f_1, p_2, f_2) = \mathbb{P}_{max}$$

All but one distributions of both players' pieces on the field, so that no square is occupied more than once, is reachable from the start position, by following the rules. It follows that the number of positions grouped into one Abstract Position is exactly the number of possible piece distributions with no square occupied multiple times. However, there is one exception to this.[1]

[1] Indeed there exists more than one exception. For example, a position where the only piece out of the box is standing on its A-square and the related player is not next to move, is also not reachable from the starting position following the set of rules, as the position is lacking a predecessor. However, it is clear how the game could progress from this position according to the rule set. It can be regarded as an alternative starting position. This is why we will not sort those positions out.

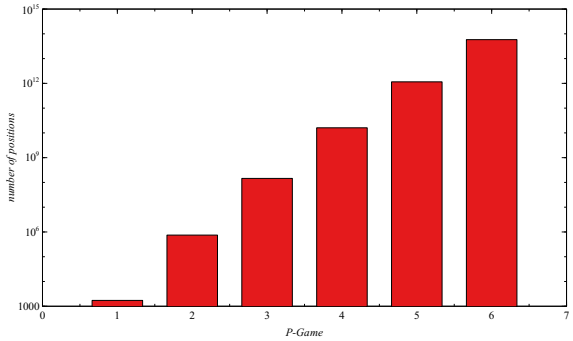



Figure 4.1: Number of positions of the P-games

It is not possible to reach a state where both players have finished all their pieces, as one player had to bring in his first and would have ended the game. Thus, $A(P, P, P, P) = \emptyset$.
If we have a given the abstract profile $(p_1, f_1, p_2, f_2)$ with $p_1, p_2 \leq P$ and $f_1 \leq p_1$ as well as $f_2 \leq p_2$, we can count the number of positions in the following way: For the first player we can choose (without consideration of order) $f_1$ squares out of the $P$ finish-zone-squares and $p_1 - f_1$ (number of pieces on the main field) squares out of the $S$ main-field-squares. For the second player we can choose $f_2$ squares out of his $P$ finish-zone-squares as well, as these squares cannot be occupied by the first player's pieces. Though, the main field has already $(p_1 - f_1)$ squares occupied. This means the second player can only choose squares to place his $p_2 - f_2$ pieces on the remaining $S - (p_1 - f_1)$ free squares. Technically speaking, as there are four B-squares, there are multiple possibilities to place pieces on them as well. But as their position on the B-squares does not influence the game at all, we sum up the box as one square and just as 'out of the game'. For all Abstract Positions except $A(P, P, P, P)$ we arrive at:

$$|A(p_1, f_1, p_2, f_2)| = \binom{S}{p_1 - f_1} * \binom{P}{f_1} * \binom{S - (p_1 - f_1)}{p_2 - f_2} * \binom{P}{f_1} \tag{4.1}$$

ans because the Abstract Positions form a partition of $\mathbb{P}_{max}$ we get

$$|\mathbb{P}_{max}| = \sum_{p_1=0}^{P} \sum_{f_1=0}^{p_1} \sum_{p_2=0}^{P} \sum_{f_2=0}^{p_2} |A(p_1, f_1, p_2, f_2)| \tag{4.2}$$

Using this formula for the standard variant we arrive at roughly 16 billion different positions (16053053985).
The formula has been derived for an variable number of main squares and pieces as one can think of modifications to the game such as playing with less (or more) than 4 pieces. We call such a game *P-game*. As they have the same rules and only the complexity of the game is decreased (increased), they are well suited for testing and can provide useful information for the 'next-order' game, like for example clues for better starting values for the iteration (see 4.2). An illustration of the reduction of complexity can be seen in Figure 4.1. Note the logarithmic scale!

### 4.1.2 Encoding and Decoding

For the encoding and decoding of positions we stay very close to the formula of calculating the number of different possible positions. We split the position ID's in sections for every Abstract

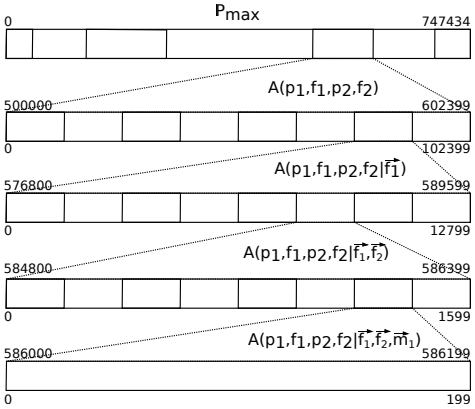



Figure 4.2: Exemplary hierarchical ID structure. The upper layer denotes all position $\mathbb{P}_{max}$. Each lower layer is 'zooming in' on one part of the upper layer.

Position. We can partition the Abstract Position further by grouping together all positions with the same piece distribution in the finish zone of player 1. This distribution can be interpreted as a $f_1$-combination of $[0, P-1]$ and can be depicted as a $P$-bit string with $f_1$ 1's marking the positions of the player's pieces on the $P$-squared finish zone. There are $\binom{P}{f_1}$ different possibilities of distributing player 1's pieces in the finish zone. Let $\vec{f_1}$ denote such a distribution and let $A(p_1, f_1, p_2, f_2|\vec{f_1})$ be the subset of the Abstract Position $A(p_1, f_1, p_2, f_2)$, with player 1's pieces distributed according to $\vec{f_1}$. Then it is

$$a_f(f_1, p_1, f_2, p_2) := |A(p_1, f_1, p_2, f_2|\vec{f_1})| = \frac{|A(p_1, f_1, p_2, f_2)|}{\binom{P}{f_1}}.$$

Note that the kind of piece distribution does not influence the size of the subset. For all piece distributions the size is the same. This will hold for the next to partitions as well.
Along the same lines we partition $A(p_1, f_1, p_2, f_2|\vec{f_1})$ further by grouping together positions with the same piece distribution of player 2's finish zone pieces $\vec{f_2}$. We denote this group as $A(p_1, f_1, p_2, f_2|\vec{f_1}, \vec{f_2})$ and its size always is:

$$a_{ff}(f_1, p_1, f_2, p_2) := |A(p_1, f_1, p_2, f_2|\vec{f_1}, \vec{f_2})| = \frac{|A(p_1, f_1, p_2, f_2|\vec{f_1})|}{\binom{P}{f_2}}.$$

And we partition $A(p_1, f_1, p_2, f_2|\vec{f_1}, \vec{f_2})$ one last time by grouping together positions with the same piece distributions of player 1's pieces on the main field $\vec{m_1}$. The notation is accordingly and the group size is:

$$a_{ffm}(f_1, p_1, f_2, p_2) := |A(p_1, f_1, p_2, f_2|\vec{f_1}, \vec{f_2}, \vec{m_1})| = \frac{|A(p_1, f_1, p_2, f_2|\vec{f_1}, \vec{f_2}|}{\binom{S}{p_1 - f_1}} = \binom{S - (p_1 - f_1)}{p_2 - f_2}.$$

