Bianca Beatrice TEUFL , BSc

Deterministic Asymmetric Two-Player-Games with Perfect Information

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Assoc.Prof. Dipl.-Ing. Dr.techn. Oswin Aichholzer

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, April 2018

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

Abstract

Games have been an integral part of many people's lives for thousands of years. Long before the printing press was invented, people developed games. The rules of these games were passed on from person to person. As they were told and retold, they have accumulated changes. This is one of the reasons why some games have many different variations. These different variations are especially interesting in asymmetric two-player games. These are games in which the two players may have different game goals and therefore different strategies to reach them.

This thesis is about asymmetric two-player games with perfect information, using the example of the games "Fuchs und Henne" and "Fox and Geese". We describe these two games in detail and present some possible variations. Then we consider an upper bound of the state space of these two games. Moreover, we develop an encoding and decoding process for strongly solving the games' variations using a `C++` framework and we implement the games in this framework. This allows us to play the games and some variations of them in a simple console application.

We show how to use the results of the different games' variations to compare them based on the number of their winning and losing states. This information can be used to find a fairer variation of the games.

Kurzfassung

Seit tausenden Jahren spielen Spiele eine wesentliche Rolle im Leben vieler Menschen. Schon lange bevor die Druckerpresse erfunden wurde, entwickelten Menschen Spiele. Die Regeln dieser Spiele wurden von Mensch zu Mensch weitergegeben, indem sie erzählt und weitererzählt wurden. Durch die verbale Kommunikation veränderten sich die Beschreibungen. Das ist einer der Gründe, warum von manchen Spielen so viele unterschiedliche Varianten existieren.

Diese Varianten sind bei asymmetrischen Zwei-Spieler-Spielen besonders interessant. Das sind Spiele, in denen die beiden Spieler unterschiedliche Spielziele verfolgen und daher verschiedene Strategien haben, diese Ziele zu erreichen.

In dieser Arbeit behandeln wir asymmetrische Zwei-Spieler-Spiele mit perfekter Information am Beispiel der Spiele “Fuchs und Henne” und “Fox and Geese”. Wir beschreiben diese Spiele im Detail, stellen einige mögliche Varianten vor und bilden eine obere Schranke für ihren Zustandsraum. Außerdem entwickeln wir einen Codierungs- und Decodierungsprozess, den wir nutzen, um eine starke Lösung für die Varianten der Spiele mit Hilfe eines `C++` Frameworks zu finden. Wir implementieren die beiden Spiele in diesem Framework. Das ermöglicht es uns, die Spiele und einige ihrer Varianten in einer einfachen Konsolenanwendung zu spielen.

Wir zeigen, wie wir die Ergebnisse der verschiedenen Spielvarianten nutzen können, um sie anhand der Anzahl ihrer Gewinn- und Verlustzustände zu vergleichen. Diese Information kann verwendet werden, um eine möglichst faire Variante der Spiele zu finden.

Contents

List of Figures

List of Tables

1 Introduction

In this chapter we describe the key terms used for this thesis. This will provide us with a common basis for understanding the topic. Afterwards, we introduce the idea for this thesis and the main goal of our work. In the end of the chapter, we deal with the structure of this document and outline the contents of the following chapters briefly.

1.1 Terms

1.1.1 Perfect Information

Within games we talk about perfect information, if the following conditions are present:

- The player knows which actions and moves have been done so far, up to the actual point of the game
- No other player is allowed to move while one player makes his/her decision.

If one of this conditions is not fulfilled, we talk about imperfect information. If all players of a game have perfect information, we call the game a game with perfect information [18].

1.1.2 Determinism in Games

We call a game a deterministic game, if the produced result after an equal move of a player at a given game situation is the same at any time without any exception. In contrast, we call a game non-deterministic, if the result of two identical game situations can be different without taking the move of the player into account. This means the game is not predictable. [1]

1.1.3 Combinatorial Games

Combinatorial games are typically deterministic two-player games with well-defined rules in which the players take alternate turns. As both players have all information about the game as described in Section 1.1.1, combinatorial games are games with perfect information. [5]

Well-known games of this category are for example:

- Tic Tac Toe
- Chess
- Chomp
- Connect 4
- Fuchs und Henne
- Fox and Geese

In combinatorial games either one player wins and the other one loses or the game results in a draw. These games start at a defined initial state. Then the players take alternate turns and take their moves depending on their individual decisions until they reach the end of the game [5].

1.1.4 Symmetric and Asymmetric Games

Symmetric games are games in which all players have the same game goals and therefore the same set of strategies. The pay-off is a result of the played strategy, not of who is the player [4]. Examples for symmetric games are “Tic Tac Toe”, “Nine Men’s Morris” and “Connect Four”.

Asymmetric two-player games are games where the two players may have different game goals and therefore different strategies to reach them. As there are different goals for the two players, there are also different rules and strategies. The initial position for the two players may be the same, but can also differ from each other.

Well-known games with these characteristics are “Fuchs und Henne”, “Fox and Geese” and “Tablut”.

There are also asymmetric games in which the rules and strategies are the same for both players, but the initial setting of the game is different for the two players. For example, in chess both players are determined to reach the same goal and have the same set of strategies. However, the starting position is not symmetric, because both queens are placed on the D-line of the game board. This means that the white queen is placed left to her king and the black one on the right side of her king [13, p. 29].

1.2 Motivation

People have developed games long before the printing press was invented. At that time, rules and instructions of the games were passed on from family to family and from generation to generation. In addition, games spread across the world. This was favoured by the introduction of trade. However, this is not the only reason why there are so many different variations for some of the games. We adapt especially asymmetric games very often. One reason for this is certainly the attempt to change the difficulty for each player. We often perceive asymmetric games as unfair. We introduce a way to find a variation that is as fair as possible within this thesis. We do this by developing a way to compare variations of the games “Fuchs und Henne” and “Fox and Geese” by strongly solving them (we will describe this term in the next chapter).

1.3 Objective

In this work we look at deterministic asymmetric two-player games with perfect information. The thesis is organized as follows: In Chapter 2 we look at the history of asymmetric two-player games. We describe the games “Fuchs und Henne” and “Fox and Geese” in detail. In Chapter 3 we discuss some game theoretical aspects, especially solving games. We also look at the `C++` framework which we use to solve “Fuchs und Henne” and “Fox and Geese”. In Chapter 4 we consider the state space of the game “Fuchs und Henne”. The encoding of states of this game is described in Chapter 5. In Chapter 6 we consider the state space of the very similar game “Fox and Geese”. We describe the encoding of the states of this game in Chapter 7. Chapter 8 is about the variations of the games “Fuchs und Henne” and “Fox and Geese”. Finally, we compare the calculated upper bound of states with the actual number of states of the variations.

In this work we use the framework described in Chapter 3.4 to strongly solve (we describe this term in the next chapter) the games “Fuchs und Henne” and “Fox and Geese”. For this we develop an encoding method in order to be able to store the states of the games in a database with as few bits as possible.

With the results we try to find a way to create a variation that might be fairer than the well-known ones.

2 ’‘Tafl’‘ Games

In this chapter we will look at the history of games, especially of so-called “Tafl” Games. This is a category of deterministic asymmetric two-player games with perfect information. We will have a closer look at two well-known games of this category: “Fuchs und Henne” and “Fox and Geese”.

2.1 History

All over the world human have been playing board games for thousands of years. Different cultures invented board games, writing different aspects of the history of the games. That’s why the history of games is such a big and considerable topic. People all over the world had the idea to create miniature battles, races, hunts and so on to have fun. Probably they were also able to learn for the real life through them. Some games stayed original in one culture while others arose through trade and other contacts from others and so changed in different places. They were influenced by trade, wars and also by science. Before the invention of the printing press, the rules of games were passed on through verbal communication. This created many different variations. The revolution of the industry not only increased the popularity of board games, but also unified the rules of modern games to some extent [12].

Some asymmetric two-player games are known under the name of “Tafl” Games. That’s a family of strategy board games originated in Germania and from the Celts [9]. The main idea of all these games is that a large number of attackers fights against an in relation very small number of defenders [7]. In detail “Tafl” games are battle games between two forces of uneven numbers: a smaller force, which is equipped with special power and a larger force. The goal of the smaller force is to destroy the other one or to break out, while the larger force tries to lock in the opponent of war. These games appeared first in northern Europe [3].

The conditions of the game boards of this game family vary from game to game. “Hnefatafl”, a battle game between the king and his army fighting against his attackers, gets played on a field of size 11 x 11. “Tablut” is a variation of this game. It gets played on a board of size 9 x 9 [14]. The game “Halatafl” gets played on a board of size 7 x 7 with two enemies of size 22. Although “Halatafl” is a game with two equal enemies, it is sometimes referred to as a version of “Fox and Geese” that was already known in the Middle Ages. We will have a closer look on “Fox and Geese” in Section 2.3. The game is known at least since the 14th century where a game called “Halatafl” was mentioned in the Grettis Saga - one of the Icelander’s sagas. It is described there as a board game where dolls with nails can be inserted into a board [11].

Even today “Tafl”-Games are very popular, especially “Fox and Geese” and “Fuchs und Henne” which are included in many game collections. We will describe these games in detail in the following sections.

2.2 Rules of “Fuchs und Henne”

For the asymmetric two-player game “Fuchs und Henne” (German for fox and hen) we can find many different descriptions and rules. The reason for this is probably the fact that the idea of this game was created a long time ago, most likely somewhere in Iceland. It was passed to many different regions as we described in the beginning of this chapter. We will look at the different rules and versions in the following sections to get an overview of them.

2.2.1 The Game Board

“Fuchs und Henne” gets played on a game board with 33 fields which are arranged in the form of a cross as shown in Figure 2.1. We use the same game board also for other games like “Fox and Geese” (see Section 2.3), “Solitaire”, “Raubritter” and many others. Originally, this board usually was made of wood [3]. Nowadays we find it in almost every game collection, probably because we can use it in so many different ways.

On this board there are horizontal and vertical lines between all playing fields. Between some fields there are also diagonally lines. We will describe the meaning of them for the game “Fuchs und Henne” in Section 2.2.3.

2.2.2 Starting Point

The most common version of “Fuchs und Henne” gets played with 20 hens and 2 foxes on the board described in Section 2.2.1. In Figure 2.2 we see the starting position of the game in this version. We represent the hens by blue filled circles and the foxes by yellow filled circles. We display the empty fields in grey. The red surrounded circles represent the hen house, we will explain its meaning later

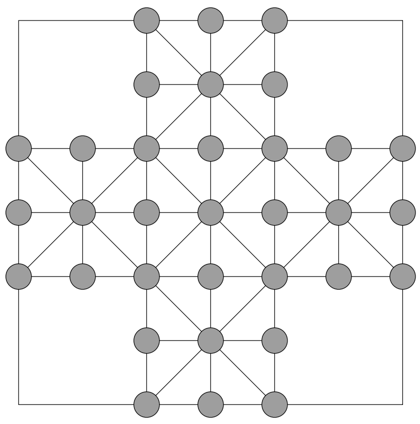


Figure 2.1: The game board of “Fuchs und Henne”

in this chapter. This coloured marker on the board is not always available. The reason is that this game field can, as described in Section 2.2.1, also be used for other games that do not require a hen house. The hen house has always a size of 3 x 3 and is placed on the side of the board where the fox player is positioned.

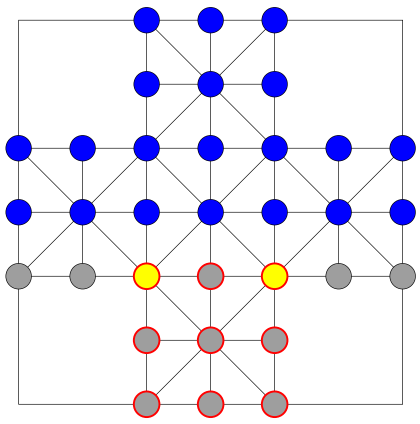


Figure 2.2: The start position of the most common version of “Fuchs und Henne”

One player plays the part of the hens, we call him/her the hen player. The other one plays the foxes, we call him/her the fox player. The players take alternate turns, whereupon the hen player opens the game.

In other versions of “Fuchs und Henne” the number of hens varies. The reason is mostly the fact, that somebody wants to modify the difficulty of the game. But also different traditions of the game can be the reason as we already mentioned above. If we play the game with 17 hens, the start arrangement of the tiles is mostly like we see in Figure 2.3. Sometimes we play the game with just one fox. We see the initial state of this version in Figure 2.4. Of course, it is possible to vary the number of hens at will, and so we can change the difficulty for both players.

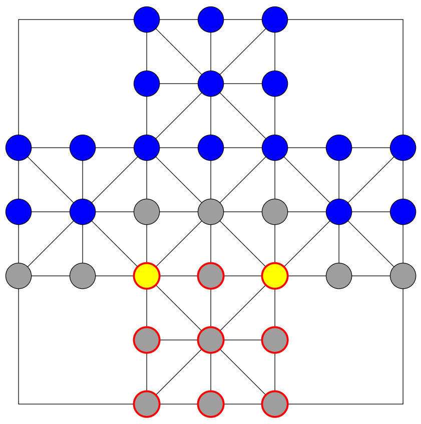


Figure 2.3: The start position of “Fuchs und Henne” with 17 hens

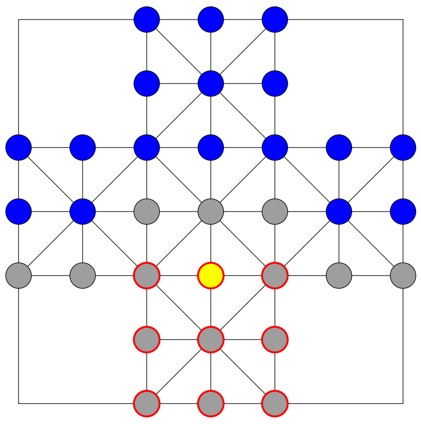


Figure 2.4: The start position of “Fuchs und Henne” with 17 hens and one fox

2.2.3 Allowed Moves

In the most common version of "Fuchs und Henne" the hens are allowed to step with a step size of one to an empty field forward or sideways but never back. The hen player is never allowed to move diagonally.

The fox player moves after the hen player and is allowed to move to an empty field forward, sideways and also backwards with a step size of one.

In some variations of the game the foxes are allowed to move diagonally. Sometimes just at the diagonal lines drawn on the game board as shown in Figure 2.1, sometimes at any possible diagonal.

Additionally, the foxes have to jump over a hen, if the field directly after the hen is empty. That at the same time means, that this hen is beaten and gets removed from the game field. It can not be rescued anymore. If the fox player ignores or overlooks the possibility to jump over a hen, the hen player is allowed to remove one fox from the game field. If the fox player beats the next hen, the second fox is rescued. Then the hen player is allowed to place the fox to any empty field on the game field [19]. In some variations we do not have this constraint to beat a hen if possible and so also the removal of the fox does not happen.

In some versions multiple jumps are allowed. That means if the fox jumped over a hen (and beat it), it can, if it is possible, jump over another hen (which also gets beaten) as long as it is possible [8]. Some variations of the game do not allow multiple jumps. In others the fox player has to jump over hens as long as possible and so beat as many hens he/she can in this one move.

The fox player can not beat a hen that is already placed on a field within the hen house.

2.2.4 End of the Game

The player who plays the hens wins, if he/she is able to carry 9 of his tiles into the hen house. This is located on the opposite of him/her on the game board. The hen house is characterised by the red outlines in Figure 2.2. If the fox player is not able to move anymore, the hen player also wins. Additionally, the hen player wins, if there are no foxes left on the game field. This can happen, if the fox player ignores or overlooks the possibility to beat a hen twice, before he/she beats a hen to get one fox back, as described above.

If the fox player beats so many hens that less than 9 are left he/she wins, because the hen house can not be filled up with hens anymore. He/she also wins, if no hen is able to move. Some situations where the hen player is not able to move anymore are shown in Figure 2.5.