The order of partitioning is chosen arbitrarily. An exemplary and figurative depiction of the position ID structuring can be seen in Figure 4.2. The upper bar denotes all position IDs (in this example $[0, 747434]$), which are split up in the different Abstract Positions. The second bar is 'zooming in' on one Abstract Position holding the IDs $[500000, 602399]$. This is again split in the

Abstract Positions with determined distribution of player 1's finish zone pieces. This hierarchy goes on until the fifth layer, where the only free variable is the distribution of player 2's pieces on the main field. Note that the subdivision of the ID intervals, which is indicated by the vertical lines in the bars, are just exemplary. This does not have to be a real scenario. What is true, however, is that the bars 2 to 4 are indeed equidistantly subdivided, meaning that every chunk holds the same number of IDs. This matches the formulas depicted above.
Following this hierarchy, the encoding and decoding of positions breaks down to encoding and decoding of combinations. We will elaborate on them a little later. For now let $e_k^n : [0,1]_k^n \to \{0, \ldots, \binom{n}{k} - 1\}$ be the encoding function, mapping a $n$-bit string with exactly $k$ 1's, and $d_k^n : \{0, \ldots, \binom{n}{k} - 1\} \to [0,1]_k^n$ its inverse.
Given a position $p$ we first determine the abstract profile and look up its starting ID, the base $b(A(f_1, p_1, f_2, p_2))$ ($= 500000$ in the example above). We now have 'zoomed in' on this abstract profile and descend in the next layer of hierarchy. If we go with the example in Figure 4.2, we are now searching globally for a ID in the interval $[500000, 602399]$, but locally (in this layer) we compute an offset $o$ in $[0, 102399]$ so that $b(A(f_1, p_1, f_2, p_2)) + o$ is gives us the position ID. For that we determine the ID of the first player's finish piece distribution $e_{f_1}^P(\vec{f_1})$ and because we know the size of each chunk in this layer $a_f(f_1, p_1, f_2, p_2)$, we can compute the starting ID to this specific piece configuration as $a_f(f_1, p_1, f_2, p_2) * e_{f_1}^P(\vec{f_1})$: interpretable as the base for the second layer and search for the corresponding offset in the third layer. We follow this scheme until the last layer, where the offset is determined by the ID of the second player's main field pieces. Summing up: having a position $p$ with abstract profile $(f_1, p_1, f_2, p_2)$ we can note the encoded ID as

$$\begin{aligned} e(p) = b(A(f_1, p_1, f_2, p_2)) &+ a_f(f_1, p_1, f_2, p_2) * e_{f_1}^P(\vec{f_2}) \\ &+ a_{ff}(f_1, p_1, f_2, p_2) * e_{f_2}^P(\vec{f_1}) \\ &+ a_{ffm}(f_1, p_1, f_2, p_2) * e_{p_1 - f_1}^S(\vec{m_1}) \\ &+ e_{p_2 - f_2}^{S-(p_1-f_1)}(\vec{m_2}). \end{aligned}$$

Thereby $\vec{m_2}$ is the distribution of the second player's main field pieces. If we cross out the squares on the main field which are already occupied by the first player ($p_1 - f_1$ in total), $\vec{m_2}$ can be understood as a $(p_2 - f_2)$-combination of the remaining $S - (p_1 - f_1)$ squares.
The decoding process makes use of the partition hierarchy as well: Let $z$ be the encoded position ID. We can the determine the abstract profile by determining in which interval chunk in the first layer $z$ is contained. Subtracting the base of the Abstract Position $o \leftarrow z - b(A(...))$ we are searching for the offset $o$ in the following layers. From there on $o$ div $a_f(...)$ retrieves us the first player's finish distribution ID, which can be decoded to the piece distribution with $d_{f_1}^P$, and $o \mod a_f(...)$ the searched offset in the third layer. We continue in this manner until we reach the fifth layer. In the end we can use the retrieved piece distributions of the first and second player's pieces on the main field and in their finish zones to reconstruct the position. The complete decoding algorithm can be seen in 4. By precomputing $b(...)$, $a_f(...)$, $a_{ff}(...)$, $a_{ffm}(...)$ for every Abstract Position, the encode (decode) process gets reduced to encoding (decoding) four combinations: two for both players' finish zone pieces and two for their main field pieces.

### Encoding and Decoding Combinations

As we have seen the core component for our encoding and decoding of positions is the encoding and decoding of combinations. If $[0,1]_k^n$ denotes all $n$-bit strings with exactly $k$ 1's, then its cardinality is $|[0,1]_k^n| = \binom{n}{k}$. We define the positions of the string different to the usual convention.

**Algorithm 4** Position Decode

**procedure** DECODEPOSITION($z$)
  find abstract profile $p_{rof} = (f_1, p_1, f_2, p_2)$ with maximal $b(A(p_{rof}))$ so that $b(A(p_{rof})) \leq z$
  $o \leftarrow z - b(A(p_{rof}))$
  $id_{\vec{f_1}} \leftarrow o$ div $a_f(p_{rof})$ ▷ retrieve distribution id of first player's finish zone pieces
  $o \leftarrow o \mod a_f(p_{rof})$
  $id_{\vec{f_2}} \leftarrow o$ div $a_{ff}(p_{rof})$ ▷ retrieve distribution id of second player's finish zone pieces
  $o \leftarrow o \mod a_{ff}(p_{rof})$
  $id_{\vec{m_1}} \leftarrow o$ div $a_{ffm}(p_{rof})$ ▷ retrieve distribution id of first player's main field pieces
  $id_{\vec{m_2}} \leftarrow o \mod a_{ffm}(p_{rof})$ ▷ retrieve distribution id of second player's main field pieces
**return** $p$, constructed from $d^P_{f_1}(id_{\vec{f_1}})$, $d^P_{f_2}(id_{\vec{f_2}})$, $d^S_{p_1-f_1}(id_{\vec{m_1}})$ , $d^{S-(p_1-f_1)}_{p_2-f_2}(id_{\vec{m_2}})$

The left outermost bit is given position $n-1$ and the right outermost 0. We denote an access to the position $p$ of a bit-string as $t(p)$. The reason for this unusualness is the resulting ordering of positions. By setting the 'most valued bit' of the positions in the beginning we achieve that positions with the pieces closer to the box get a higher ID. If we iterate through the IDs in ascending order, we would first examine the positions with the pieces close to the finish zone. This is good for fast propagation as we have seen in the paragraph 'Serial Examination' above.

**Encoding Combinations** We are searching for a bijective function $e^n_k : [0,1]^n_k \to \{0, \ldots, \binom{n}{k} - 1\}$ as well as its inverse (in the next paragraph). We define our encoding function recursively like this:

$$e^n_k(t_0, t_1, \ldots, t_{n-1}) = \begin{cases} 0, & \text{if } k = 0 \\ e^{n-1}_k(t_1, \ldots, t_{n-1}), & \text{if } k > 0 \text{ and } t_0 = 0 \\ \binom{n-1}{k} + e^{n-1}_{k-1}(t_1, \ldots, t_{n-1}), & \text{if } k > 0 \text{ and } t_0 = 1 \end{cases} \tag{4.3}$$

We can imagine the application of the encoding function as an iteration, which is starting at the left end of the bit-string (on position $n-1$) and proceeds toward the other end. The index in the superscript $n$ is always keeping track of the rest string length and the index $k$ keeps track of how many 1's are still to come. Arriving at a 1 with $k$ 1's to come, we summon a binomial coefficient which expresses how many different combinations were 'skipped' by placing the 1. If it were not placed on this position we would have $k$ 1's which could be distributed among the rest of the bit string of size $n-1$, thus $\binom{n-1}{k}$. Note that the superscript $n$ is also denoting the position of the input string which has been passed in the last iteration. Therefore $n-1$ is the position currently looked at.
To evaluate the encoding function for one $t \in [0,1]^n_k$ it is enough to know the positions $n-1 \geq n_k > n_{k-1} > \ldots > n_1 \geq 0$ of the 1's in the bit-string. In this case we do not need to find the positions through the recursion. It simplifies and we get:

$$e^n_k(t) = \binom{n_k}{k} + \binom{n_{k-1}}{k-1} + \ldots + \binom{n_1}{1} \tag{4.4}$$

From here on now we use this simpler evaluation of $e^n_k$, as the positions will be explicitly given and have not to be found by the recursion first.
We now want to argue for the bijectiveness of our provided function. For this we take a look at the following enumerator of the possible combinations in $[0,1]^n_k$:

**Algorithm 5** Combination Enumerator

```
1:  procedure ENUMERATE
2:      initialize t ∈ [0,1]_k^n with 1's at positions n_1 ← 0, ..., n_k ← k − 1
3:      repeat
4:          print(t)
5:          Find the 1 with the lowest position n_i < n − 1 so that t(n_i + 1) ≠ 1
6:          if there exists no such 1 then return
7:          else
8:              n_i ← n_i + 1                          ▷ Move the 1 one position further
9:              for j in 1, ..., i − 1 do              ▷ Reset all n_j to beginning for j < i
10:                 n_j ← j − 1
11:     until True
```

The Algorithm starts with the combination

| Position | $n-1$ | $\dots$ | $k$ | $k-1$ | $\dots$ | $0$ |
|---|---|---|---|---|---|---|
| Bit-String | 0 | $\dots$ | 0 | 1 | $\dots$ | 1 |

Table 4.2: Starting combination

In an arbitrary iteration, before the modifications of the bit string have begun (Before line 9) it has the form

| Position | $\dots$ | $n_i+1$ | $n_i$ | $n_{i-1}$ | $\dots$ | $n_1$ | $\dots$ |
|---|---|---|---|---|---|---|---|
| Bit-String | $\dots$ | 0 | 1 | 1 | $\dots$ | 1 | $\dots$ |

Table 4.3: Arbitrary combination before the modifications

Note that left of position $n_i + 1$ there are additional 1's at the positions $n_k, \dots, n_{i+1} > n_i + 1$. Everything right of $n_1$ is filled with 0's. At this point the modifications of the bit-string begin. The position of the 1 at $n_i$ is incremented by one and the $i-1$ lower 1's are reset to the beginning on the positions $i-2$ to 0. Following lines 9 and 10 of the Algorithm we get to the combination

| Position | $\dots$ | $n_i+1$ | $n_i$ | $\dots$ | $i-3$ | $i-2$ | $\dots$ | $0$ |
|---|---|---|---|---|---|---|---|---|
| Bit-String | $\dots$ | 1 | 0 | $\dots$ | 0 | 1 | $\dots$ | 1 |

Table 4.4: Arbitrary combination after the modifications

The values at positions higher than $n_i+1$ have not changed. After a total of $\binom{n}{k}-1$ modifications the algorithm eventually arrives at

| Position | $n-1$ | $\dots$ | $n-k$ | $n-k+1$ | $\dots$ | $0$ |
|---|---|---|---|---|---|---|
| Bit-String | 1 | $\dots$ | 1 | 0 | $\dots$ | 0 |

Table 4.5: Terminal combination

where it terminates, as the only 1 which has a free next square would be at position $n-1$ and that is already maximally advanced.
It is pretty obvious that the algorithm is indeed printing out all possible combinations. Additionally no combination is printed out more than once. This can be formally verified by interpreting the bit-string as a binary number, with the left outermost bit being the most significant. In this case the number would strictly increase with every iteration step, contradicting that one bit-string is printed out twice.
In the following we will see that the order at which a bit-string is printed by the Enumerator is equal to the labeling order our encoding function provides. For this cause we will need the following identity for all $n \geq k \geq 0$:

$$\binom{n+k}{k} - 1 = \sum_{i=1}^{k} \binom{n+i-1}{i} \tag{4.5}$$

*Proof.* For $n \geq k = 0$ it follows:

$$\binom{n+0}{0} - 1 = 0 = \sum_{i=1}^{0} \binom{n+0-1}{i}$$

We now assume that the identity is true for some $k \geq 0$. Because it holds that $\binom{n-1}{j} + \binom{n-1}{j-1} = \binom{n}{j}$ for all $n \geq j \geq 1$ it follows for $k+1$:

$$\sum_{i=1}^{k+1} \binom{n+i-1}{i} = \binom{n+k}{k+1} + \sum_{i=1}^{k} \binom{n+i-1}{i} = \binom{n+k}{k+1} + \binom{n+k}{k} - 1 = \binom{n+k+1}{k} - 1$$

By means of induction the identity is proven for all $n \geq k \geq 0$. □

We now come to the equivalence of the enumerator and the encoding function. Let $t$ be the lastly printed combination by the enumerator then it is in the form of Table 4.3. Let $n_k, \ldots, n_1$ be the positions of the 1's just like denoted in the table. As for every $j < i$ the next position $n_j + 1$ is occupied by the next 1, it is $n_2 = n_1 + 1, \ldots, n_3 = n_2 + 1 = n_1 + 2$ etc. In general we ensue $n_j = n_1 + j - 1$ for all $j \in \{1, \ldots, i\}$. Therefore our encoding function maps $t$ to

$$\begin{aligned} e_k^n(t) = \binom{n_k}{k} + \ldots + \binom{n_1}{1} &= \binom{n_k}{k} + \ldots + \binom{n_{i+1}}{i+1} + \binom{n_1+i-1}{i} + \ldots + \binom{n_1+1-1}{1} \\ &= \sum_{j=i+1}^{k} \binom{n_j}{j} + \sum_{j=1}^{i} \binom{n_1+j-1}{j} \end{aligned}$$