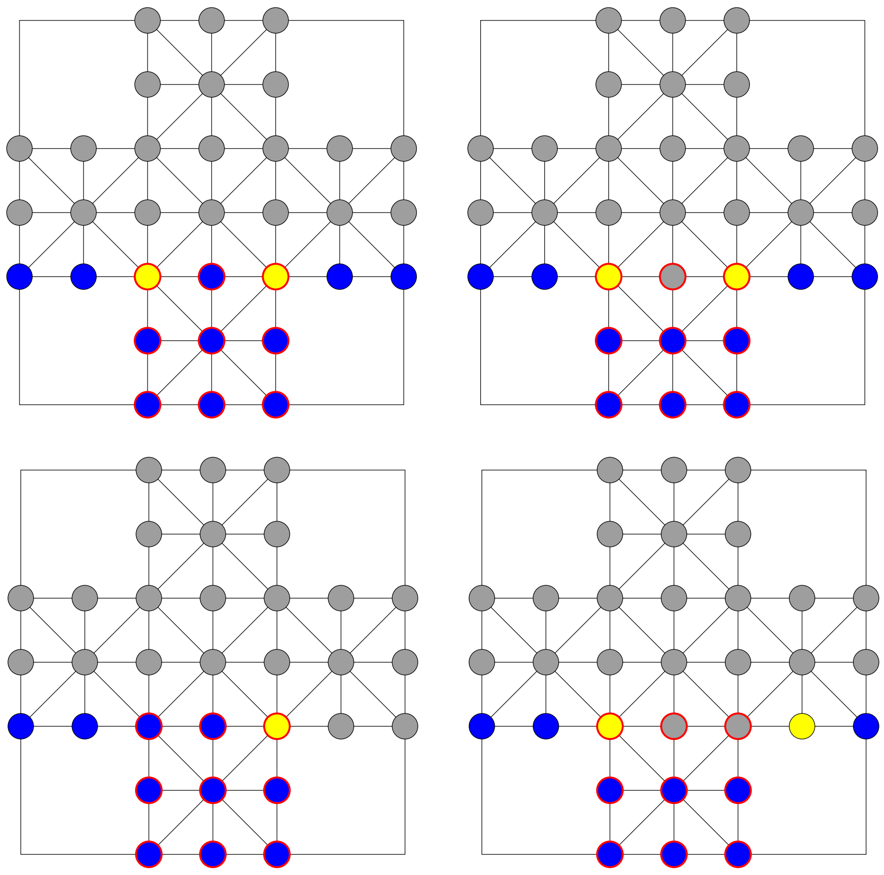


Figure 2.5: Situations of “Fuchs und Henne” in which the hens are not able to move

2.2.5 Selected Version

For the further work we selected the following set of rules:

- 20 hens
- 2 foxes
- Hens are allowed to move forward and sideways but never back (and never diagonally)
- Foxes are allowed to move one step in all directions and also over diagonals but just if there is a line between the fields
- Foxes have to beat if possible by jumping over a hen, otherwise a fox gets removed
- Foxes can jump over hens in all directions they are also allowed to move to
- Multiple jumps are allowed
- Hens in the hen house are not allowed to be beaten by the fox

2.3 Rules of “Fox and Geese”

“Fox and Geese” is also an asymmetric two-player game which we can play on the same game board as “Fuchs und Henne” (see Chapter 2.2).

In this game one player plays the fox figure, while the other one plays the goose figures.

2.3.1 The Game Board

The game board of “Fox and Geese” is similar to the game board described in Section 2.2.1 but we need no hen house. This is because the goal of the game is different as we will describe in Section 2.3.4

2.3.2 Starting Point

The main version of “Fox and Geese” is played with 15 geese and one fox. In Figure 2.6 we show the starting point in this version of the game. The green circles represent the geese and the yellow circle is the fox. The grey circles stand for empty fields on the game board. The players take alternate turns. In this game the geese player opens the game.

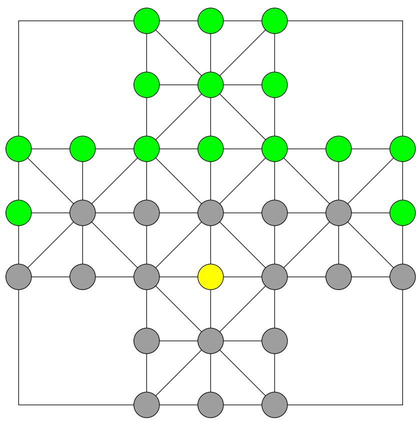


Figure 2.6: The start position of the main version of “Fox and Geese”

To alter the difficulty we sometimes reduce the number of geese to 13, and by doing so we increase the difficulty for the geese player. Sometimes we increase the number of geese to 17 or 20 for a better balance. Figure 2.7 shows the setup for 13 geese, Figure 2.8 for 17 geese and Figure 2.9 for 20 geese.

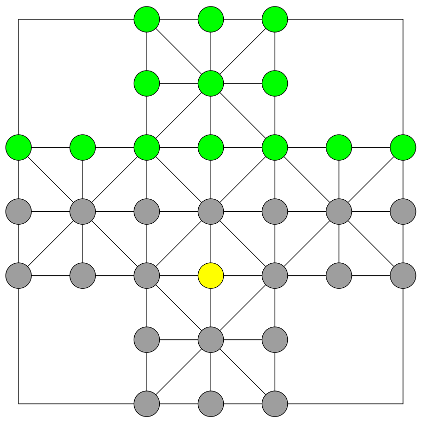


Figure 2.7: The start position of the variation of "Fox and Geese" with 13 geese

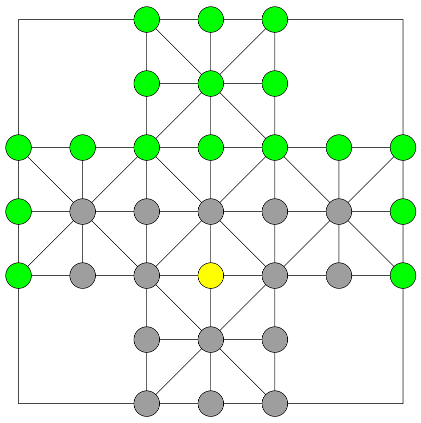


Figure 2.8: The start position of the variation of "Fox and Geese" with 17 geese

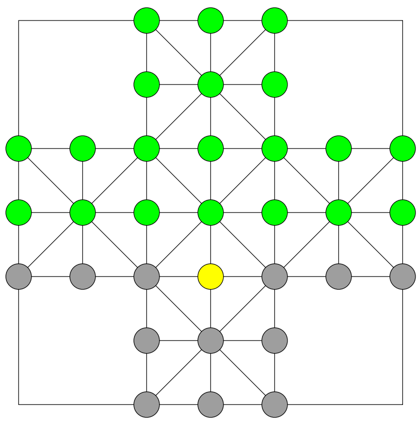


Figure 2.9: The start position of the variation of “Fox and Geese” with 20 geese

2.3.3 Allowed Moves

The fox player is allowed to move the fox to empty spaces all around the fox - forward, sideways, back and diagonally on any diagonal. Additionally the fox is allowed to jump over a goose in all directions, if the field directly behind the goose is empty. This means that the goose which has been jumped over is beaten. The fox does not have to jump. Multiple jumps are allowed. The geese player is allowed to move forward, sideways and diagonally forward but never back [6].

One variation of the game is that it is only allowed to use the drawn diagonals. There are some variations with and without multiple jumps. Another variation is that the geese player is not allowed to move diagonally.

2.3.4 End of the Game

The geese player wins, if the geese surround the fox in a way that it is not able to move anymore. That means, that the fox player wins, if there are not enough geese in the game to surround the fox.

To surround the fox 6 geese can be sufficient as shown in Figure 2.10. In the variation where only the limited number of drawn diagonals is allowed, 4 geese can be sufficient to surround the fox, as shown in Figure 2.11.

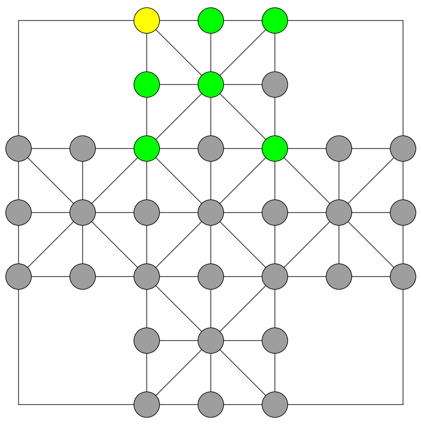


Figure 2.10: Situation in which 6 geese surround the fox

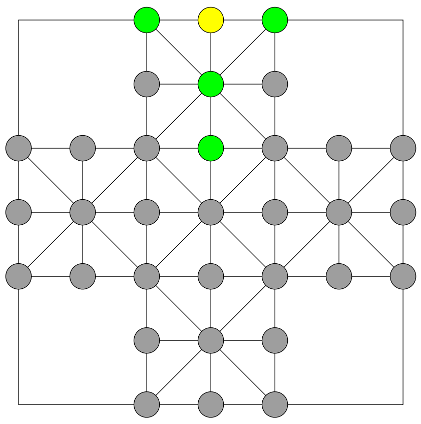


Figure 2.11: Situation in which 4 geese surround the fox in the variation with limited diagonals

2.3.5 Selected Version

For the further work we selected the following set of rules:

- 15 geese
- 1 fox
- Geese are allowed to move forward and sideways and diagonally forward, but never back
- The fox is allowed to move one step in all directions including all diagonals

- The fox may beat if possible by jumping over a goose but that is not a must
- The fox can jump over geese in all directions in which it is also allowed to move
- Multiple jumps are allowed

In this chapter we had a look at the history of table games. Especially on a category of deterministic asymmetric two-player games with perfect information - "Tafl"-Games. We described two games of this category in detail: "Fuchs und Henne" and "Fox and Geese". We looked at different versions of this games and we selected one version each for the further work.

In the next chapter we will discuss a branch of game theory: solving games. In particular we deal with the different levels of solving games. For this we need to know the terms game tree and state space and how to use symmetries. We also look at the complexity of games. The last topic of this following chapter is the description of a framework to strongly solve combinatorial games.

3 Solving Games

In this chapter we will look at a branch of game theory - solving games. At first we will describe the different levels of solving games. Additionally, we will consider the terms game tree and state space and how to use symmetries. Then we look at the complexity of games. The last topic of this chapter is the description of a framework that we can use to solve combinatorial games.

3.1 Solving Games

When we talk about solving games we know at least three definitions: Ultra-weakly solved, weakly solved and strongly solved.

Ultra-weakly solved means that we know the result of the initial state(s) - that means if the moving player has a winning strategy, the opposing player has a winning strategy or no player has a winning strategy for the initial state given on perfect play on both sides.

A game is weakly solved, if there is a sure strategy for the player who will win the game or for both if it results in a draw. That means, that we know at least the moves from the initial state to the end of an ideal game and it is proofed to be optimal.

Strongly solved means that it is possible to know a winning move from any given position of the game, no matter if there have been made bad decisions so far. In other words we know the full classification (winning, losing or draw) of all reachable states [10].

When we solve games, the goal is always to solve them strongly.

3.2 The Game Tree and the State Space

In graph theory a tree is an undirected graph that is connected and acyclic. In the game theory we use trees to build the game tree of a game. Game trees have game positions as nodes and plies as edges. In game theory we use the term ply for the move of just one player while a move is the sum of the plies of both players [16]. To build a game tree we use the initial state as the root of the tree. Then we add all next states of this initial state as child nodes. For each of this new nodes we determine the following states within the game and again add them to the tree and so on. A node which has no following nodes is a leaf of the tree and represents a final state of the game. We repeat this tree building until we have found all leaves.

Figure 3.1 shows the building of the game tree of the Game “Tic Tac Toe” within the first ply of each player, with removed similar states.

There are some characteristics of games that we can directly gather from such game trees: The branching factor of a state shows out of how many different possibilities the current player has to choose his/her move (ply). The average branching factor of a game indicates how many possible plies are available on average. Depending on the complexity of games it might not be possible to find a formula to fully calculate the average branching factor of a game. That is because there are sometimes too many different states to enumerate all of them. Then the number gets approximated. We also use this factor to estimate the size of the game tree. Additionally, we can use the game tree to calculate the game-tree complexity and the state-space complexity of a game what we will describe in Section 3.3 [16].

We call the set of all reachable states of a game the state space. Reachable means that it can be produced when playing the game starting at its initial state. When we transform the game field by operations like reflection or rotation for some games it results in equal states. Because of these symmetries, the state space can be reduced when we consider just one of those equal states. We do this in the following subsection.

3.2.1 Reducing the State Space

As already mentioned above, the state space of a game can be reduced by taking advantage of the symmetries of the game. For example, there is a horizontal, a vertical and two diagonal axes of symmetry on many game boards. When we apply these operations to a state of the game, the resulting states are equivalent to the origin state. We can also combine the operations or use them more than once.

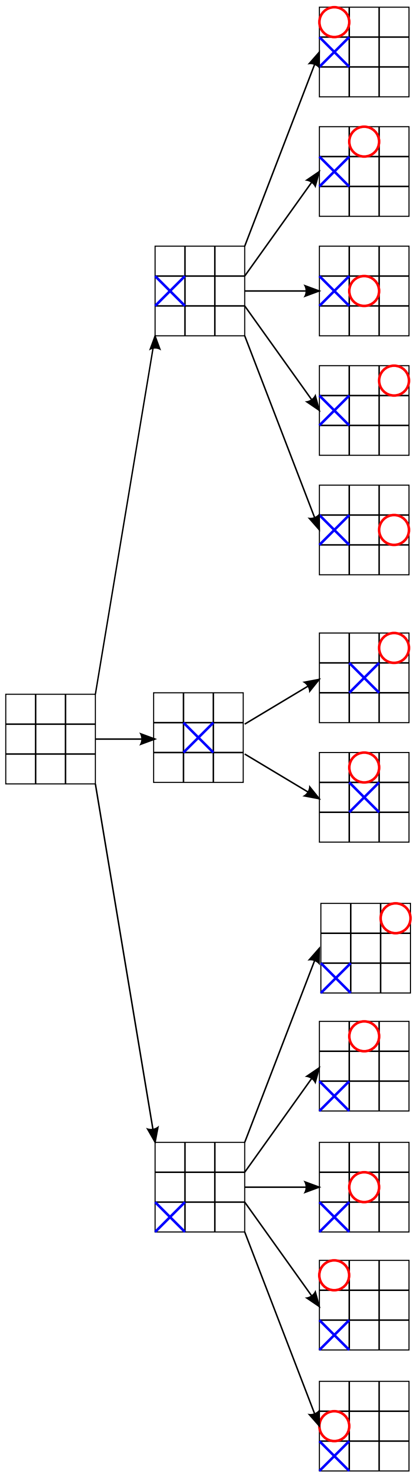


Figure 3.1: Beginning of the game tree of “Tic Tac Toe”

In the mathematical group theory these equivalent positions are called a group. We can create them by combining such transformations.

Which symmetry operations can be used depends on the game and its rules. We will now look at some well-known games that include symmetric positions.

We can rotate a state of the game “Tic Tac Toe” or mirror it horizontally, vertically and over both diagonals as shown in Figure 3.2. “Tic Tac Toe” is not an asymmetric game (we described this term in Section 1.1.4). Although we are dealing with asymmetric games in this work, we use it to describe symmetries in games, because it is very well-known.

Some games have even more symmetric positions, for example the game “Nine Men’s Morris”. It is not only possible to apply all transformations described above for the game “Tic Tac Toe”. We can also swap the outer and the inner circle of the game board to get equivalent states as shown in Figure 3.3. “Nine Men’s Morris” is not an asymmetric game too. But as it has this one extra option to create equivalent states, we use it here for the description of symmetries.

In Section 4.5 and Section 6.5, we describe how we find similar states of the asymmetric two-player games “Fuchs und Henne” and “Fox and Geese”.

We only have to store one of those equal states of a game when we solve a game. Therefore we have to define which representation of the state we want to use. We define a so-called fingerprint function to do this. This function creates a unique value for all states of the same state group. After we create a new state of the game, we call the fingerprint function for this state. This results in a state which we use to compare with the already stored data. If we already considered one state of this group, we discard it, otherwise we store it. If we would not use this fingerprint function, we would have to create each symmetric version of the state and compare each of them with the already stored states. That would significantly slow down the creation of all states of a game.

3.3 Complexity of Games

We can express the complexity of a game by the game-tree complexity and the state-space complexity.

To calculate the game-tree complexity of a game we count the leaves of the game tree. This is also the number of different instances of the game that we can play starting with the initial state [2] [10]. This means these two references define the state-space complexity as the number of different ways to play a game from the beginning to its end. The setting of the tree used for this definition is not described in detail. In general the number of nodes defines the complexity of a search tree. We are of the opinion, that counting the number of nodes of a tree, what means at the same time to count all different settings of the game, results in a more meaningful value for the complexity of a game. Of course, when we have a tree with nodes of a limited degree, there is no asymptotic difference between these two definitions of the game tree-complexity.

We calculate the state-space complexity of a game by counting all different positions that can occur in a game [2] [10].

For complex games these values are difficult to describe exactly with a formula. So sometimes it is necessary to fall back on estimated values or to define upper and lower bounds.

We show the complexity of some well-known games in Table 3.1.

Game	Game-Tree Complexity	State-Space Complexity
Chess	10^{123}	10^{46}
Connect Four	10^{21}	10^{14}
Nine Men's Morris	10^{50}	10^{10}
Go	10^{360}	10^{172}
Othello	10^{58}	10^{28}
Abalone	10^{180}	10^{25}
Pentago	10^{74}	10^{16}

Table 3.1: Complexity of some well-known games [2] [16]

3.4 The Framework

The framework was implemented in `C++` and we use it to strongly solve combinatorial two-player games with perfect information.