After the modification of $t$ we get $t'$, as it can be seen in 4.4. Because positions $n_k, \ldots, n_{i+1}$ have not changed for $t'$ we can conclude with Equation 4.5

$$\begin{aligned} e_k^n(t') &= \sum_{j=i+1}^{k} \binom{n_j}{j} + \binom{n_i+1}{i} + \binom{i-2}{i-1} + \ldots + \binom{0}{1} \\ &= \sum_{j=i+1}^{k} \binom{n_j}{j} + \binom{n_1+i}{i} = e_k^n(t) + 1 \end{aligned}$$

The starting combination $t_0$ of the enumerator, as it can be seen in 4.2, is encoded to:

$$e_k^n(t_0) = \binom{k-1}{k} + \ldots + \binom{0}{1} = 0.$$

Therefore the combinations printed out by the enumerator are given consecutive ID's starting with 0. Because every combination is printed out at exactly one time, it follows that the encoding function is injective and because of the finite and equal cardinality of the domain and value range it has to be bijective as well.
The encoding function reaches its maximum at the terminal combination $t_E$, as seen in 4.5. For this combination the positions of the 1's are $n_k = n - 1$, $n_{k-1} = n - 2$ etc. This leads us to the identity

$$(e_k^n(t_E) =) \sum_{i=1}^{k} \binom{n-i}{i} = \binom{n}{k} - 1 \tag{4.6}$$

for all $n \geq k \geq 0$.

**Decoding Combination IDs** Because we can bijectively encode combinations, there has to be an inverse decoding function. One by-product of the bijectivity of $e_k^n$ is the following

**Theorem 4.1.1.** *Let be $z \in \{0, \ldots, \binom{n}{k} - 1\}$. Then $z$ is composed uniquely by the summation of $k$ binomial coefficients of the form*

$$z = \sum_{i=1}^{k} \binom{n_i}{i}$$

*with $n - 1 \geq n_k > n_{k-1} > \ldots > n_1 \geq 0$.*

*Proof.* Because $z$ is in the value range of $e_k^n$ and the encoding function is surjective, there exists a combination $t \in [0,1]_k^n$, so that $e_n^k(t) = z$. As we can see in Equation 4.4 the encoding function evaluates to $k$ binomial coefficients with the $n_i$'s being the positions of the 1's and fulfilling $n - 1 \geq n_k > n_{k-1} > \ldots > n_1 \geq 0$.
Assuming that there are two sets of binomial coefficients which sum up to $z$. So there are $n - 1 \geq n_k > n_{k-1} > \ldots > n_1 \geq 0$ and $n - 1 \geq n'_k > n'_{k-1} > \ldots > n'_1 \geq 0$ with

$$\sum_{i=1}^{k} \binom{n'_i}{i} = z = \sum_{i=1}^{k} \binom{n_i}{i}$$

The $n_i$ and $n'_i$ can be interpreted as positions of 1's in combinations $[0,1]_k^n$. According to Equation 4.4 those induced combinations $t$ and $t'$ are encoded to the equal sums $e_k^n(t) = \sum_{i=1}^{k} \binom{n_i}{i}$ and $e_k^n(t') = \sum_{i=1}^{k} \binom{n'_i}{i}$. Because of the injectivity of the encoding function, this results in $t = t'$ and therefore $n_i = n'_i$ for all $i \in \{1, \ldots, k\}$. □

With this Uniqueness Theorem we can give a correctness proof for the following decoding algorithm, which is inverting the ID $z$ to its corresponding combination:

**Algorithm 6** Decoder

```
1: procedure DECODE(n, k, z)
2:     initialize t ← 0^n
3:     while k > 0 do
4:         Find maximal n_max ≤ n − 1, so that z ≥ (n_max choose k)
5:         t(n_max) ← 1
6:         z ← z − (n_max choose k)
7:         n ← n_max
8:         k ← k − 1
9: return t
```

*Proof of correctness.* Let $z \in \{0, \ldots, \binom{n}{k} - 1\}$ be an encoded ID. According to the Uniqueness Theorem 4.1.1 it can be uniquely written as the sum of binomial coefficients

$$z = \sum_{i=1}^{k} \binom{n_i}{i}$$

whereby $n - 1 \geq n_k > \ldots > n_1 \geq 0$ are the positions of the 1's in the original combination. As the function parameters are modified during the execution of the algorithm we remember the parameters as capital letters $K$, $N$ and $Z$.
After the $i$-th iteration of the while-loop in the provided decoding algorithm it holds that $k = K - i$, $z = \sum_{j=1}^{k} \binom{n_j}{j}$ and most importantly the $i$ most significant 1's are correctly set in $t$, meaning that at positions $n_K, \ldots, n_{K-i+1}$ are 1's and all other positions are left blank.
After the 0-th (before the first) iteration the invariant holds. Let now be the beginning of the $(i+1)$-th iteration and let the invariant be true for the $i$-th iteration. This means that $z = \sum_{j=1}^{k} \binom{n_j}{j}$ with $k = K - i$. We argue that in line 4 $n_k$ is the maximal position which fulfills $z \geq \binom{n_k}{k}$. For that we assume that the algorithm instead finds a $n^* > n_k$ with $z \geq \binom{n^*}{k}$. Then it is because $n_j > n_{j-1}$ for all $j \in \{2, \ldots, k\}$ and Equation 4.6

$$0 \leq z - \binom{n^*}{k} < z - \binom{n_k}{k} = \sum_{j=1}^{k-1} \binom{n_j}{j} \leq \sum_{j=1}^{k-1} \binom{n_k - j}{j} = \binom{n_k}{k-1} - 1$$

This means that $z - \binom{n^*}{k} \in \{0, \ldots, \binom{n_k}{k-1} - 1\}$ and with the Uniqueness Theorem 4.1.1 we get the existence of $n_k - 1 \geq n'_{k-1} > \ldots > n'_1 \geq 0$. so that:

$$z - \binom{n^*}{k} = \binom{n'_{k-1}}{k-1} + \ldots + \binom{n'_1}{1}$$

entailing a second way of writing $z$

$$z = \binom{n^*}{k} + \binom{n'_{k-1}}{k-1} + \ldots + \binom{n'_1}{1}$$

with $N - 1 \geq n^* > n_k$ and therefore contradicting the Uniqueness Theorem.
Having found position $n_k$ in line 4 the algorithm sets correctly the $i+1$-th 1 in line 5. k gets decremented by one to $k = K - (i+1)$ and again fulfilling $z = \sum_{j=0}^{k} \binom{n_j}{j}$. Summing up, the invariant is true after the $(i+1)$-th iteration. It follows that after the $K$-th iteration of the algorithm we leave the while-loop because $k = K - K = 0$ and having correctly identified the positions of all 1's we return the correctly decoded combination. □

The fifth line of the algorithm does not specify exactly how $n_{max}$ should be found. But because we have discrete and finite search space $[0, n-1]$ in every iteration step, this vague instruction can be implemented easily. One possible way is to search the space linearly, beginning at the upper boundary and decrementing until one $n$ fulfills the condition $z \geq \binom{n}{k}$. Another possible way is to search the space in a binary manner.

**Runtime of Combination Labeling** As the encoding and decoding are used very frequently throughout the calculation of the winning probabilities, it makes sense to give thought to their runtimes. Because the calculation of binomial coefficients is the most expensive operation, we will dedicate the analysis to the counting of them.

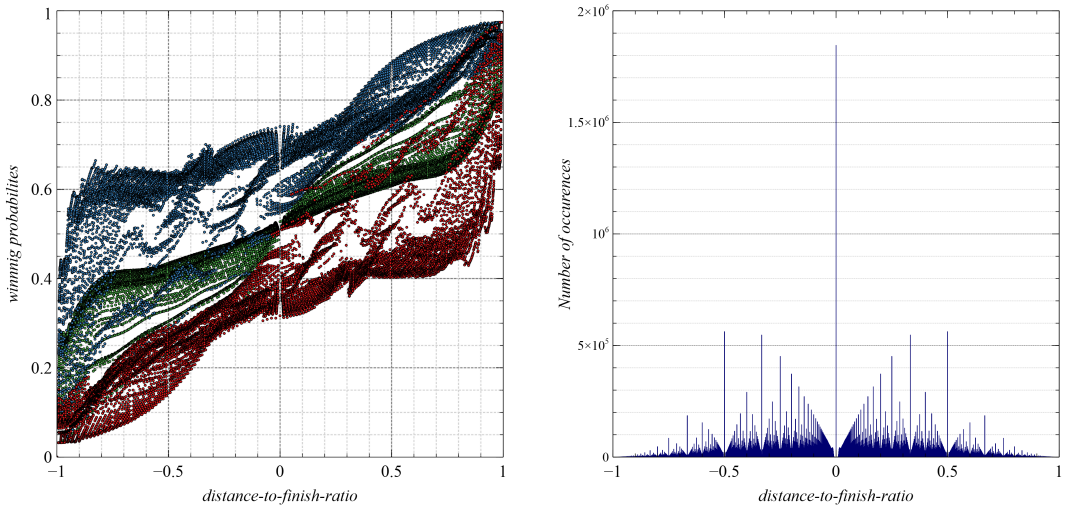
The encoding of a combination in $[0,1]_k^n$ has to solely compute $k$ binomial coefficients, if we are aware of the positions of the 1's.
The decoding of a combination ID is way slower: If we use the linear approach, we will have to iterate through the whole search space in the worst case, computing one binomial coefficient at every element. This gives us $n$ binomial coefficients in total. The binary approach uses $k$ binary searches after the next position of the 1 over the the search space of size $n$. In the worst case we have to compute roughly $k * \log n$ binomial coefficients. If $n$ is significantly larger than $k$, this approach yields faster results. However for small $n$, the linear approach showed to have a slight edge. In our implementation we use the linear approach for piece distributions in the finish zone and the binary approach for the big main field.

## 4.2 Choosing good start values

With the Algorithm 3 and the encode and decode algorithm at hand we already have a functioning tool set to calculate the winning probabilities of MadN, assuming optimal play. The next two sections are dedicated to optimize the implementation to save resources like time and memory. In this section our aim is to restrain the computation time as far as possible. This has already been done by tweaking the algorithm and the encoding and decoding function in all conscience as well as parallelizing the computation to reduce wall-clock time, even when the CPU time cannot be reduced in this way.
Now we take a look at a different aspect. Our method is a fix point iteration in essence. We start with an arbitrary initialized vector $x$ and iterate until it becomes 'close enough' to a fixed vector. Thereby a fixed point iterations convergence speed is heavily influenced by the choice of the start value. Imagine we would initialize $x$ with the correct probabilities. Then the algorithm would terminate after the first iteration, saving us lot of further iterations and computation time. Of course, we have no knowledge about the winning probabilities and calculating those is just as expensive as the initial problem. Instead we are searching for a tradeoff: a heuristic which can be evaluated fast and gives us reasonable good estimates of the actual winning probabilities.
Madn is 'running game': The first player to bring his pieces into his finish zone, wins. In general it can be said, that the player who is fewer moves away from achieving this goal, has the better winning chances. An indicator of how 'close' a player's pieces are to the finish zone can be obtained by counting the total number of squares the pieces still have ahead of them. We call this indicator *distance to finish* and note it for a position $p$ as $d_{max}(p)$ for the Max-player as well as $d_{min}(p)$ for the Min-player. If the Max-player has all his pieces in his finish zone we get $d_{max}(p) = 0$ and if he has all his pieces in the box $d_{max}(p) = (6+43)+(6+42)+(6+41)+(6+40) = 190$. Thereby the parenthesis denote the distance for each piece. Every piece has to be brought out of the box by rolling a 6 and then advanced into the finish zone. To bring both player's distance to goal in relation, we introduce the *distance-to-finish-ratio*

$$r(p) = \begin{cases} \dfrac{d_{min}(p)}{d_{max}(p)} - 1, & \text{if } d_{max}(p) >= d_{min}(p) \\ 1 - \dfrac{d_{max}(p)}{d_{min}(p)}, & \text{if } d_{max}(p) < d_{min}(p) \end{cases} \tag{4.7}$$

This ratio is well defined for all positions $p$, as the case $d_{max}(p) = d_{min}(p) = 0$, meaning that both players have finished their pieces, cannot occur. The ratio has the nice mathematical property of being in the interval $[-1, 1]$. A ratio of 1 implies a win for the Max-player, $-1$ a lose and $r = 0$ implies that both players have the same distance to goal.
We assume a positive correlation between the ratio and the winning probability of the Max-player, but we are lacking data to assign a concrete relation function. This is where the $P$-games