When we want to strongly solve a game using this framework, our first step is to calculate the state space by determining the successor states of all already known states of a game. For this the algorithm uses depth first search. It also takes care of the extremely important point of looking at each already known state only once, regardless of how often it occurs during the game. Thus it prevents unnecessary long computation time or even an infinite loop.

After the framework creates all reachable states of a game, it classifies them. This can either be done forward or backwards.

When we do the classification forward, we store the result of the classify method for every state of the game. This is the setup phase. After this, the classification of the terminal states of the game is already correct. Then the Algorithm iterates through the states. If it finds a state that is not classified so far, it classifies all his successor states. If one of them is a losing state, it classifies the originally considered state as a winning state for the current player. But if there isn't a losing state within the successor states, the state stays not classified, if not all successor states have been classified so far. Otherwise the algorithm scans the successor states for a draw state. If it finds one or more, it classifies the state as a draw state. If there is no draw state too, the original state is a losing state. The algorithm repeats this until all states are classified.

When classifying backwards the algorithm needs an additional data structure. It uses it to count the number of successor states that have not been classified as a winning state so far. The first step of this classification is to set all states to draw states except the terminal states. The algorithm also sets the value of the count variable of these states to the respective number of successor states. A losing state always means that all its predecessor states are winning states. So in case of a losing state the algorithm sets the predecessor states of the state directly to winning states. After classifying a state as a winning state, the algorithm reduces the count variable of all predecessor states by one, because this variable represents the number of not yet classified successor states of a state. If this number reaches zero at any state, this means that all successor states are winning states for the other player. So the current state is a losing state. We use the count variables to avoid repeated access to the states.

The two algorithms need a similar amount of storage but the backward classifying algorithm does less calculations and less read/write operations, so it is several times faster than the forward classification algorithm. But for some games it is not possible to get the predecessor states of each state. Then we have to use the forward classification [15].

In this chapter we explained the three levels of solving games. We also had a look at the game tree and the state space of games and how to use symmetries to reduce it. We compared some games by their game-tree complexity and state-space complexity. The last topic was the `C++` framework that we use to strongly solve combinatorial games.

In the next chapter we will consider the state space of the game “Fuchs und Henne”.

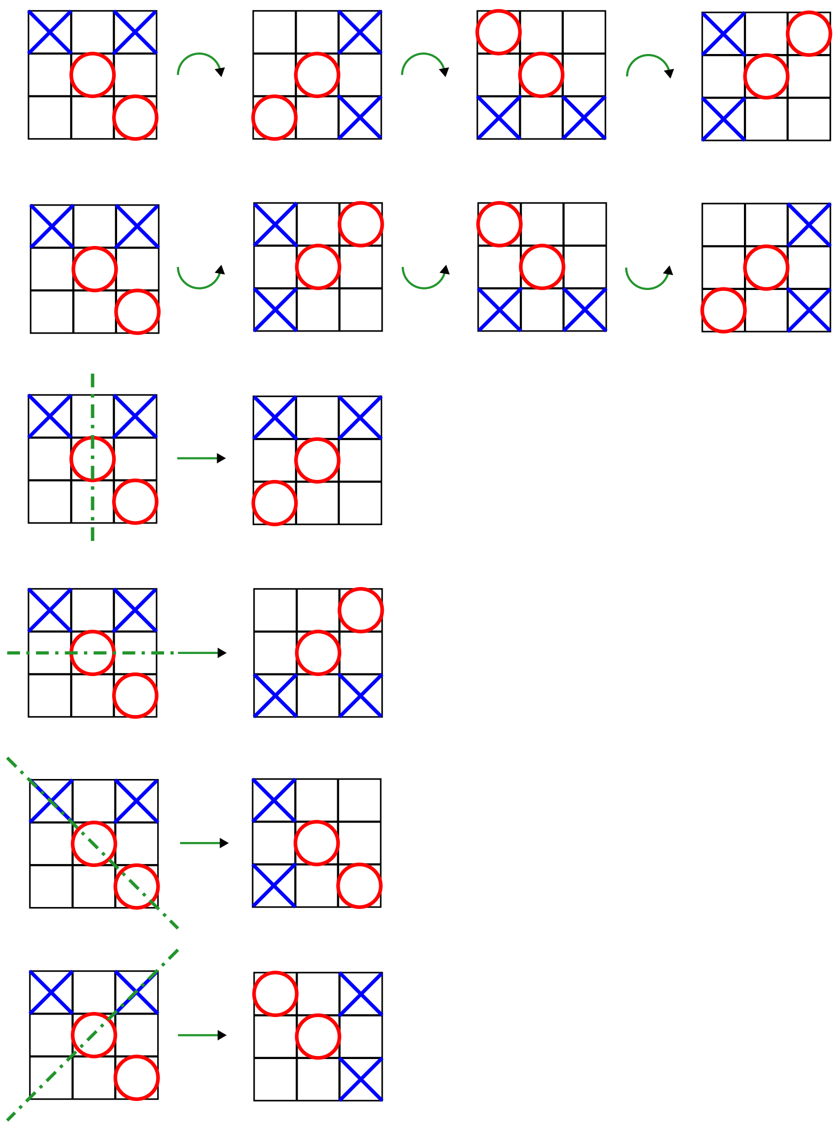


Figure 3.2: Symmetry operations of a “Tic Tac Toe” state

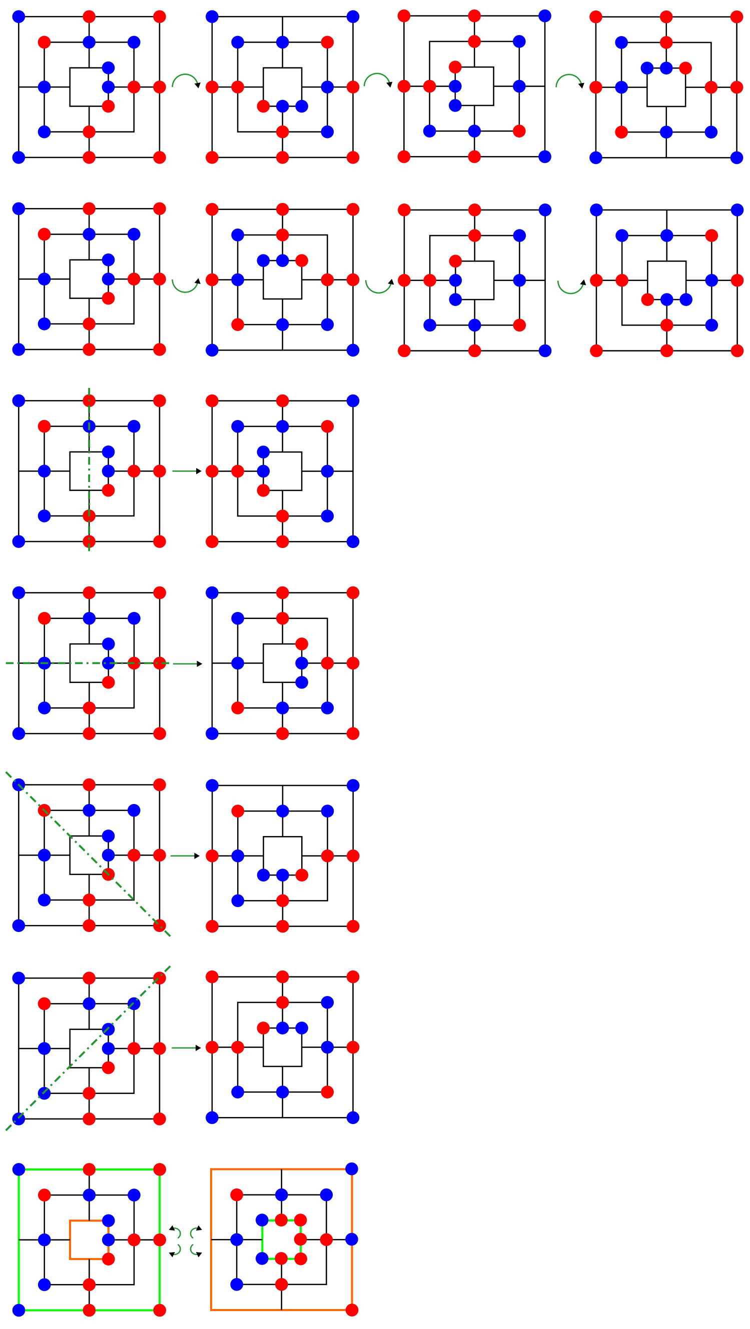


Figure 3.3: Symmetry operations on a “Nine Men’s Morris” state

4 State Space of “Fuchs und Henne”

In Chapter 3 we described that games can be solved by considering their state space. To do so computer programs are used, because the exact number of states can usually not be described by a simple formula. But to get an approximate idea of how many states a game has, an upper bound is usually calculated. For the game “Fuchs und Henne” we will do this in this chapter.

When considering the possible different arrangements of the game pieces on the game board this results in an upper bound of the number of states in the game. This is caused by the fact that some of the states may can’t be reached within the course of the game. We show some of them in Section 4.7. We calculate the upper bound of states for the game ’Fuchs und Henne’ in the following sections. For the game “Fox and Geese” we will do this in Chapter 6.

4.1 Placing Foxes and Hens

To calculate the upper bound of states for “Fuchs und Henne” we calculate all different placements of the foxes and hens on the game board. To do this we use combinatorics, more precisely, the combination without repetition.

The game board of “Fuchs und Henne” as shown in Figure 2.1 has 33 fields on which the hens and the foxes can be placed. As the number of foxes is a maximum of two, we place them first. We distribute the 9 to 20 hens to the remaining fields as we will describe in detail in the following sections. This includes also placings which can probably not be reached during playing the game starting at its initial state. That is why this is just an upper bound.

4.2 Positions of the Foxes

4.2.1 Possible Placements for two Foxes

The game “Fuchs und Henne” starts initially with two foxes. We place these 2 foxes anywhere on the game board. We use combination without repetition. The game board has 33 fields so we use the binomial coefficient 33 choose 2 to get all situations:

$$\binom{33}{2} = 528$$

4.2.2 Possible Placements for one Fox

In “Fuchs und Henne” the hen player is allowed to remove a fox of the game board if the fox player misses to beat a hen if it would be possible. Consequently there is also the opportunity to have just one fox in the game. This one fox can be placed anywhere on the game board. Here we have to place this one fox on the board with 33 fields, so we use the binomial coefficient 33 choose 1 to get all situations:

$$\binom{33}{1} = 33$$

4.2.3 Possible Placements for no Foxes

Of course, it is also possible that there is no fox on the game board of “Fuchs und Henne”. This situation means, that the fox player loses. It can be reached if the fox player misses to beat a hen - if it would be possible - twice, without beating a hen in the meantime, what would have given him/her back one fox.

For this situation there is just one way as 33 choose 0 results in 1.

$$\binom{33}{0} = 1$$

We will not consider these states further as they represent the end of the game.

4.2.4 Total Placements for the Foxes

To get all possible placements for the 0 to 2 foxes anywhere on the game field, we have to sum up the results of the above considerations.

$$528 + 33 + 1 = 562$$

So all in all there are 562 ways to place the two to zero foxes anywhere on the game field. This number of course is, as already mentioned, just an upper bound as not all settings have to be reachable when playing the game. Some ways also drop out as the game field is symmetrical. We describe this in detail in Section 4.5.

4.3 Positions of the Hens

The main version of "Fuchs und Henne" initially starts with 20 hens and to reach the goal of filling the hen house at least 9 hens are needed. So there may be 9 to 20 hens on the field. As all settings with less than 9 hens result in the end of the game, these emplacements will not be considered here.

After placing the zero to two foxes as we described above, there are still 31 to 33 possible empty fields for the hens left on which we can place the 9 to 20 hens. We can represent the upper bound of the number of different placements by binomial coefficients again.

Having a look at placing the hens on 31 free fields when still both foxes are in the game, the following combinations are possible:

Place 9 to 20 hens on 31 fields:

$$\binom{31}{9} = 20\,160\,075$$

$$\binom{31}{10} = 44\,352\,165$$

$$\binom{31}{11} = 84\,672\,315$$

$$\ldots$$

$$\binom{31}{18} = 206\,253\,075$$

$$\binom{31}{19} = 141\,120\,525$$

$$\binom{31}{20} = 84\,672\,315$$

To get the number of possible settings for 9 to 20 hens on 31 fields, we have to sum up the values above:

$$\sum_{i=9}^{20} \binom{31}{i} = 2\,060\,049\,510$$

If one fox gets removed from the board, there are 32 possible fields left to again place the 9 to 20 hens:

$$\binom{32}{9} = 28\,048\,800$$

$$\binom{32}{10} = 64\,512\,240$$

$$\binom{32}{11} = 129\,024\,480$$

$$\dots$$

$$\binom{32}{18} = 471\,435\,600$$

$$\binom{32}{19} = 347\,373\,600$$

$$\binom{32}{20} = 225\,792\,840$$

We can calculate the number of different placements of 9 to 20 hens on 32 remaining fields again by the sum of the values above:

$$\sum_{i=9}^{20} \binom{32}{i} = 4\,043\,315\,430$$

In the case both foxes got removed from the board, the game ends because the fox player is not able to move any more. As there is no further information in this game-ending settings, all those settings without a fox do not have to be considered further.

So the whole number of possible settings for 9 to 20 hens on 31 or 32 fields on the game board can be calculated as follows:

$$\sum_{i=9}^{20} \binom{31}{i} + \sum_{i=9}^{20} \binom{32}{i} = 2\,060\,049\,510 + 4\,043\,315\,430 = 6\,103\,364\,940$$

4.4 All Possible Placements

As shown in Section 4.2.1 there are 562 different ways to place two foxes on the game board and Section 4.3 shows that there are 2 060 049 510 different ways to place the 9 to 20 hens on the remaining 31 fields.

To get all possible placements in the case of two foxes the number of possible fox settings and the number of hen settings has to be multiplied:

$$\sum_{i=9}^{20} \binom{31}{i} * 528 = 1\,087\,706\,141\,280$$

If there is only one fox in the game, there are 33 different settings for just one fox, as shown in Section 4.2.2. To place the 9 to 20 hens on the remaining 32 fields, there are 4 043 315 430 different ways, as described in Section 4.3. Again these two values have to be multiplied to get all game settings for these circumstances:

$$\sum_{i=9}^{20} \binom{32}{i} * 33 = 133\,429\,409\,190$$

The number of different settings for 1 or 2 foxes and 9 to 20 hens can be calculated by adding these two values:

$$\sum_{i=9}^{20} \binom{31}{i} * 528 + \sum_{i=9}^{20} \binom{32}{i} * 33 = 1\,221\,135\,550\,470$$

As every state could occur either after the move of the hen player or after the move of fox player, this number has to be multiplied by two to also take care of this information:

$$(\sum_{i=9}^{20} \binom{31}{i} * 528 + \sum_{i=9}^{20} \binom{32}{i} * 33) * 2 = 2\,442\,271\,100\,940$$

So “Fuchs und Henne” has an upper bound of 2 442 271 100 940 different states without taking care of symmetric positions. We will consider this in the following sections.

4.5 Symmetric Position

Section 3.2.1 describes how the state space of a game can be reduced using symmetries. We can also use this property to improve the upper bound of the state space of "Fuchs und Henne" we calculated above.

As already mentioned before, the game board of "Fuchs und Henne" includes symmetric positions. This is caused by the fact, that the game board has a vertical axis of symmetry as shown in Figure 4.1. At first glance, it looks like there is also a horizontal axis of symmetry. But this axis wouldn't be correct as there is the hen house on the lower part of the game board. Another reason why the playing field can not be mirrored horizontally is the fact, that the hens are allowed to move in three directions only, as they are not allowed to move back.

For example, this means the two settings shown in Figure 4.2 can be considered the same.

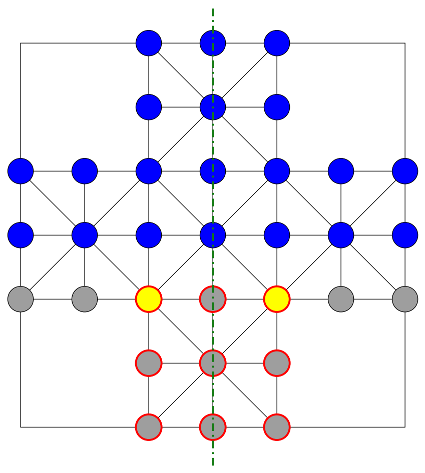


Figure 4.1: The game board of "Fuchs und Henne" with its axis of symmetry

When solving a game, the upper bound of the state space as calculated in Section 4.4 can be reduced by using these symmetries. The similar states can be found by vertical mirroring (in other games also by horizontally mirroring or different rotations). To know which of the symmetrical states should be stored, a mapping function is needed. This function returns the mapped state - so-called fingerprint - of the state. This is described in detail in Section 3.2.1. In the following section we describe how this fingerprint of symmetric states is defined for "Fuchs und Henne".