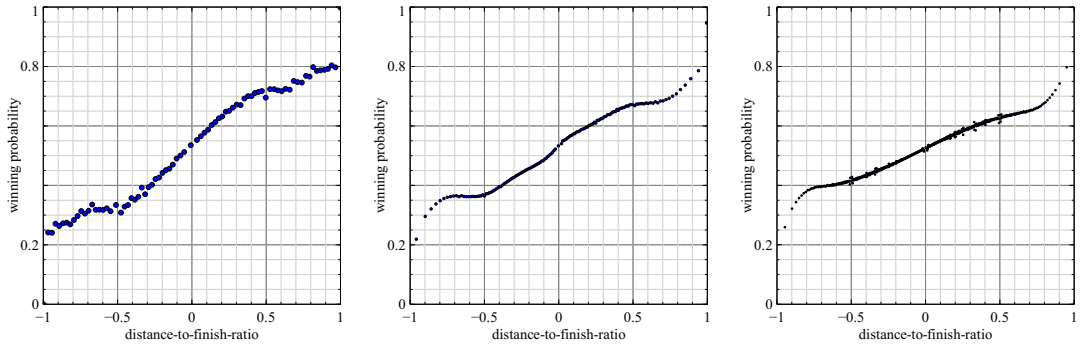



(a) Correlation of ratio and winning probability

(b) Occurences of ratios

Figure 4.3: Analysis of 3-Game ratios. Subfigure (a) shows the correlation of distance-to-finish-ratio and winning probability. Blue dots represent the maximal winning probability, red dots the minimal and green dots the average over all winning probabilities occurring for the ratio. Subfigure (b) shows the how many times the different distance-to-finish ratios occur throughout the possible positions.

come into play which were introduced in the subsection about the number of positions 4.1.1. By reducing the number of pieces per player we generate new games with very equal properties like the standard variant. These new games have significantly less possible positions and therefore less complexity: the fixed point iteration terminates a lot faster for arbitrary starting values. Once we have computed the winning probabilitites for the smaller games, we can analyze the correlation between distance-to-finish-ratio and winning probability. A scatter plot depicting the correlation for the 3-Game can be seen in Figure 4.3a. For every distance-to-finish-ratio which occurred in the possible positions of the 3-Game, we can see three dots in the plot. In blue the maximal, in red the minimal and in green the average winning probability. We can see a positive correlation very clearly. However, the discrepancy between minimal and maximal winning probability for one specific ratio is pretty high and even the averaged probabilities are fluctuating. To generate a smoother curve the ratios of 200 thousand positions were averaged to one data point. The same procedure was done for the 1-Game and 2-Game with different chunk sizes. The results can be seen in Figure 4.4. The averaged winning probability for the distance-to-finish-ratio of 0 is slightly above 50%, which most certainly is due to the first-move advantage of the Max-player. It is also interesting that this advantage is decreasing with increasing complexity of the game (for the 1-Game it is almost at 55%, while for the 3-Game it is roughly at 52%). The rest of the curve seems to be symmetrical to the point at the ratio 0. Therefore, constant values slightly above 50% (or 50%) seem to be good starting values for the non-ending positions. Being a bit more ambitious, we can regress the curve to a polynomial and use its mapping rule to initialize the non-ending positions. Using this procedure, rather than constant values of 50%, we could achieve a reduction in iterations by 7% for the 3-Game using the regression polynomial of the 2-Game.




(a) Correlation for 1-Game (b) Correlation for 2-Game (c) Correlation for 3-Game

Figure 4.4: Scatter plots after averaging ratios for different $P$-Games. For the 1-Game the adjacent ratios of 20 different positions were averaged, for the 2-Game 5 thousand and for the 3-Game 200 thousand.

## 4.3 Compression of Results

Our algorithm manages a huge probability vector with one float entry for every position. These are roughly 16 billion entries for the standard variant of the game. Once the algorithm terminates we have to save the computed results to the hard disk. The most obvious thing to do is to dump the whole probability vector into a binary file. Knowing that a float uses 4 Bytes, we would not need to save additional information about the location of the probability for position, let us say, with ID $i$. Instead we can look up the winning probability at (Byte-)position $i*4$ in the resulting file. The positions of the saved probabilities in the file free us from further overhead. Nevertheless, the resulting file size is about $16000000000 * 4\text{B} = 64\text{GB}$ and already pretty inconvenient. If we are interested in the winning probabilities we have no choice, but to work with a file of this size. However, not all users have to be interested in the winning probabilities. While it is for sure very interesting to see how close one is to the brink of defeat or victory, this information is not necessary to implement an optimal MadN artificial intelligence. It would suffice to merely know the best move at all times. This means that we have to save the best move for every die roll in every possible position. The most promising successor state can be easily extracted from the probability vector (referring back to the $W_i^x(p)$ in the mathematical foundation). Because every player is wielding four pieces, there are only four different possible moves. We can check which of these four moves is transposing to the best successor state: this has to be the optimal move. As there are only four possible moves the best move can be encoded with 2 Bit. For every positions there are six different die rolls possible. In total we have to store $6 * 2 = 12$ Bit of information to know the best move for every position. This is $3/8$ of the space needed when saving the probabilities. In total we arrive at a reduced file size of roughly 24GB.

# Chapter 5

# Results

In the previous section we have described a procedure which is generating a 64GB file containing approximate winning probabilities for roughly 16 billion positions. This section is dedicated to analyze the computed data.

## 5.1 Convergence of Algorithm

We have never formally shown that the Algorithm 3 is converging to a fixed vector. Though, in practice it does seem to converge very well. Referring back to the 'Algorithm and Implementation' chapter 4, we could choose our wanted 'precision' $\epsilon$. Two vector-entries are considered equal when they are not more than $\epsilon$ apart. The Fixed Point Iteration terminates whenever it executes one iteration and all entries of the vector are considered equal to the previous iteration. The result is 'almost' a fixed vector. For $\epsilon = 10^{-6}$ the algorithm terminates for the 1-Game in 39 iterations, for the 2-Game in 67 iterations and for the 3-Game in 105 iterations, when the start vector is initialized with 0.5 for every non-ending position. Up until this point the number of iterations seems to increase linearly, roughly by 30-40. Extrapolating this scheme to the standard variant, the 4-Game, we arrive at a maximum of 145 iterations. Considering the tweaks proposed in the section 'Choosing Good Start Values', we are additionally hoping for a decrease of roughly 7% in iterations, by using the distance-to-finish-ratio data obtained through the 3-Game. This gets us to the estimate of about 135 iterations and surprisingly the exact number of iterations until termination. The convergence graph, showing the number of changes per iteration, can be seen in Figure 5.1. Thereby a change is an entry which changed by more than $\epsilon$ from one to the next iteration. Starting at around iteration 60 the number of changes begins to drop rapidly, until iteration 105. From then on only few position probabilities change significantly until the termination of the algorithm. It is arguable whether the number of changes is a good indicator for convergence since it might be the case that the fixed point iteration is going to drift slowly away from the 'almost'-fixed vector, always by less than $\epsilon$ per entry in every iteration. While bounding the probabilities is certainly an important piece of future work, it seems unlikely that the results will change significantly by further iterations: it can be formally shown that the maximal entry-change per iteration is monotonically decreasing, meaning that once all entries changed by less than $\epsilon$ they will continue to do so for every subsequent iteration. This means that it requires 10 000 iterations to alter the computed results by 1% if the change manages to stay roughly constant at around $10^{-6}$ per iteration. That would be shockingly surprising regarding the fact, that the changes dropped to $\epsilon$ in only 135 iterations.

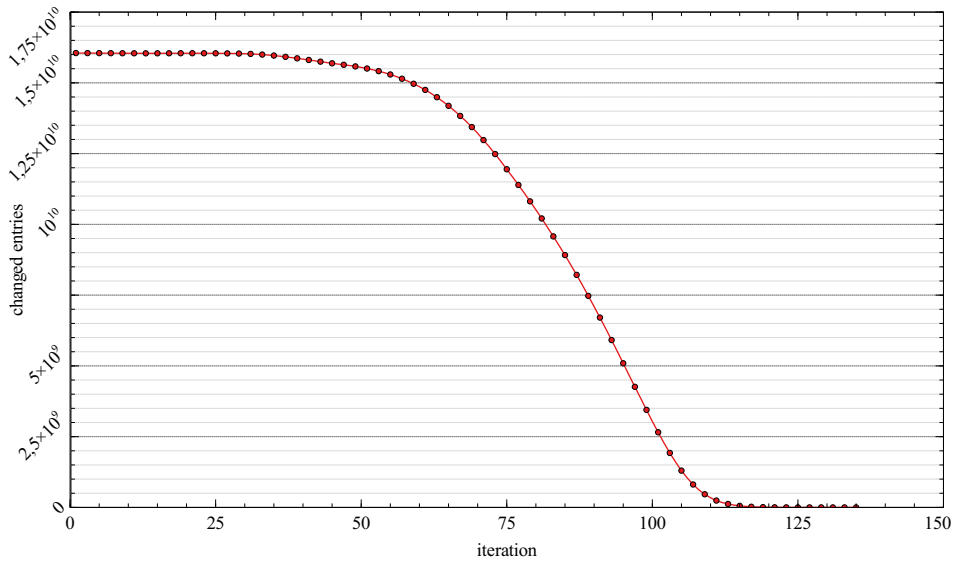



Figure 5.1: Convergence Graph showing the number of changed entries (change $> \epsilon$) for every iteration

| | Min | Max | Start |
|---|---|---|---|
| 1-Game | 8.3158 | 93.3474 | 53.0736 |
| 2-Game | 5.4906 | 95.6076 | 50.9187 |
| 3-Game | 3.2250 | 97.4200 | 50.7843 |
| 4-Game | 1.2579 | 98.9937 | 50.7538 |

Table 5.1: Winning probabilities in percent for the different $P$-games. The 'Start'-column describes the winning probabilities for the starting position from the view of the starting player, Min and Max the minimal and maximal winning probabilities one can have over all positions with being the next player to move.

### 5.1.1 Computation Time

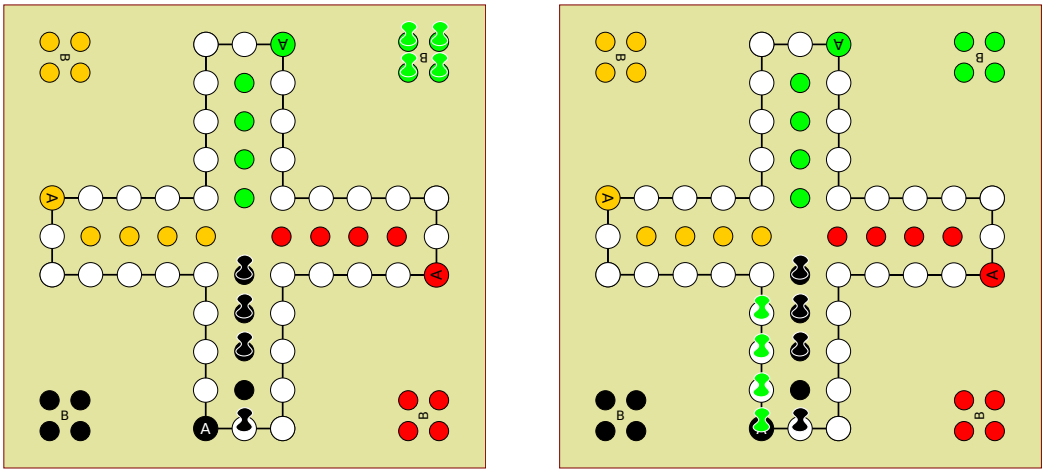
The computation has been done on a Intel Skylake Platinum 8160 processor with a clock rate of 2.1Ghz, which was helpfully provided by the RWTH Aachen, in parallel using 48 cores. With roughly 1 hour per iteration and additional time to backup the results once in a while, the algorithm terminated after roughly 140 hours of computing. It is interesting to point out that the same setup for the 3-Game needed only about 1.5 hours wall-clock time. Even when the number of iterations is increasing linearly from $P$-Game to $(P+1)$-Game, the number of positions is increasing exponentially leaving us with a very bad scalability. Calculating the winning probabilities for the 5-Game would have not been possible (in feasible time) with the procedure described in this work and/or the hardware at hand.

## 5.2 Interesting probabilities

According to the winning probabilities for the starting position, which can be seen in Table 5.1 MadN seems to be an unbalanced game between both players. For every $P$-Game the player to take the first turn has a slightly higher winning probability than his opponent. This is not especially surprising since the first player will always be one move ahead in bringing his pieces into the finish zone. It is interesting how the first-move advantage is shrinking with increasing number of pieces. This makes sense, because the more pieces are in the Game, the more moves have to be played to determine the end result. Thus, the influence of the one move head start gets smaller regarding the amount of moves played.
Next to the winning probabilities for the starting position we can see the maximal and minimal winning probabilities a player can have when being next to move. In other words: this is how close one can get to the brink of defeat or victory, without actually losing or winning. It is the best or worst position, which is not lost or won yet. This is almost true. Of course the 'worst position' can get even worse when the (almost lost) player has not even the privilege of being the next player to move. In this case the winning probability is the complement to the maximal winning probability in the Max-column.
The positions, for which the probabilities reach their maximum/minimum are depicted in Figure 5.2. If the reader is interested he can try to guess the worst or best possible position for the different $P$-Games; he might end up surprised. As the game can be understood as a race between both players, it would be natural that the biggest imbalance occurs, when one player has almost finished all his pieces, while the opponent has not made any progress towards his goal yet. Such a position is depicted in Figure 5.2a for the 4-Game. For the $P$-Games with $P \in \{1, 2, 3\}$ the equivalent is rather obvious: The disadvantaged player has his $P$ pieces in the box while the advantaged player has $P - 1$ pieces in the finish zone and one piece waiting for the winning die roll right in front of the finish zone. Indeed such a position is the worst possible from the view of the player wielding the green pieces for the 1-, 2- and 3-Game. Surprisingly this is not the case for the standard variant of the game. The most unbalanced position can be seen in 5.2b. Even though the player wielding the green pieces has been gifted roughly half of his needed distance to the finish zone, his situation got even more helpless. The reason for that is quite simple: because all green pieces have passed the black finish zone, they will not be able to capture the remaining black piece. The only 'threat' for black is that the green player will finish all his pieces first. In the 4-Game, unlike the other $P$-Games with $P < 4$, the threat of being captured and as a result lose the game is bigger than the threat that the opponent finishes his pieces first, even when let half of the distance off.