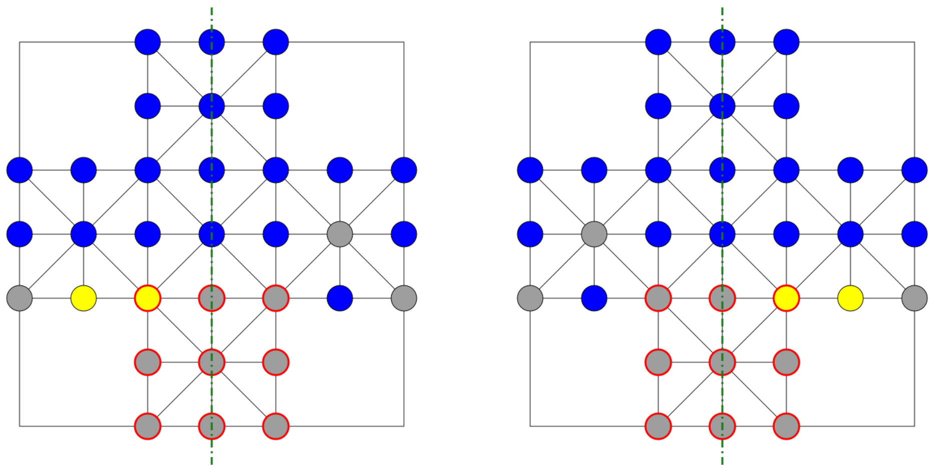


Figure 4.2: Two settings of “Fuchs und Henne” which we consider to be the same

4.5.1 Fingerprint of “Fuchs und Henne” States

To get the fingerprint of a state of “Fuchs und Henne”, we scan the game board from the left upper corner to the right lower corner as described in Pseudo-code 1.

At first glance attention is paid to the foxes. If both foxes are on the left side of the board, we already found the fingerprint. If both foxes are on the right side of the board, we have to mirror the game board vertically to have both foxes on the left side.

If one fox is placed on the left side and one fox on the right side, the higher located fox determines the fingerprint as seen in Figure 4.3 in which we marked the higher located fox of the original state by an orange circle. If the right fox is higher located, we mirror the board vertically to create the fingerprint of the state. If both foxes are placed on the same line, we index the foxes as shown in Figure 4.4. The fox with the lower index has to be on the left side of the board. Otherwise we mirror the board vertically to get the fingerprint of the state.

The foxes can also be arranged symmetrically. They can be placed one on the right side of the game board and the other one on the symmetrically same spot on the other side of the game board. We see an example for this in Figure 4.5 where one symmetric position is coloured orange and the other one violet. Another way for a symmetrical fox position occurs, if both foxes are placed on the center line of the game board. We see the fields of the center line in Figure 4.6 where we coloured them turquoise. In these cases the fingerprint is determined by the positions of the hens. The first position with a hen on one side of the game board and an empty field on the symmetrical position on the other side of the game board defines the fingerprint. If this first representative hen is on the right side, the game board has to be mirrored vertically. If this hen is left, the fingerprint is already found.

Of course it is also possible, that both hens and foxes are arranged symmetrically. In this case, the fingerprint is already given.

Algorithm 1 Calculating the fingerprint for “Fuchs und Henne” state

```
procedure getFingerprint
    $bSize \leftarrow 7$
    $i \leftarrow 0$
    $j \leftarrow 0$
    $pHen \leftarrow '.'$
    for $i = 0 \rightarrow i = 6$ do
        for $j = 0 \rightarrow j = 2$ do
            if $board(i)(j) \neq board(i)(bSize - j - 1)$ then
                if $board(i)(j) = FOX$ then
                    return
                else if $board(i)(bSize - j - 1) = FOX$ then
                    reflectBoardLeftRight
                    return
                else if $board(i)(j) = HEN$ AND $pHen = '.'$ then
                    $pHen \leftarrow 'l'$
                else if $pHen = '.'$ then
                    $pHen \leftarrow 'r'$
                end if
            end if
        end for
    end for
    if $pHen = 'l'$ then return
    else if $pHen = '.'$ then return
    else
        reflectBoardLeftRight
    end if
end procedure
```

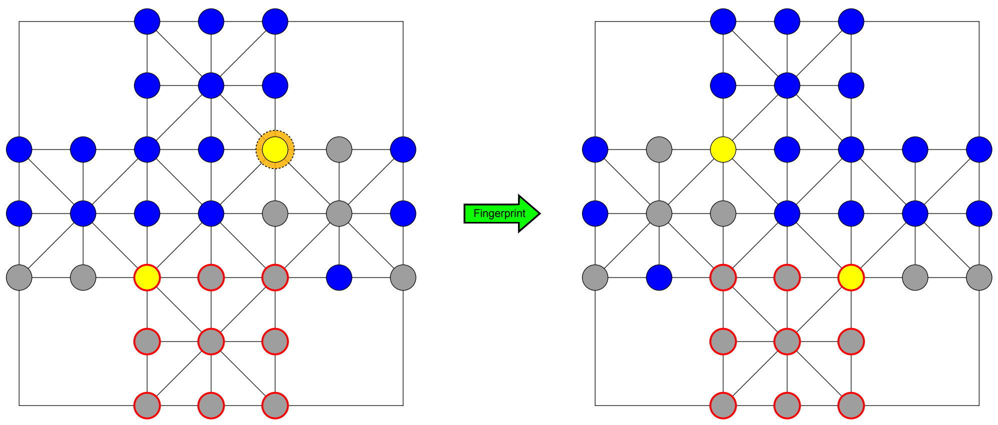


Figure 4.3: Fingerprint of “Fuchs und Henne” given by higher located fox

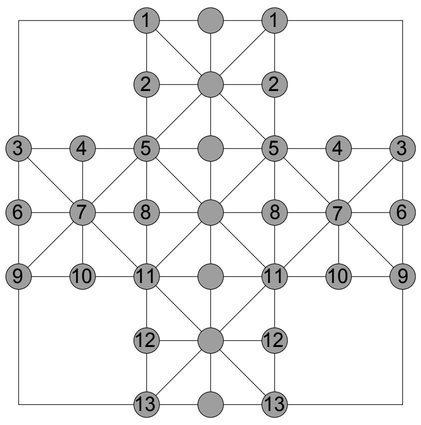


Figure 4.4: Left and right indexing of game board

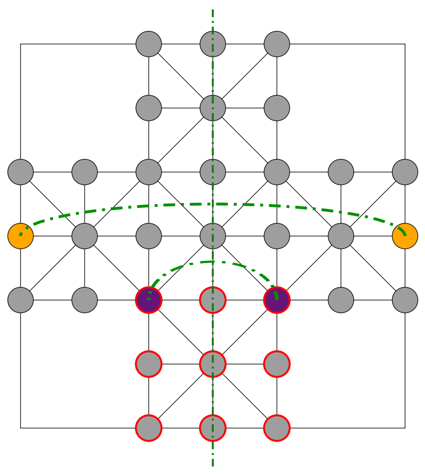


Figure 4.5: Example for symmetric fields in “Fuchs und Henne”

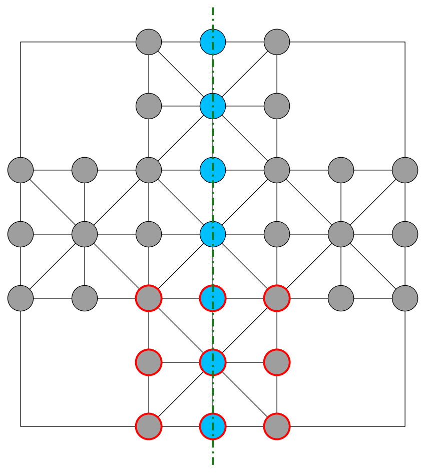


Figure 4.6: Center line of the game board of “Fuchs und Henne”

4.5.2 Reduce Upper Bound of the State Space by looking at Symmetric Positions

4.5.2.1 Symmetric Positions with two Foxes

To place two foxes on the game board, there are 3 placement categories. Either both foxes are on the same side of the game board (left or right) including the center line, or the foxes are on different sides of the game board. In the second case there are again two different ways. Either the position is full symmetric or not.

To place both foxes on the same side of the game board, the foxes must be placed on the 20 left or respectively right fields including the center line. Which fields are meant is shown in Figure 4.7, where we coloured the fields on the left side including the center line pink. We calculate the number of settings for this conditions as follows:

$$\binom{20}{2} = 190$$

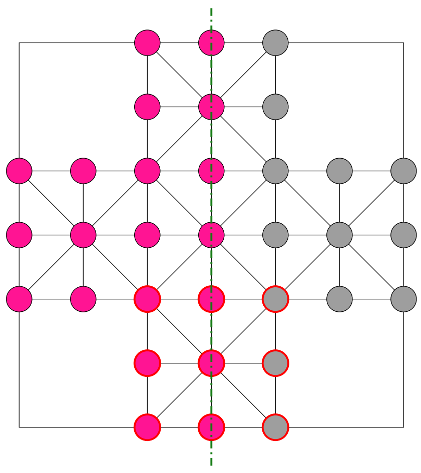


Figure 4.7: Positions of the game board described as left side

If the two foxes are placed on different sides of the game board in symmetric position, there are 13 different ways to place them. We see this combinations in Figure 4.8 where we coloured the symmetric position of a field in the yellow box in the same colour as in the orange box.

If the foxes are not both on the same side of the board and not in symmetric position, the number of placements can be calculated as follows:

$$\binom{13}{2} = 78$$

This is caused by the fact that if they are not in symmetric position, the fox on the right side can (just for calculation) be mirrored to his symmetric position on the left side and so there are two foxes on the 13 left fields.

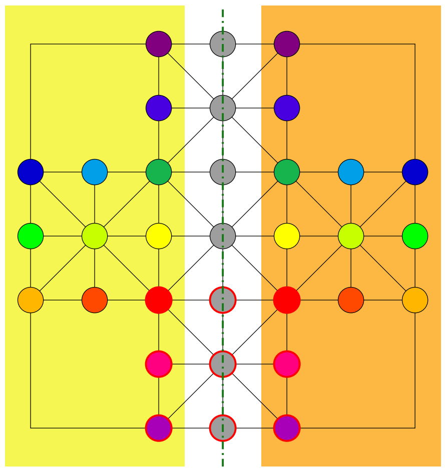


Figure 4.8: Symmetric fields coloured in 13 different colours

4.5.2.2 Symmetric Positions with one Fox

If there is just one fox on the game board, it can be placed on one of the 20 fields of one side of the game field including the center line (either on the right or on the left side). In Figure 4.7 we coloured this fields pink.

4.5.2.3 Fox Positions with Focus on Symmetries

As described above, when observing the symmetries there are 190 ways to place two foxes on the same side of the board, 13 ways for a symmetric position with one fox on each side of the board and 78 ways for positions with one fox on each side of the board without a symmetric position. If there is just one fox there are 20 different positions:

$$190 + 13 + 78 + 20 = 301$$

We reduced the number of fox position from 562 (as calculated in Section 4.2.4) to 301, if symmetries are taken into account.

4.6 All Possible Placements with Focus on Symmetries

In Section 4.4 the upper bound of states for the game "Fuchs und Henne" was calculated to be 2 442 271 100 940 but this number also includes symmetric positions, as mentioned above.

After taking care of symmetrical positions, we calculate the upper bound of the total number of states for "Fuchs und Henne" as follows:

$$(\sum_{i=9}^{20}\binom{31}{i} * 281 + \sum_{i=9}^{20}\binom{32}{i} * 20) * 2 = 1\,319\,480\,441\,820$$

There are still some symmetric positions included, because we count some states with symmetric foxes twice with this formula. The Pseudo-code 1 shows how to find a unique fingerprint for those states by looking at the hens positions. This is not done in this formula, so we count them more than once. This adjustment would complicate the formula and would not improve the upper bound significantly, so we will not make that adjustment.

4.7 A Selection of not reachable States

The number of states we calculated above includes some states that can't occur while playing the game starting at its defined initial state. Some of them are caused by the fact, that the hens are not allowed to move backwards on the game field. This means, that the hens can never reach their setting of the initial state again after the game has started, because there is a compulsion to move. Some states that can not be reached are shown in Figure 4.9.

In this chapter we found an upper bound of states for the game "Fuchs und Henne". To solve the game, the real states have to be calculated. How to store the states in a database using as less bits as possible, is described in the following chapter.

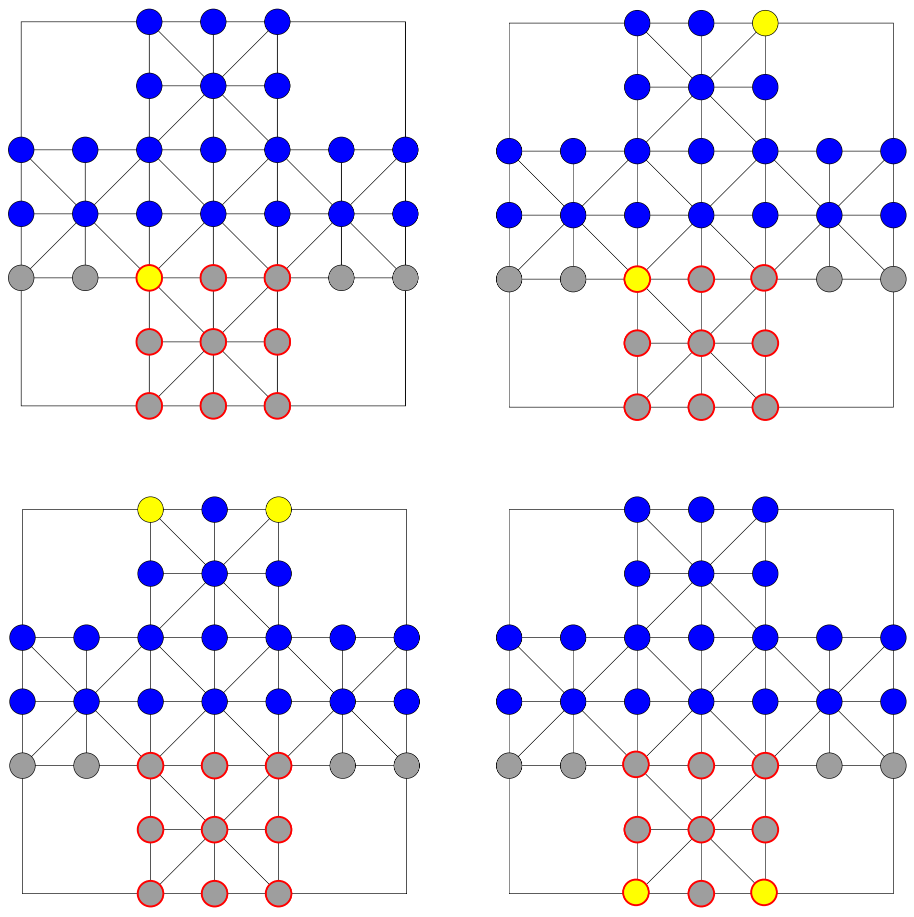


Figure 4.9: A selection of not reachable states of “Fuchs und Henne”

5 Encoding of “Fuchs und Henne”

To solve “Fuchs und Henne” strongly, we implemented the game using the `C++` game framework described in [15]. We use it to solve combinatorial games strongly. In Section 3.4, we describe how the framework solves games.

To store all possible positions of the game, we have to encode the game board to a number that requires as little bits as possible. Of course, the coding has to be unique for each position and it also has to be possible to decode it again.

To accomplish this, our very first step is to apply the fingerprint function for the respective state as described in Section 4.5.1.

For every game state the unique coding consists of 3 parts:

- The positions of the foxes
- The positions of the hens
- The current player

We code each of these three parts to a unique number. We then combine these three numbers. This results in the encoded representation of the particular game state. We describe the full encoding and decoding process of the game “Fuchs und Henne” in the following sections.

5.1 Encoding of the Game Field

To get a unique code for each game situation, we have to calculate three different numbers as we already mentioned above:

- Unique code for the foxes
- Unique code for the hens
- Unique code for the player

To reach the goal of getting a code for a game position which needs as little bits as possible, each of these three codes has to need as less bits as possible too.

In the following subsections, we will describe how these codes get calculated.

5.1.1 Encoding of the Foxes

In Section 4.5.2.3 we already described, that there are three different categories of placements for two foxes on the game board:

- Both foxes are on the same side of the board (including center line) (190 placements)
- One fox on each side of the board in symmetric position (13 placements)
- One fox on each side of the board in non-symmetric position (78 placements)

Additionally, there are 20 ways to place one fox and one way for no fox.

The fox code reaches from 0 to 301 and gets split up as follows:

Type	**Fox code range**
No fox	0
One Fox	1 - 20
Two foxes in symmetric position	21 - 33
Two foxes in non-symmetric position	34 - 111
Two foxes both left + center line	112 - 301

Table 5.1: Fox code ranges of “Fuchs und Henne ”

5.1.1.1 Fox Code of no Fox

There is just one way to place no fox on the game field, so every game situation without a fox receives the fox code 0. If there is no fox in the game, the fox player loses. Thus, it is not necessary to consider those states further, as already mentioned.