## 5.3 Games against simple AIs

It would be interesting to examine how well a human player is performing against the optimal player. Because of the high factor of chance, reliable results require a huge amount of played games. Playing thousands and thousands of games against the optimal AI is too time consuming. Building an AI which is representing oneself and subsequently simulate games against the optimal AI is a better option. In the best case one would determine one's playing policy, like described in the Mathematical Foundation, i.e. choosing the desired successor for every position for every die roll. Unfortunately, because of the 16 billion different positions, this is pretty inconvenient. As remedy, we will approximate 'human behavior' through simple playing policies, which are based on own playing experiences. All AIs, which are going to be introduced subsequently, have been pitted against each other. For every pairing there were two matches played, each match containing of 1 million games. In the first match one AI was granted the first-move advantage,



(a) Best/Worst position for 1-Game, 2-Game and 3-Game

(b) Best/Worst position for the standard variant of Madn

Figure 5.2: Worst and best possible positions of MadN. Player wielding the black pieces is at advantage over the player wielding the green pieces.

| | Optimal | Random | FPoC | FPnPoC |
|---|---|---|---|---|
| Optimal | X | 87.9516 | 69.3913 | 64.9809 |
| Random | 12.6015 | X | 21.8489 | 17.9132 |
| FPoC | 31.9699 | 79.1810 | X | 46.3623 |
| FPnPoC | 36.5915 | 83.0041 | 55.2596 | X |

Table 5.2: Pitting different AIs against each other. Every entry of the table is the result of 1 million games. The entry $p$ at position $i, j$ says that AI $i$ with the first-move advantage won against AI $j$ in $p$ percent of the games.

in the second the other. The results of all pairings can be seen in the cross-table 5.2 .

### 5.3.1 Furthest Piece or Capture

A very simple move rule is to capture an opponent's piece whenever possible and, otherwise, move the furthest piece (closest to finish zone), to bring it as fast as possible in the finish zone, saving it from opponent's captures. If there are two possible captures, then the furthest piece should capture. We will refer to this playing policy as *Furthest Piece or Capture*(FPoC). Capturing a piece can hardly be a bad move, as it throws back the opponent in his game progress, thus worsens his position and, because of the zero-sum property, improves one's own. Against a random player, which chooses his move out evenly distributed among all possibilities, FPoC shows a healthy edge by scoring roughly 79% when having the first-move advantage and 78% when not. Pitting FPoC and the optimal AI against each other, results in a clear disadvantage of FPoC, scoring a win only 32 percent of times with first-move advantage and 30 percent without.

### 5.3.2 Furthest Piece no Passing or Capture

When not able to capture, moving the furthest piece is very straightforward, easy to implement and fast to apply. Nevertheless, this is a vulnerability: staying 'behind' the opponent's pieces is often, speaking out of own experience, a very good strategy. Passing an opponent's piece gives the passed piece the opportunity to capture the passer in the next move(s). Instead, staying behind turns the tables: the opponent's piece cannot capture one's stood back piece and with a bit of luck the next die roll(s) will allow to capture the opponent's piece. Following this line of thought we try to improve FPoC by not passing an opponent's piece, when possible. Summing up the playing strategy of *First Piece no Passing or Capture*(FPnPoC): The first priority is to capture an opponent's piece, the second priority is to pass as few opponent's pieces as possible and the third priority is to move the furthest piece on the play-field. Indeed, FPnPoC turns out to be stronger than FPoC. Against the random player it scores better, with and without the first-move advantage. In the direct encounter with FPoC it maintains a slight edge winning about 55% of the games with the first-move advantage and 54% without. Still, against the optimal player it scores poorly: only 36.5% of the games can be won with the advantage of the first-move.

# Chapter 6

# Conclusion

MadN is a very simple game. The rules can be explained in a few minutes and the number of possible actions during the game as well as the number of conceivable decision moments is very limited. This made it feasible for us to tackle the game in a very perfectionist way: solving it once and for all. 'Solving' can be interpreted in many different ways. Common practice is to find two playing strategies which are in a Nash-Equilibrium. Then the playing strategies can be regarded as optimal since both players' winning probability can only decrease by a strategy deviation, if the opponent sticks to his playing strategy. In the mathematical section of this work, we showed that the search for a pure strategy equilibrium breaks down to the search for an eigenvector to the eigenvalue of 1 for a specific stochastic transition matrix. Additionally the entries of the eigenvector describe the winning probabilities for all positions, assuming the players stick to the optimal strategies. The eigenvector can also be regarded as a fixed vector, which led us to the idea of a fixed point iteration to put the theory into practice and compute the probability vector. After some refinements of the naive fixed vector iteration, we resulted in a feasible implementation, which terminated in 140 hours wall-clock time and provided us with (most likely) good approximations of the actual winning probabilities.
Because of the simplicity, even first graders are able to play the game and score well against adult players. This is why it is hard to believe that the game cannot be played (close to) optimally by following a handful of simple rules. Nevertheless the strongest AI we constructed which was following simple policies, is a 2:1 underdog against the optimal, winning only roughly one third of the games.

The work is titled "Mensch Ärgere Dich nicht" and dedicated to the analysis of the game. Computing the winning probabilities and optimal playing strategies for MadN takes only the first step. There are a lot of thinkable ways to proceed the work, reaching from proving the correctness of the calculated winning probabilities (or disprove when they are false) to a more thorough refinement of humanly understandable playing strategies which can compete with the optimal player. The latter would be a blessing for family games nights: When, by chance, the impudent little sibling captures your piece again right before it reached its finish zone and starts gloating all over his face, you don't have to lose your temper blaming your own decisions, but instead keep a cool head with the clear conscience that you have played optimally and could have not done any better. In a way this work helps to translate the request into action, which forms the title of the game: "Don't Worry!"

# Bibliography


[1] John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[2] John Nash. Non-cooperative games. *Annals of mathematics*, pages 286–295, 1951.

[3] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[4] Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015.

[5] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[6] Schmidt Spiele. Spielanleitung mensch ärgere dich nicht.

[7] Charles Miller Grinstead and James Laurie Snell. *Introduction to probability*. American Mathematical Soc., 2012.

[8] Hans Peters. *Game theory: A Multi-leveled approach*. Springer, 2015.

[9] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.

[10] Lloyd S Shapley. Stochastic games. *Proceedings of the national academy of sciences*, 39(10):1095–1100, 1953.