5.1.1.2 Fox Code for one Fox

The fox code for a setting with just one fox gets a value in the range of 1-20. As the fingerprint of the game field ensures, that the fox is placed on the left side (including center line), we can use the index of the fox directly for the fox code. The numeration of the game field is shown in Figure 5.1. As 0 is already reserved for situations without a fox, we have to increment the calculated fox code by one.

5.1.1.3 Fox Code for two Foxes in Symmetric Position

The range for the value of the fox code for game situations with foxes in symmetric positions reaches from 21 to 33. We calculate the respective unique fox code as follows: Because the foxes are in symmetric position, one fox sets the unique position of the other fox. Because of this, the index of the fox on the left side of the game board specifies the fox code. The range for this case reaches from 21 to 33, so we have to increase the index by 21 to fit in the range and also to be able to distinguish it from a fox code of only one fox.

5.1.1.4 Fox Code for two Foxes in Non-Symmetric Position

The fox code range for game situations with two foxes, where one fox is placed on the left side of the game board and the second fox is placed on the right side of the game board, reaches from 34 to 111. To calculate the individual number, we mirror the fox on the right side of the game field to the corresponding position on the left side of the game field. It is important that we use the fingerprint function before the encoding, because otherwise a unique decoding will not be possible. We convert the first 13 places of the game field into a binary sequence, in the order shown in Figure 5.1, where positions with a fox are represented by a 1 and the others by 0. We encode this binary value by using the algorithm described in Section 5.3. We increase the resulting value by 34 to fit the range and to be able to uniquely identify the type of the fox setting.

5.1.1.5 Fox Code for two Foxes on the same Side of the Board

The range for the fox code for game situations with both foxes on the left side of the game board (including center line) reaches from 112 to 301. Because we applied the fingerprint function as first step of the encoding process, we can be sure to have both foxes on the left side of the board. They can be placed on the 20 fields on the left side of the board including the center line. To get the unique value for those game situations, we create a binary value of size 20 for these fields. The places where a fox is located, are represented by 1 and the others by 0. We also encode this binary value using the algorithm described in Section 5.3. To get the final value for the fox code, we have to increase the resulting number by 112 to fit the given range.

5.1.2 Encoding of the Hens

In order to get a unique code for the hens placed on the game field, in the first step the board gets traversed column by column from top to bottom and converted into a binary number. This happens in the following way: If a hen is placed at the current position, we write a one for this position. In any other case, we write a zero.

In Figure 5.1 the game field is numbered the way the encoding process goes through the game field.

We encode the resulting binary number with the algorithm described in Section 5.3.

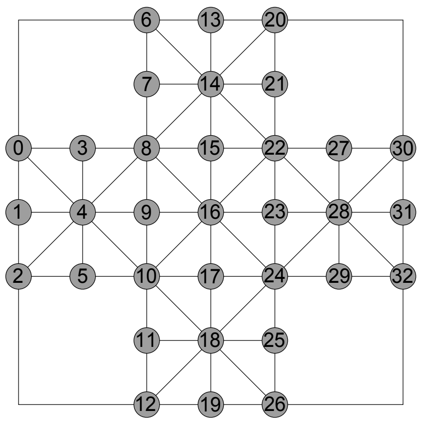


Figure 5.1: Game board for “Fuchs und Henne”, numbered in the order used for encoding.

The algorithm needs to know the number of ones included in the binary sequence, to know the starting point on the Pascal triangle. We need it for the encoding and also for the decoding process. This means, that we have to store this information in the code for the hens. That is why we add the following sum to the hen code:

$$\sum_{k=8}^{n_h-1} \binom{33}{k}$$

with: n_h – Number of hens (ones)

With the minimum of 9 hens the hen-player could still reach the goal of filling up the hen house. States with 8 hens are losing states. That’s why we start the lower index of the sum at a value of 8.

At this point we could improve the encoding a bit. The fields where a fox is placed could be skipped when we create the binary sequence for the hens. But we have to keep in mind, that the number of foxes can vary from zero to two. The states without a fox are final states of the game. They do not have to be considered in detail, as already mentioned before. That means, that there is at least one fox on the game field and we could leave out its position at the encoding of the hens. This wouldn't improve the coding significantly, so we don't do this improvement here.

5.1.3 Encoding of the Player

To save the current player of the respective game situation, there is just one bit necessary. If it is the hen player's turn, we set the code to 0 and if it is the fox player's turn, we set the code to 1.

5.1.4 Calculating a Unique Code for a Game Situation

The unique code for the whole game situation consists of the code for the hen positions, the code for the fox positions and the code for the player. We have to combine these three codes to get the unique code for the respective game situation. But it is important to connect the codes in a way to be able to clearly split them again for the decoding process.

When combining the numbers the best way is to start with the hen‘s code as it is the largest number. We multiply this number by the number of different fox codes (what is the same as the maximum code increased by one). The fox code for "Fuchs und Henne" reaches from 0 to 301 so there are 302 different fox codes. This multiplication reserves the necessary space for the corresponding fox code which we add to the now enlarged code. The last step is to add the code for the player. Therefore we multiply the code by 2 (what again is the maximum code for the player increased by one) to reserve the last bit for the code of the player. As last step we add the code of the player to this enlarged number.

To sum this up we calculate the code gets as follows:

$$code := (C_h * num(C_f) + C_f) * 2 + C_p \tag{5.1}$$

with:

C_h	-	Code for hens
C_f	-	Code for foxes
C_p	-	Code for player
$num(C_f)$	-	Total number of fox codes = maximum fox code + 1

In the next section we will describe how we can decode this codes for states of “Fuchs und Henne”.

5.2 Decoding of the Game Field

In this section we look at the decoding of “Fuchs und Henne” states. The codes consist of three parts:

- Unique code for the foxes
- Unique code for the hens
- Unique code for the player

We will have a closer look at the decoding process in the following subsections.

5.2.1 Split the Code in its Parts

The formula to combine the three parts of the code of a “Fuchs und Henne” state was the following:

$$code := (C_h * num(C_f) + C_f) * 2 + C_p \tag{5.2}$$

with:

C_h	-	Code for hens
C_f	-	Code for foxes
C_p	-	Code for player
$num(C_f)$	-	Total number of fox codes = maximum fox code + 1

At the decoding process this calculation has to be reversed as follows:

$$C_p := \text{ code } \& \ 0x01 \tag{5.3}$$
$$C_f := (code >> 1) \bmod num(C_f) \tag{5.4}$$
$$C_h := ((code >> 1) - C_f)/num(C_f) \tag{5.5}$$

with:

C_h	-	Code for hens
C_f	-	Code for foxes
C_p	-	Code for player
$num(C_f)$	-	Total number of fox codes = maximum fox code + 1
$\&$	-	Bitwise AND
$0x01$	-	Binary number 00000001
$>>$	-	Bitwise right shift operator

We can now decode the three codes individually. This will result in one game situation. In the following subsections we will describe the necessary details.

5.2.2 Decoding of the Player

The last element of the binary representation of the code is the code for the player. It is either zero or one. We use the binary AND operation to receive it from the code. This operation uses a logical conjunction on each pair of corresponding bits. In the formula above, we used "&" as the notation for this operation. Specifically, we use the binary AND to connect the binary number 00000001 with the code. This results in a one if the last element of the binary representation of the code is also a one. Otherwise it results in a zero. States with a player code of zero are states of the hen player. States with a player code of one are states of the fox player.

After we extracted the player code, we can truncate the last digit of the code in the next calculations. We use bitwise shifting to the right for this. We used the notation ">>" for this operation in the formulas above. With this, the number is shifted right by the number of digits specified after the operator. The binary number is padded with zeros from the left. A bit shift by 1 digit means an integer division by 2.

5.2.3 Decoding of the Hens

We encoded the hens by converting them to a binary sequence. For the encoding of this sequence we used the algorithm described in Section 5.3.

When decoding the hen code we have to rebuild the binary sequence out of the code again. We do this as described in the description of the algorithm. But because we need to know the number of ones included in the sequence to use this algorithm, we stored this information in the code while the encoding process. We have to extract this number before we apply the algorithm.

To achieve this, we subtract $\binom{33}{i}$ from the code as long as the result is positive. The minimum number of hens in this game is 9. All states with 8 hens are losing states. That is why we start with $i = 8$. For the further encoding of the hens we use the last positive result of this calculation. We reconstruct the binary sequence for the hens out of this value by using the algorithm.

We transfer the recovered sequence to the game field. We go through the sequence from left to right and distribute the values on the board in the order shown in Figure 5.1. At places with a one in the sequence we place a hen.

5.2.4 Decoding of the Foxes

To decode the fox code we first have to look which fox code range the code fits. The ranges are shown in Table 5.1. The decoding process depends on the range. We will now look at the different ways of the fox decoding.

5.2.4.1 Decoding of States with no Fox

If the fox code is 0 the original state has no fox. This is a losing state for the fox player. States with no fox represent the end of the game and we do not need to consider them further - as already mentioned.

5.2.4.2 Decoding of States with one Fox

If the fox code fits the range 1 to 20 the state has one fox. Since the fingerprint function was applied before the coding process, we can be sure that the fox is on the left side of the game field, including the center line. We decrement the code by one because this value has been added to fit the range. The resulting code represents the index of the fox on the game field where the fox has to be placed. We can see the indices in Figure 5.1.

5.2.4.3 Decoding of States with two Foxes in Symmetric Position

If the fox code fits the range 21 to 33 the state has two foxes - one fox on the left side of the game board and one fox on the right side. They are placed in symmetric position. As first step we reduce the fox code by 21, because this value has been added to fit the range. The resulting code represents the index of the fox on the left side of the game board. The indices are shown in Figure 5.1. We get the position of the fox on the right side of the game board by mirroring the position of the fox on the left side as shown in Figure 4.8.

5.2.4.4 Decoding of States with two Foxes in Non-Symmetric Position

If the fox code fits the range 34 to 111 the state has two foxes in non-symmetric position with one fox on the left side of the board and one fox on the right side of the board. The first step is to reduce the value by 34 to get the real value because we added this value to fit the range. The remaining value has been encoded using the algorithm described in Section 5.3. Because we mirrored the fox on the right side of the board to the left side during the encoding, we know that 2 foxes are on these 13 left fields. This means that the starting point for the decoding is $\binom{13}{2}$. This results in a binary sequence of length 13 including two ones. Because we know that one fox has to be on the right side of the game board, one fox has to be mirrored. We can find out which one it is when we compare the index they get when we apply the scheme shown in Figure 4.4. The fox with the higher index gets mirrored to the right side. Please mention that this is not the same index as used or the creation of the binary sequence.

5.2.4.5 Decoding of States with two Foxes on the left Side of the Board including the Center Line

If the fox code fits the range 112 to 301 the state has two foxes both on the left side of the board including the center line. To encode it, at first we have to reduce the code by 112 because this value has been added to fit the range. The resulting code has been encoded using the algorithm described in Section 5.3. The original binary sequence has a length of 20 and includes two ones. That means that the starting point for the decoding is $\binom{20}{2}$. We transfer the resulting sequence to the game field. We go through the sequence from left to right and distribute the values on the board in the order shown in Figure 5.1. At places with a one in the sequence we place a hen.

5.3 The Encoding Algorithm

In this section we will describe an algorithm to encode binary sequences of length n with weight w where w means the number of ones included in this sequence [17]. We use this algorithm for the encoding and for the decoding of the states of the games.

The following formula got proofed by induction:

$$i(t) = \sum_{k=1}^{n} t_k \binom{n-k}{w_k}$$

$$w_k = \sum_{i=k}^{n} t_i$$

$$\binom{n}{w} = 0 \text{ for } w > n$$

with:	n	-	length of the binary sequence
	$i(t)$	-	resulting code for the sequence
	t_k	-	$\in \{0,1\}$
	w_k	-	number of ones in the sequence

Since the binomial coefficient can be read from the Pascal triangle, this coding algorithm can be easily shown on this triangle.

We use the example shown in this paper to describe the algorithm. If the binary sequence '010100' gets encoded using the formula above, we move the following path through the pascal triangle, what is shown in Figure 5.2: The sequence has a size of 6 and includes 2 ones. This means the starting point is $\binom{6}{2} = 15$. We scan the binary number from left to right digit by digit. If there is a zero on the current place, we navigate one step in x-direction. If there is a one, we navigate one step in y-direction and note the number that is standing next on x-direction.

In the example, we move from the starting point 15 in x-direction towards 10, as there is a zero on the first position of the sequence. Because the next position is a one, the path leads us in y-direction to 4. We note the next number on x-direction - the 6. A zero comes next, what means that we move in x-direction to 3. Because of the following one the next step is in y-direction again, to 1. We note the next position in x-direction, in this case a 2. The next two digits are zeroes, so we move in x-direction though the triangle up to its top.

We add the noted numbers: $6 + 2 = 8$

The encoding for the sequence ’010100’ is 8.

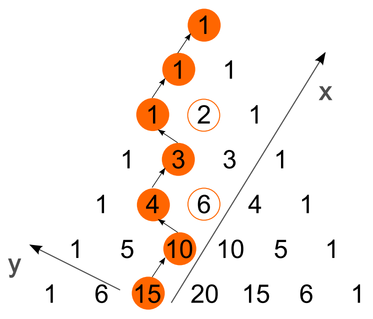


Figure 5.2: Path which the encoding algorithm uses on Pascal triangle

To decode a code, in this case an 8, we will use the Pascal triangle again. The starting point is again the number $\binom{6}{2} = 15$. We compare the code with the next number in x-direction, in this case 10. As $8 < 10$ the path goes in y-direction to 10 and we note a zero. The next step is to compare the code 8 again with the number in x-direction. As $8 > 6$ we note a one which results in ’01’ and a step in y-direction to 4 follows. Now the new current code is $8 - 6 = 2$. We compare it to the next number in x-direction what is 3. Because of $3 > 2$ we note a zero resulting in ’010’ and we next move one step in x-direction to 3. The current code 2 is not less than the next number in x-direction so we note a one resulting in ’0101’ and the path leads us in y-direction. We reduce the code by 2 and the result is 0. That means we have to do the last two steps in x-direction and we add zeros to our sequence: ’010100’. The result is as expected the same sequence that we encoded before and the path through the Pascal triangle is the same as in the encoding process [17].

5.4 Encoding and Decoding of an Example State of “Fuchs und Henne”

In this section we encode an example state of “Fuchs und Henne” and decode the result again. The selected state is shown in Figure 5.3. As the current player also gets encoded, we define that the state is a state of the fox player.

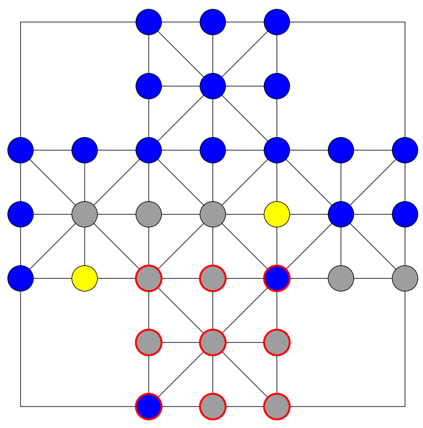


Figure 5.3: State of “Fuchs und Henne” for encoding and decoding example

5.4.1 Encoding of the Example State

The first step of the encoding of a state of “Fuchs und Henne” is to create the fingerprint of the respective state as described in Section 4.5.1. In this state the foxes are not placed in a symmetric position, so it is possible to define the fingerprint of the state by looking at the foxes. We compare the indices the foxes get out of the schema shown in Figure 4.4. The fox with the higher index has to be on the right side of the board. So we have to mirror the game board vertically. We can see the result in Figure 5.4.

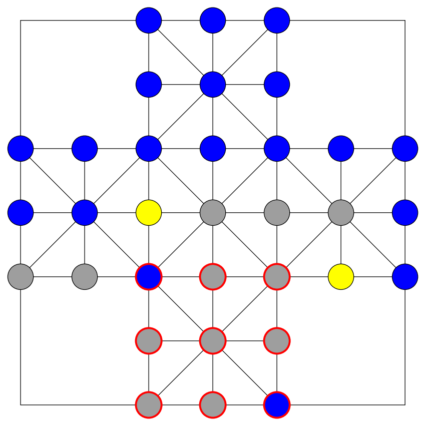


Figure 5.4: Fingerprint of “Fuchs und Henne” state for encoding and decoding example

Now we encode this fingerprint state. The first step is to encode the foxes. At first the type of fox position gets determined. In this case the foxes are positioned on different sides of the game board in non-symmetric position. That means the range for the fox code is 34 - 111 as shown in Table 5.1. The fox on the right side of the board gets mirrored to its position on the left side of the board. Then the first 13 places get converted into a binary sequence. The order in which we traverse the board is shown in Figure 5.1.

The example state results in the following binary sequence for the foxes: '0000010001000'

We encode this sequence using the algorithm described in Section 5.3.

$$C_f = \binom{7}{2} + \binom{3}{1} = 21 + 3 = 24$$

To fit the range, 34 gets added to this number:

$$C_f = 24 + 34 = 58$$

The code 58 represents the position of the two foxes on the game board.

The next step is to encode the hens. Therefore we scan the game field again the way shown in Figure 5.1 and convert it into a binary sequence. Fields with a hen get represented by a one, empty fields or fields with foxes get represented by zero: '110110111010011100001110001100111'.

We also encode this sequence using the algorithm described in Section 5.3:

$$\begin{gathered}
C_h = \\
1 * \binom{32}{19} + 1 * \binom{31}{18} + 0 * \binom{30}{17} + 1 * \binom{29}{17} + \\
1 * \binom{28}{16} + 0 * \binom{27}{15} + 1 * \binom{26}{15} + 1 * \binom{25}{14} + \\
1 * \binom{24}{13} + 0 * \binom{23}{12} + 1 * \binom{22}{12} + 0 * \binom{21}{11} + \\
0 * \binom{20}{11} + 1 * \binom{19}{11} + 1 * \binom{18}{10} + 1 * \binom{17}{9} + \\
0 * \binom{16}{8} + 0 * \binom{15}{8} + 0 * \binom{14}{8} + 0 * \binom{13}{8} + \\
1 * \binom{12}{8} + 1 * \binom{11}{7} + 1 * \binom{10}{6} + 0 * \binom{9}{5} +
\end{gathered}$$

$$0 * \binom{8}{5} + 0 * \binom{7}{5} + 1 * \binom{6}{5} + 1 * \binom{5}{4} +$$
$$0 * \binom{4}{3} + 0 * \binom{3}{3} + 1 * \binom{2}{3} + 1 * \binom{1}{2} +$$
$$1 * \binom{0}{1} =$$

$$347\,373\,600 + 206\,253\,075 + 0 + 51\,895\,935 +$$
$$30\,421\,755 + 0 + 7\,726\,160 + 4\,457\,400 +$$
$$2\,496\,144 + 0 + 646\,646 + 0 +$$
$$0 + 75\,582 + 43\,758 + 24\,310 +$$
$$0 + 0 + 0 + 0 +$$
$$495 + 330 + 210 + 0 +$$
$$0 + 0 + 6 + 5 +$$
$$0 + 0 + 0 + 0 +$$
$$0 = 651\,415\,411$$

The algorithm to encode binary sequences needs to know the number of ones included in the sequence for the encoding. We must add this information to the code because the number of hens may varies from 8 to 20. We add the following sum to the code for the hens:

$$\sum_{k=8}^{n_h-1} \binom{33}{k}$$

n_h is the number of Hens. In our example there are 19 hens so the upper bound of the summation is $n_h - 1 = 18$:

$$C_h = 651\,415\,411 + \sum_{k=8}^{18} \binom{33}{k} =$$
$$651\,415\,411 + 6\,493\,264\,836 = 7\,144\,680\,247$$

After this, the third factor of the encoding is the player. This state was defined as state of the fox player, so the code for the player is $C_p = 1$.

The last step is to combine those three codes as described in Section 5.1.4. The number of different codes for the foxes is 302 as shown in Table 5.1.

$$(C_h * num(C_f) + C_f) * 2 + C_p =$$
$$(7\,144\,680\,247 * 302 + 58) * 2 + 1 = 4\,315\,386\,869\,305$$

Thus, the example state is fully and uniquely encoded.

5.4.2 Decoding of the Example State

In the section above, we encoded the example state to the unique number 4 315 386 869 305. In this section we will decode it again.

Because the last step of the encoding was the addition of the player bit, the last bit of the code represents the player. To decode it again, bit operations can be used. With a binary AND of the code and the binary number 0x01 (what is binary ’00000001’) we get the last bit of the code - the code of the player.

$$C_p = 4\,315\,386\,869\,305 \ \& \ 0x01 = 1$$

Because the last bit of the code is a one, the current player of this state is the fox player.

The next step is to use a bit shift right by 1 position to reverse the multiplication by 2.

$$4\,315\,386\,869\,305 >> 1 = 2\,157\,693\,434\,652$$

This code includes the code for the hens and the code for the foxes. Because the hen code got multiplied by the maximum code of the foxes to be able to add it to the end of the code, the reverse operation is a modulo calculation. After calculating the code modulo 302 (that is the number of different fox codes), we receive the code for the foxes:

$$C_f = 2\,157\,693\,434\,652 \text{ mod } 302 = 58$$

To get the code for the hens, the code has to be subtracted by the fox code and then divided by 302:

$$2\,157\,693\,434\,652 - 58 = 2\,157\,693\,434\,594$$
$$C_h = 2\,157\,693\,434\,594 / 302 = 7\,144\,680\,247$$

As an alternative to these two steps, we could also directly do an integer division by 302, which would lead us to the same result.

After we have extracted the individual code elements, we are able to decode them.

To decode the hens, we have to reproduce the binary sequence out of the hen code. The binary sequence has a length of 33 and 8 to 20 ones (hens) are included. The number of hens was stored within the code of the hens. To get the number of hens, again binomial coefficients get used. $\binom{33}{ones}$ gets subtracted from the hen code as long as the hen code is larger than the binomial coefficient. After each subtraction the number of ones gets increased by one, it starts at 8 as this is the minimum number of hens.

The calculation of the number of hens for the example is the following:

$$7\,144\,680\,247 - \binom{33}{8} = 7\,130\,796\,091$$

$$7\,130\,796\,091 - \binom{33}{9} = 7\,092\,228\,991$$

$$7\,092\,228\,991 - \binom{33}{10} = 6\,999\,667\,951$$

$$6\,999\,667\,951 - \binom{33}{11} = 6\,806\,131\,231$$

$$6\,806\,131\,231 - \binom{33}{12} = 6\,451\,313\,911$$

$$6\,451\,313\,911 - \binom{33}{13} = 5\,878\,147\,471$$

$$5\,878\,147\,471 - \binom{33}{14} = 5\,059\,338\,271$$

$$5\,059\,338\,271 - \binom{33}{15} = 4\,022\,179\,951$$

$$4\,022\,179\,951 - \binom{33}{16} = 2\,855\,376\,841$$

$$2\,855\,376\,841 - \binom{33}{17} = 1\,688\,573\,731$$

$$1\,688\,573\,731 - \binom{33}{18} = 651\,415\,411$$

$$651\,415\,411 - \binom{33}{\mathbf{19}} = -167\,393\,789$$

The number of hens of the state is 19 and the code that represents the hens is the last positive number of the listing above: 651 415 411. We decode it using the algorithm described in Section 5.3. The start point is $\binom{33}{19}$.

Comparison	Binary	Calculation
$651\,415\,411 \geq \binom{32}{19}$	$\mathbf{1}$	$651\,415\,411 - \binom{32}{19} = 304\,041\,81$
$304\,041\,811 \geq \binom{31}{18}$	$\mathbf{1}$	$304\,041\,811 - \binom{31}{18} = 97\,788\,736$
$97\,788\,736 < \binom{30}{17}$	$\mathbf{0}$	
$97\,788\,736 \geq \binom{29}{17}$	$\mathbf{1}$	$97\,788\,736 - \binom{29}{17} = 45\,892\,801$
$45\,892\,801 \geq \binom{28}{16}$	$\mathbf{1}$	$45\,892\,801 - \binom{28}{16} = 15\,471\,046$
$15\,471\,046 < \binom{27}{15}$	$\mathbf{0}$	
$15\,471\,046 \geq \binom{26}{15}$	$\mathbf{1}$	$15\,471\,046 - \binom{26}{15} = 7\,744\,886$
$7\,744\,886 \geq \binom{25}{14}$	$\mathbf{1}$	$7\,744\,886 - \binom{25}{14} = 3\,287\,486$
$3\,287\,486 \geq \binom{24}{13}$	$\mathbf{1}$	$3\,287\,486 - \binom{24}{13} = 791\,342$
$791\,342 < \binom{23}{12}$	$\mathbf{0}$	
$791\,342 \geq \binom{22}{12}$	$\mathbf{1}$	$791\,342 - \binom{22}{12} = 144\,696$
$144\,696 < \binom{21}{11}$	$\mathbf{0}$	
$144\,696 < \binom{20}{11}$	$\mathbf{0}$	
$144\,696 \geq \binom{19}{11}$	$\mathbf{1}$	$144\,696 - \binom{19}{11} = 69\,114$
$69\,114 \geq \binom{18}{10}$	$\mathbf{1}$	$69\,114 - \binom{18}{10} = 25\,356$
$25\,356 \geq \binom{17}{9}$	$\mathbf{1}$	$25\,356 - \binom{17}{9} = 1\,046$

Comparison	Binary	Calculation
$1\,046 < \binom{16}{8}$	**0**	
$1\,046 < \binom{15}{8}$	**0**	
$1\,046 < \binom{14}{8}$	**0**	
$1\,046 < \binom{13}{8}$	**0**	

Comparison	Binary	Calculation
$1\,046 \geq \binom{12}{8}$	**1**	$1\,046 - \binom{12}{8} = 551$
$551 \geq \binom{11}{7}$	**1**	$551 - \binom{11}{7} = 221$
$221 \geq \binom{10}{6}$	**1**	$221 - \binom{10}{6} = 11$
$11 < \binom{9}{5}$	**0**	
$11 < \binom{8}{5}$	**0**	
$11 < \binom{7}{5}$	**0**	
$11 \geq \binom{6}{5}$	**1**	$11 - \binom{6}{5} = 5$
$5 \geq \binom{5}{4}$	**1**	$5 - \binom{5}{4} = 0$
$0 < \binom{4}{3}$	**0**	
$0 < \binom{3}{3}$	**0**	
$0 \geq \binom{2}{3}$	**1**	$0 - \binom{2}{3} = 0$
$0 \geq \binom{1}{2}$	**1**	$0 - \binom{1}{2} = 0$
$0 \geq \binom{0}{1}$	**1**	$0 - \binom{0}{1} = 0$

After this calculation we combine the binary values to the following sequence: '11011011101001110000111000 1100111'. It is the same sequence as in the encoding in Section 5.4.1. We apply the sequence to the playing field in the order in which it was taken. The order is shown in Figure 5.1. The result is shown in Figure 5.5 where the blue circles represent the hens on the game board.

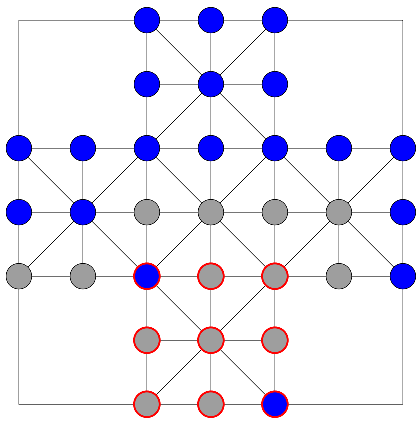


Figure 5.5: Decoded hens of example state of “Fuchs und Henne”

In order to decode the fox code ’58’, at first we check in which fox code range it belongs. The different ranges are shown in Table 5.1. In this case it is the range 34 - 111: Two foxes in non-symmetric position. To get the actual code, the fox code has to be reduced by 34: 58 - 34 = 24. To decode 24 we apply the technique described in Section 5.3 again. In this case the number of ones in the binary sequence is already known. This information was hold in the fox code range. At this state we have two ones in a sequence of the length 13. Therefore, the starting point is $\binom{13}{2}$.

Comparison	Binary	Calculation
$24 < \binom{12}{2}$	**0**	
$24 < \binom{11}{2}$	**0**	
$24 < \binom{10}{2}$	**0**	
$24 < \binom{9}{2}$	**0**	

Comparison	Binary	Calculation
$24 < \binom{8}{2}$	**0**	
$24 \geq \binom{7}{2}$	**1**	$24 - \binom{7}{2} = 3$
$3 < \binom{6}{1}$	**0**	
$3 < \binom{5}{1}$	**0**	
$3 < \binom{4}{1}$	**0**	
$3 \geq \binom{3}{1}$	**1**	$3 - \binom{3}{1} = 0$

The resulting sequence is ’0000010001’. We fill it up with zeros to get a sequence of size 13: ’0000010001000’. We transfer the sequence to the left side of the board. The order is shown in Figure 5.1. The resulting setting of the foxes is shown in Figure 5.6.

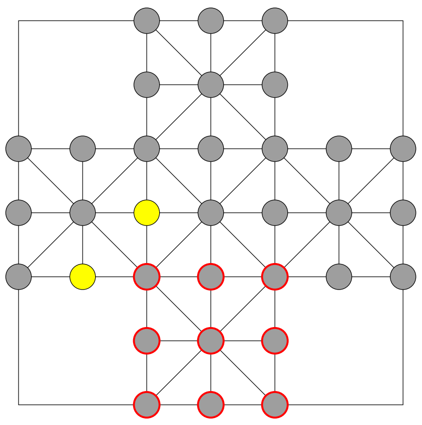


Figure 5.6: Decoded foxes of example state of “Fuchs und Henne” both on left side

One of the foxes has to be moved to the right side of the game field. For this, we compare their index of the schema shown in Figure 4.4. Please notice that that is not the same schema as for the binary sequences. The fox with the higher index gets moved to the right side of the game field. The result of the decoding of the fox code is shown in Figure 5.7.

The last step of the decoding process is to combine the results onto one single game field as shown in Figure 5.8.

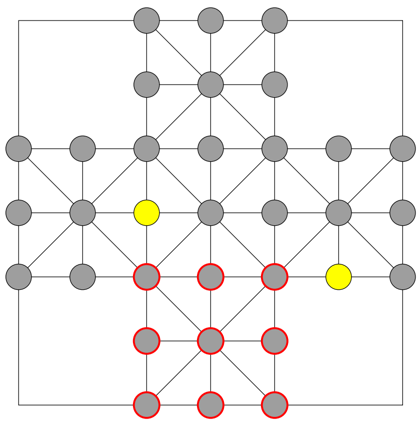


Figure 5.7: Decoded foxes of example state of “Fuchs und Henne”

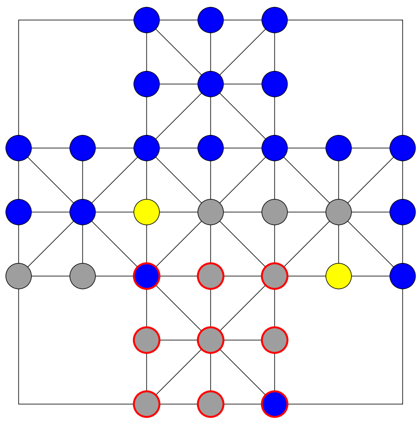


Figure 5.8: Decoding result of an example state of “Fuchs und Henne”

5.5 Minimum and Maximum Code for “Fuchs und Henne”

In this section, we determine the minimum and the maximum code for states of this game. We need this information in order to estimate the range of the codes for “Fuchs und Henne”.

5.5.1 Minimum Code for “Fuchs und Henne”

To calculate the minimum code for “Fuchs und Henne” we have to find a state in which all three parts of the code are minimal. We described the three parts of the code in Section 5.1.

The minimum code for the player is 0, thus it is a state of the hen player.

In order to find the setting of the hens with the minimum code we have to create a binary sequence with 8 ones, which is a losing state for the hen player. The value we add to store the information of how many ones are included in the sequence is 0 for this minimum number of hens. In order to encode the binary sequences we use the algorithm described in Section 5.3. If all ones are at the end of the sequence then the code is 0. Figure 5.1 shows the order in which we transfer the fields of the board to the binary sequence. That means we have to place the nine hens on the fields with the highest index as shown in Figure 5.9.

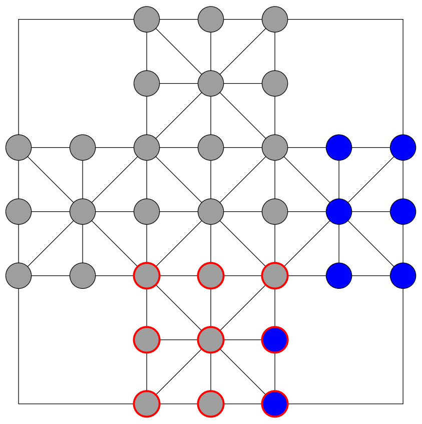


Figure 5.9: Setting of the hens with the minimum hen code

A state of the game “Fuchs und Henne” has zero to two foxes as described in Section 2.2.3. We can see the ranges of the fox code in Table 5.1. The minimum code for the foxes is 0, which means that there is no fox placed on the game field. Considering the definition of the fingerprint function (see Section 4.5.1) we would have to mirror this state vertically before the encoding process. After this, the hens wouldn’t be placed at the end of the board anymore, but they would be on the beginning of the board. Binary sequences with all ones at the beginning result in the largest code. Because of this, the state without a fox is not the state with the minimum code.

The next smallest fox code is 1. We assign this code if a state has only one fox and this fox is placed in the field with the index 0. We see this field in Figure 5.1.

We see the resulting state in Figure 5.10.

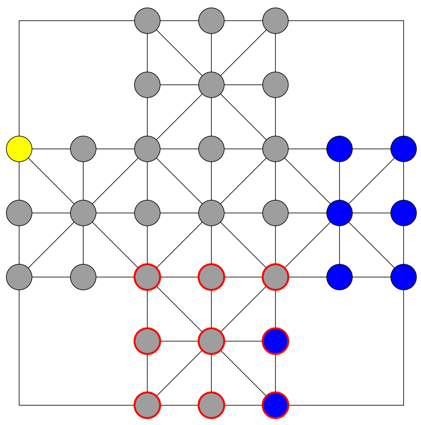


Figure 5.10: State of “Fuchs und Henne” that results in the minimum code

Because this state has the fox on the left side of the board it will not be changed by the fingerprint function.

5.5.1.1 Calculation of the Minimum Code

We now encode the state described above. The state is a state of the hen player. That means the player code is 0:

$$C_p = 0$$

The fox is placed on the field with the index 0 thus the fox code is 0. To fit the range we have to add 1 to this value:

$$C_f = 1$$

To encode the positions of the hens we have to create the binary sequence as described above. The resulting sequence is ’000000000000000000000000011111111’

$$C_h =$$

$$0 * \binom{32}{9} + 0 * \binom{31}{9} + 0 * \binom{30}{9} + 0 * \binom{29}{9} +$$
$$0 * \binom{28}{9} + 0 * \binom{27}{9} + 0 * \binom{26}{9} + 0 * \binom{25}{9} +$$

$$0*\binom{24}{9}+0*\binom{23}{9}+0*\binom{22}{9}+0*\binom{21}{9}+$$

$$0*\binom{20}{9}+0*\binom{19}{9}+0*\binom{18}{9}+0*\binom{17}{9}+$$

$$0*\binom{16}{9}+0*\binom{15}{9}+0*\binom{14}{9}+0*\binom{13}{9}+$$

$$0*\binom{12}{9}+0*\binom{11}{9}+0*\binom{10}{9}+0*\binom{9}{9}+$$

$$0*\binom{8}{9}+1*\binom{7}{8}+1*\binom{6}{7}+1*\binom{5}{6}+$$

$$1*\binom{4}{5}+1*\binom{3}{4}+1*\binom{2}{3}+1*\binom{1}{2}+$$

$$1*\binom{0}{1}=$$

$$1*0+1*0+1*0+1*0+1*0+1*0+1*0+1*0=0$$

To store the information on how many ones the binary sequence consists of we have to add the result of the following sum to the hen code:

$$\sum_{k=8}^{n_h-1}\binom{33}{k}$$

n_h (here: 8) is the number of hens. That means the value we have to add to the hen code is 0.

The last step is to combine the three codes with the formula described in Section 5.1.4 as follows:

$$C=(C_h*num(C_f)+C_f)*2+C_p$$

$$C=(0*302+1)*2+0=2$$

The calculation shows that the minimum code for “Fuchs und Henne” is 2.

5.5.2 Maximum Code for “Fuchs und Henne”

To find the maximum code of “Fuchs und Henne”, the three parts that the code consists of must be as large as possible. Since the hens represent the basis of the code and the other two codes are appended at the end, above all this code must be as large as possible.

We get the maximum hen code when all hens are at the beginning of the binary sequence that we create for the encoding. The value we add to store the number of hens in the sequence is higher, if more hens are included. That is why we use a state with the maximum number of hens which is 20 in this game. We see this setting in Figure 5.11.

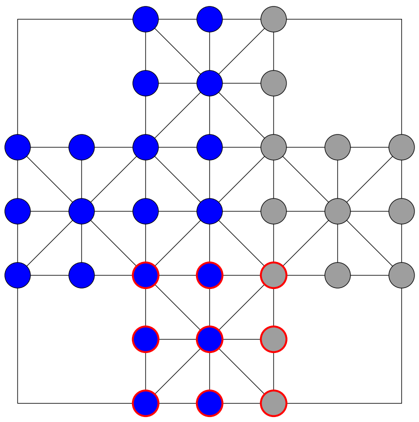


Figure 5.11: Setting of the hens with the maximum hen code

If we would place a fox on the remaining fields, the fingerprint function would mirror the game field vertically. Then the hens would not be placed in the position with the maximum code anymore. That means this setting without a fox is already the state with the maximum code. However, if we have a closer look at this state we see that this state is not reachable when we play the game starting at its initial state. A state without a fox is a final state in this game. That means the fox was removed during the previous move. This is just allowed, when the fox player missed an opportunity to beat a hen. However, in this state due to the positions of the hens there is no such opportunity.

That means we have to add a fox. We have to place it on the left side of the game board including the center line. We select the field with the highest index to keep the hen code as high as possible. This results in the state that we see in Figure 5.12.

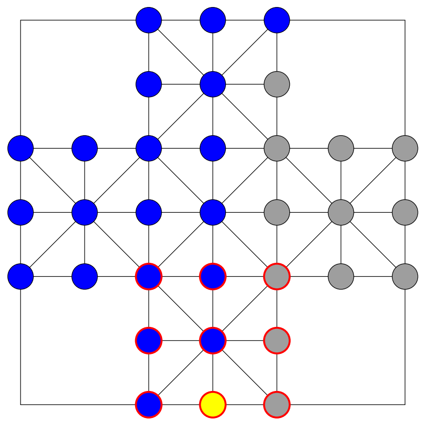


Figure 5.12: State of "Fuchs und Henne" that results in the maximum code

We define the state as state of the fox player, because this results in the higher player code.

5.5.2.1 Calculation of the Maximum Code

We now encode the state described above.

The state is a state of the fox player. That means the player code is 1:

$$C_p = 1$$

The state has one fox so we encode the fox as described in Section 5.1.1.2. The index of the fox is 19. We increase it by 1 to fit the range.

$$C_f = 20$$

To encode the hens we create the following binary sequence: '111111111111111111101000000000000'. We encode the binary sequence using the algorithm described in Section 5.3. The starting point is $\binom{33}{20}$.

$$C_h =$$

$$1 * \binom{32}{20} + 1 * \binom{31}{19} + 1 * \binom{30}{18} + 1 * \binom{29}{17} +$$
$$1 * \binom{28}{16} + 1 * \binom{27}{15} + 1 * \binom{26}{14} + 1 * \binom{25}{13} +$$

$$1 * \binom{24}{12} + 1 * \binom{23}{11} + 1 * \binom{22}{10} + 1 * \binom{21}{9} +$$
$$1 * \binom{20}{8} + 1 * \binom{19}{7} + 1 * \binom{18}{6} + 1 * \binom{17}{5} +$$
$$1 * \binom{16}{4} + 1 * \binom{15}{3} + 1 * \binom{14}{2} + 0 * \binom{13}{1} +$$
$$1 * \binom{12}{1} =$$
$$225\,792\,840 + 141\,120\,525 + 86\,493\,225 + 51\,895\,935 +$$
$$30\,421\,755 + 17\,383\,860 + 9\,657\,700 + 5\,200\,300 +$$
$$2\,704\,156 + 1\,352\,078 + 646\,646 + 293\,930 +$$
$$125\,970 + 50\,388 + 18\,564 + 6\,188 +$$
$$1\,820 + 455 + 91 + 0 +$$
$$12 = 573\,166\,438$$

We have to add the result of the following sum to store the information of the number of hens in the sequence:

$$\sum_{k=8}^{n_h-1} \binom{33}{k}$$
$$\sum_{k=8}^{19} \binom{33}{k} = 7\,312\,074\,036$$

We add this value to the hen code:

$$573\,166\,438 + 7\,312\,074\,036 = 7\,885\,240\,474$$

Our last step is to combine the three values with the formula described in Section 5.1.4:

$$C = (C_h * num(C_f) + C_f) * 2 + C_p$$
$$C = (7\,885\,240\,474 * 302 + 20) * 2 + 1 = 4\,762\,685\,246\,337$$

Our encoding results in the maximum code of 4 762 685 246 337 for the game “Fuchs und Henne”. This means we need 43 bits to store the data.

To sum up, in this chapter we introduced a method to encode states of the game “Fuchs und Henne”. We applied the method to an example state and then decoded the state again. In addition we looked at the minimum and maximum code for this game. In the next chapter we will examine the state space of the very similar game “Fox and Geese”.

6 State space of “Fox and Geese”

In Chapter 4 we considered the state space of the game “Fuchs und Henne”. In this chapter we consider the state space of a very similar game called “Fox and Geese”. That is why we focus mainly on the particularity of “Fox and Geese” in this chapter. At first we will have a look at the possible different placements of the tokens on the board. Thereby we calculate an upper bound of the number of reachable states of this game. Because we also consider states that we can not reach when we play the game starting at its initial state, this is just an upper bound. We then take care of symmetric positions, what will improve this upper bound.

6.1 Placing Fox and Geese

We want to calculate the upper bound of states for the game “Fox and Geese”. Therefore we use combinatorics without repetition to calculate the possible arrangements on the game board.

We play “Fox and Geese” on a game board with 33 fields as shown in Figure 2.1. Our first step is to place the fox on one of this fields. After this, we place the geese on the remaining fields. This results also in states that can not be reached when we play the game starting at its initial state. That is why this is just an upper bound of the number of states for this game.

6.2 Positions of the Fox

We play the game “Fox and Geese” with one single fox. We can place it anywhere on the game board. The game board for this game has 33 fields. We use the binomial coefficient 33 choose 1 to get all situations:

$$\binom{33}{1} = 33$$

This number is just an upper bound as already mentioned above. It also includes symmetric positions. We will describe this in detail in Section 6.5.

6.3 Positions of the Geese

We play “Fox and Geese” in our selected version with 15 geese. The geese player wins if he/she is able to surround the fox in a way it is not able to move anymore. We described this in Section 2.3.4. At our selected version we need at least 6 geese to achieve this as we already described in Section 2.3.4.

That means there are 6 to 15 geese on the game board.

After we placed the fox on the game board, there are 32 free fields left to place these 6 to 15 geese. To calculate the different possibilities to place them on the board, we use binomial coefficients again:

$$\binom{32}{6} = 906\,192$$

$$\binom{32}{7} = 3\,365\,856$$

$$\binom{32}{8} = 10\,518\,300$$

$$\binom{32}{9} = 28\,048\,800$$

$$\binom{32}{10} = 64\,512\,240$$

$$\binom{32}{11} = 129\,024\,480$$

$$\binom{32}{12} = 225\,792\,840$$

$$\binom{32}{13} = 347\,373\,600$$

$$\binom{32}{14} = 471\,435\,600$$

$$\binom{32}{15} = 565\,722\,720$$

To get the number of possible settings for 6 to 15 geese on the 32 remaining fields, we have to sum up the values we calculated above.

$$\sum_{i=6}^{15} \binom{32}{i} = 1\,846\,700\,628$$

6.4 All Possible Placements

We calculated the number of different placements for the fox in Section 6.2. The result was 33. After this we calculated the number of different placements for 6 to 15 geese on the remaining 32 fields. The result for this was 1 846 700 628.

To get the total number of different settings we have to combine this two values by multiplication:

$$\sum_{i=6}^{15} \binom{32}{i} * 33 = 60\,941\,120\,724$$

Each of these states can either be a state of the fox player or a state of the geese player. That is why we have to multiply the number of states by two to get the full number of states:

$$\sum_{i=6}^{15} \binom{32}{i} * 33 * 2 = 121\,882\,241\,448$$

The game “Fox and Geese” has an upper bound of 121 882 241 448 different states. Because we did not take care of symmetric positions, we are able to improve this upper bound. We will consider this in the following sections.

6.5 Symmetric Positions

In Section 3.2.1 we described how the state space of a game can be reduced by taking care of symmetric positions. In Section 4.5 we already did the reduction of the state space for the game “Fuchs und Henne”.

The game board of “Fox and Geese” has a vertical axis of symmetry. That means we can create similar states by mirroring the game board vertically. There is no horizontal axis of symmetry although there is no hen house in this game. It is not possible that we create similar states by mirroring horizontally because the geese are not allowed to move back.

In Figure 6.1 we see two states of “Fox and Geese” that we consider to be the same.

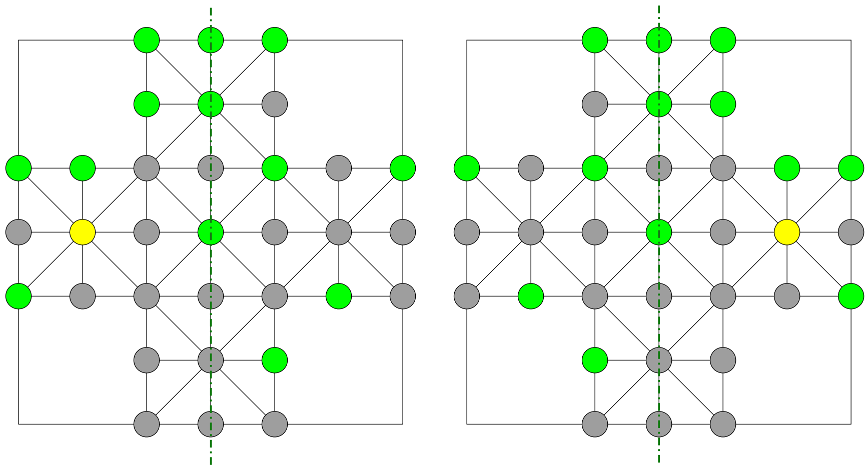


Figure 6.1: Two states of “Fox and Geese” which we consider to be the same

In Section 6.4 we calculated an upper bound of the states for “Fox and Geese”. We can reduce this number when we use the symmetry we described above. To know which of the similar states we want to store, we again need a fingerprint function as we already described in Section 3.2.1 and also for the game “Fuchs und Henne”. In the following subsection we will describe the fingerprint function for “Fox and Geese”.

6.5.1 Fingerprint of “Fox and Geese” States

To get the fingerprint of a state of “Fox and Geese” we have a look at the position of the fox. We define the fingerprint of the state as the version where the fox is on the left side of the game board. If the fox is on the right side of the board we mirror the board vertically to get the fingerprint. If the fox is on the center line of the game board what means that the fox is placed directly under the axis of symmetry the fingerprint depends on the position of the geese. We compare the fields in the same order as we did it for the game “Fuchs und Henne”. We see the order in Figure 4.4. The first position where on one side of the board is a goose and on the corresponding field on the other side is an empty field determines the fingerprint. We define the state in which the goose is on the left side of the board and the empty field is on the right side of the field as the fingerprint. If we have the opposite positions we have to mirror the game field. If the foxes and all geese are positioned symmetric the fingerprint is this one state.

6.5.2 Reduce Upper Bound of the State Space by looking at Symmetric Positions

To improve the upper bound for the number of states of “Fox and Geese” we use the symmetries we described above.

After we apply the fingerprint function to one state of “Fox and Geese” we can be sure to have a state with a fox on the left side of the game board including the center line. That means we are able to reduce the number of positions for the fox from 33 - we calculated this number in Section 6.2 - to 20.

6.6 All Possible Placements with Focus on Symmetries

We calculated the upper bound of states in Section 6.4. The result was 121 882 241 448. This number also includes the symmetric positions we described above.

To improve the upper bound for the state space of “Fox and Geese” we have to use the improved number of different states for the fox:

$$\sum_{i=6}^{15} \binom{32}{i} * 20 * 2 = 73\,868\,025\,120$$

This formula still includes some symmetric positions. We still count the settings with the fox placed on the center line twice. That means the upper bound can still be improved. But since this adjustment would complicate the formula and would not improve the upper bound significantly, we will not make that adjustment.

In this chapter we considered an upper bound of the state space of “Fox and Geese”. We took care of symmetric positions to improve this upper bound. In the following chapter we will describe the encoding and decoding of the states of this game.

7 Encoding of “Fox and Geese”

In Chapter 6 we considered the state space of the game “Fox and Geese”. In this chapter we create an encoding method for states of this game. The encoding method is similar to the one that we created for the game “Fuchs und Henne”. We also want to strongly solve the game using the framework we described in Section 3.4 which we also used for the game “Fuchs und Henne”. At the end of this chapter we encode an example state of “Fox and Geese” and decode the result again. The last section of this chapter deals with the minimum and the maximum code for states of “Fox and Geese”.

To encode a situation of the game we need to find its numeric representation. This number has to consist of as few bits as possible. We also have to be able to decode it again.

7.1 Encoding of the Game Field

Our first step of the encoding is always to apply the fingerprint function for the state.

The next step is to create a unique code for each of the following three parts of a game state:

- The position of the fox
- The positions of the geese
- The current player

We encode each part separately to a unique number. After this we combine the numbers and thus get the full code for one game state.

7.1.1 Encoding of the Fox

In Section 6.5.1 we defined that the fingerprint function always results in the version of the state which has the fox on the left side of the board including the center line. That means that the fox is placed on one of the 20 fields on the left side of the game board. We number the game board to get the index of the fox position. We show the numeration of the fields on the board in Figure 5.1. Finally, we use this index of the fox directly for the fox code.

7.1.2 Encoding of the Geese

We encode the geese nearly the same way we encoded the hens in the game “Fuchs und Henne” in Section 5.1.2. The main difference for the encoding is that “Fox and Geese” has always exactly one fox on the game board. That means when creating the binary sequence for the encoding algorithm (Section 5.3) we don’t need to consider the location of the fox. We convert the game field in the order we see in Figure 5.1. We write a zero for each empty field and a one for each field with a goose. We skip the field on which the fox is placed. The result is a binary sequence of length 32.

In Section 5.1.2, we described that the encoding algorithm needs to know the number of ones (geese) included in the sequence. We add this information to the geese code. As the length of the sequence for “Fox and Geese” is just 32, we have to add the following sum to the geese code:

$$\sum_{k=4}^{n_g-1} \binom{32}{k}$$

with: n_g – Number of geese (ones)

7.1.3 Encoding of the Player

The encoding process of the player is exactly the same as used for “Fuchs und Henne”. If it is the geese player’s turn we set the player code to 0 and if it is the fox player’s turn we set the code to 1.

7.1.4 Calculating a Unique Code for a Game Situation

To create the unique code for one game situation we have to combine the following three values: the code for the fox, the code for the geese and the code for the player. We combine the three codes in the same way we did for “Fuchs und Henne”, see Section 5.1.4.

$$code := (C_g * num(C_f) + C_f) * 2 + C_p \tag{7.1}$$

with:

C_g	-	Code for geese
C_f	-	Code for foxes
C_p	-	Code for player
$num(C_f)$	-	Total number of fox codes = maximum fox code + 1

In this game the total number of fox codes is 20. It reaches from 0 to 19.

7.2 Decoding of the Game Field

As mentioned above, the unique code of a "Fox and Geese" state consists of three parts:

- Unique code for the fox
- Unique code for the geese
- Unique code for the player

In the following subsections, we will decode this unique code for a state of the game "Fox and Geese". In order to achieve this, we will split this code in its parts and then we will decode each part separately.

7.2.1 Split the Code in its Parts

In Section 5.2.1, we created a formula to split the code of a "Fuchs und Henne" state in its three parts. In this section we reuse this formula that we used for the game "Fuchs und Henne".

$$C_p := \text{ code } \& \ 0x01 \tag{7.2}$$
$$C_f := (code >> 1) \text{ mod } num(C_f) \tag{7.3}$$
$$C_g := ((code >> 1) - C_f)/num(C_f) \tag{7.4}$$

with:

C_g	-	Code for geese
C_f	-	Code for foxes
C_p	-	Code for player
$num(C_f)$	-	Total number of fox codes = maximum fox code + 1
$\&$	-	Bitwise AND
$0x01$	-	Binary number 00000001
$>>$	-	Bitwise right shift operator

Since the decoding process for the game "Fox and geese" is the same as for "Fuchs und Henne", we won't describe the extraction of the tree codes in this section.

In the following subsections we will look at the decoding of the three parts of the code for "Fox and Geese".

7.2.2 Decoding of the Player

Once we have extracted the player code, we can immediately determine whether it is the fox player's turn or the geese player's turn. If the player code is 0, it is a state of the geese player. If the player code is a 1, it is a state of the fox player.

7.2.3 Decoding of the Fox

The fox code is a number from 0 to 19. It directly represents the index of the fox on the game board as shown in Figure 5.1. We place the fox on its index before we place the geese, because we left out the place of the fox during the encoding process.

7.2.4 Decoding of the Geese

We decode the extracted geese code with the algorithm described in Section 5.3. This results in a binary sequence of size 32. We transfer this sequence to the game board in the order we see in 5.1. We place a goose on each field which corresponds with a 1 in the binary sequence. We must keep in mind to leave out the place where the fox is located, because we didn't encode this field.

7.3 Encoding and Decoding of an Example State of "Fox and Geese"

In this section we encode an example state of the game "Fox and Geese". We will also decode it again. We selected the state shown in Figure 7.1. We define the state as state of the fox player.

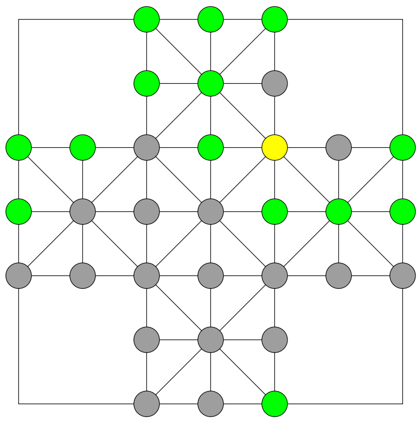


Figure 7.1: State of "Fox and Geese" for encoding and decoding example

7.3.1 Encoding of the Example State

Our first step at the encoding process is to apply the fingerprint function to the respective state. The fox has to be on the left side of the board which means we have to mirror the state vertically as shown in Figure 7.2.

Now we can start with the encoding. The first step is to encode the fox. We use the index of the fox like shown in Figure 5.1. In our example the fox has the index 8.

$$C_f = 8$$

The next step is to encode the geese. We transform the game field into a binary sequence with a length of 32. The order is again as shown in Figure 5.1. We leave out the field where the fox is located. For fields with a goose we write a one and for empty fields we write a zero: '11001010100111100001100000100110'

We encode the resulting sequence with the algorithm described in Section 5.3:

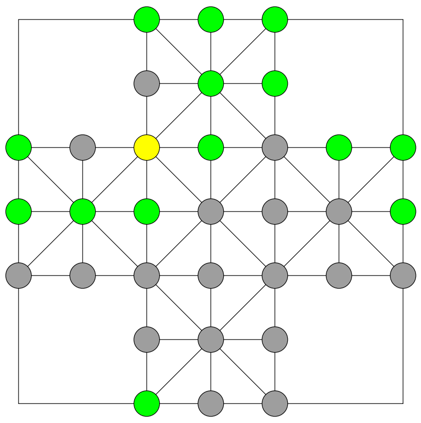


Figure 7.2: Fingerprint of the state of "Fox and Geese" for encoding and decoding example

$$C_g =$$

$$1 * \binom{31}{14} + 1 * \binom{30}{13} + 0 * \binom{29}{12} + 0 * \binom{28}{12} +$$

$$1 * \binom{27}{12} + 0 * \binom{26}{11} + 1 * \binom{25}{11} + 0 * \binom{24}{10} +$$

$$1 * \binom{23}{10} + 0 * \binom{22}{9} + 0 * \binom{21}{9} + 1 * \binom{20}{9} +$$

$$1 * \binom{19}{8} + 1 * \binom{18}{7} + 1 * \binom{17}{6} + 0 * \binom{16}{5} +$$

$$0 * \binom{15}{5} + 0 * \binom{14}{5} + 0 * \binom{13}{5} + 1 * \binom{12}{5} +$$

$$1 * \binom{11}{4} + 0 * \binom{10}{3} + 0 * \binom{9}{3} + 0 * \binom{8}{3} +$$

$$0 * \binom{7}{3} + 0 * \binom{6}{3} + 1 * \binom{5}{3} + 0 * \binom{4}{2} +$$

$$0 * \binom{3}{2} + 1 * \binom{2}{2} + 1 * \binom{1}{1} + 0 * \binom{0}{0} =$$

$$265\,182\,525 + 119\,759\,850 + 17\,383\,860 + 4\,457\,400 +$$

$$1\,144\,066 + 167\,960 + 75\,582 + 31\,824 +$$

$$12\,376 + 792 + 330 + 10 +$$

$$1 + 1 = 408\,216\,577$$

The algorithm needs to know the number of ones that are encoded in the sequence for the decoding. That is why we need to add the following sum to the code for the geese:

$$\sum_{k=6}^{n_g-1} \binom{32}{k}$$

n_g is the number of geese. In our example there are 14 geese, which means the upper bound of the summation is $n_g - 1 = 13$. The lower bound of the summation is 6, because in this version at least 6 geese are necessary to lock the fox:

$$C_g = 408\,216\,577 + \sum_{k=6}^{13} \binom{32}{k} =$$
$$408\,216\,577 + 809\,542\,308 = 1\,217\,758\,885$$

The third and last factor that we need for the encoding is the code for the player. We defined the state as state of the fox player. That means the code of the player is $C_p = 1$.

To get a unique code for the whole state of “Fox and Geese” we have to combine the three values as described in Section 7.1.4. The number of different codes for the fox is 20.

$$(C_g * num(C_f) + C_f) * 2 + C_p =$$
$$(1\,217\,758\,885 * 20 + 8) * 2 + 1 = 48\,710\,355\,417$$

Now we have encoded the example state uniquely.

7.3.2 Decoding of the Example State

In the section above, we encoded the example state into the unique code 48 710 355 417. In this chapter we decode it again.

Our first step is to split the code in its three parts as described in Section 7.2.1. To get the code for the player we use the binary AND operation with the binary number 0x01 (’00000001’). The result is the last bit of the code which at the same time is the code for the player.

$$C_p = 48\,710\,355\,417 \;\&\; 0x01 = 1$$

This means the state is a state of the fox player.

As we don't need the last bit of the code anymore, we shift the number to the right by one bit:

$$48\,710\,355\,417 >> 1 = 24\,355\,177\,708$$

This remaining code includes the code for the geese and the code for the fox. The next step is to extract the fox code:

$$C_f = 24\,355\,177\,708 \bmod 20 = 8$$

The last step is to get the code for the geese. We subtract the fox code from the code and divide the result by the maximum fox code, thus by 20. Of course, we can also directly do an integer division by 20 without subtracting the fox code first. With this aproach the result would be the same. We do the fox code subtraction just to show all steps of the reversal of the encoding formula.

$$24\,355\,177\,708 - 8 = 24\,355\,177\,700$$
$$C_g = 24\,355\,177\,700/20 = 1\,217\,758\,885$$

Now we can decode the three individual codes.

We can directly decode the player code. One stands for a state of the fox player.

The fox code directly represents the index of the fox on the game board. We compare it with Figure 5.1 to get the position and place the fox on the game board. We see the result in Figure 7.3

To decode the geese code we use the algorithm described in Section 5.3. We added the number of geese to the geese code, because we need this information for the decoding algorithm. In order to find this value we have to subtract the binomial coefficient $\binom{32}{ones}$ as long as the geese code is larger than the binomial coefficient. We start with 6, because this is the minimum number of geese.

Now we do this calculation for our example:

$$1\,217\,758\,885 - \binom{32}{6} = 1\,216\,852\,693$$
$$1\,216\,852\,693 - \binom{32}{7} = 1\,213\,486\,837$$

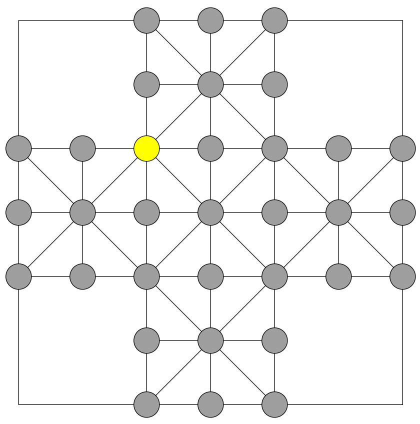


Figure 7.3: State of "Fox and Geese" example after fox decoding

$$1\,213\,486\,837 - \binom{32}{8} = 1\,202\,968\,537$$

$$1\,202\,968\,537 - \binom{32}{9} = 1\,174\,919\,737$$

$$1\,174\,919\,737 - \binom{32}{10} = 1\,110\,407\,497$$

$$1\,110\,407\,497 - \binom{32}{11} = 981\,383\,017$$

$$981\,383\,017 - \binom{32}{12} = 755\,590\,177$$

$$755\,590\,177 - \binom{32}{13} = 408\,216\,577$$

$$408\,216\,577 - \binom{32}{\mathbf{14}} = -63\,219\,023$$

The number of geese is 14 and the last positive number is the code for the geese: 408 216 577. Our next step is to decode it. For this we use the algorithm described in Section 5.3. The starting point is $\binom{32}{14}$, because it is a sequence of length 32 and includes 14 ones.

Comparison	Binary	Calculation
$408\,216\,577 \geq \binom{31}{14}$	**1**	$408\,216\,577 - \binom{31}{14} = 143\,034\,052$
$143\,034\,052 \geq \binom{30}{13}$	**1**	$143\,034\,052 - \binom{30}{13} = 23\,274\,202$
$23\,274\,202 < \binom{29}{12}$	**0**	
$23\,274\,202 < \binom{28}{12}$	**0**	
$23\,274\,202 \geq \binom{27}{12}$	**1**	$23\,274\,202 - \binom{27}{12} = 5\,890\,342$
$5\,890\,342 < \binom{26}{11}$	**0**	
$5\,890\,342 \geq \binom{25}{11}$	**1**	$5\,890\,342 - \binom{25}{11} = 1\,432\,942$
$1\,432\,942 < \binom{24}{10}$	**0**	
$1\,432\,942 \geq \binom{23}{10}$	**1**	$1\,432\,942 - \binom{23}{10} = 288\,876$
$288\,876 < \binom{22}{9}$	**0**	
$288\,876 < \binom{21}{9}$	**0**	
$288\,876 \geq \binom{20}{9}$	**1**	$288\,876 - \binom{20}{9} = 120\,916$
$120\,916 \geq \binom{19}{8}$	**1**	$120\,916 - \binom{19}{8} = 45\,334$
$45\,334 \geq \binom{18}{7}$	**1**	$45\,334 - \binom{18}{7} = 13\,510$
$13\,510 \geq \binom{17}{6}$	**1**	$13\,510 - \binom{17}{6} = 1\,134$
$1\,134 < \binom{16}{5}$	**0**	
$1\,134 < \binom{15}{5}$	**0**	
$1\,134 < \binom{14}{5}$	**0**	

Comparison	Binary	Calculation
$1\,134 < \binom{13}{5}$	**0**	
$1\,134 \geq \binom{12}{5}$	**1**	$1\,134 - \binom{12}{5} = 342$
$342 \geq \binom{11}{4}$	**1**	$342 - \binom{11}{4} = 12$
$12 < \binom{10}{3}$	**0**	
$12 < \binom{9}{3}$	**0**	
$12 < \binom{8}{3}$	**0**	
$12 < \binom{7}{3}$	**0**	
$12 < \binom{6}{3}$	**0**	
$12 \geq \binom{5}{3}$	**1**	$12 - \binom{5}{3} = 2$
$2 < \binom{4}{2}$	**0**	
$2 < \binom{3}{2}$	**0**	
$2 \geq \binom{2}{2}$	**1**	$2 - \binom{2}{2} = 1$
$1 \geq \binom{1}{1}$	**1**	$1 - \binom{1}{1} = 0$
$0 < \binom{0}{0}$	**0**	

The resulting sequence is ’11001010100111100001100000100110’. It is the same one as we encoded before.

Our last step is to transfer this sequence to the game board. We have to be careful with the field where we already placed the fox. In this case the fox has the index 8 so we modify the sequence. We mark the position of the fox in the sequence (here with an F) ’11001010 F 100111100001100000100110’. Then, we scan the game board in the order we see in Figure 5.1 to place the geese. We place the geese on fields where the binary sequence has a 1.

After this, the decoding is done. We see the resulting state in Figure 7.4.

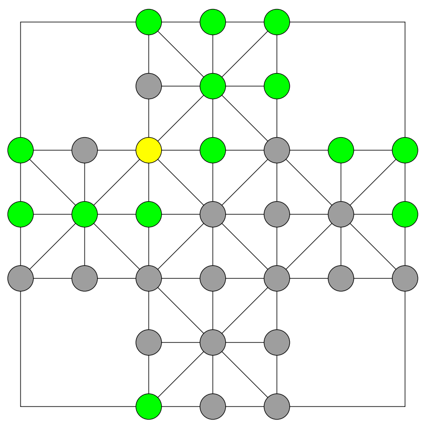


Figure 7.4: Decoding result of the “Fox and Geese” example state

7.4 Minimum and Maximum Code for “Fox and Geese”

In order to estimate the range of the code we need to determine the minimum and the maximum code for the game “Fox and Geese” in this section.

7.4.1 Minimum Code for “Fox and Geese”

To get the smallest possible code for “Fox and Geese” we have to get the smallest code for the three parts the code consists of. We described these three parts in Section 7.1.

The code for the player can either be zero or one. That is why the minimum code is of a state of the geese player and has the player code 0.

To get the smallest geese code we have to select a state with just 6 geese which is the minimum for this game. The encoding of binary sequences with all including ones at its end results in the code 0. The value that we have to add to this code, in order to know how many ones are included in this sequence, is also 0 because 6 geese are the minimum for each state. We show the calculation later in this section.

To get the smallest fox code, we have to place the fox on the field with index 0. We see the numbered game board in Figure 5.1.

We see the state with the minimum code in Figure 7.5. It is a state of the geese player.

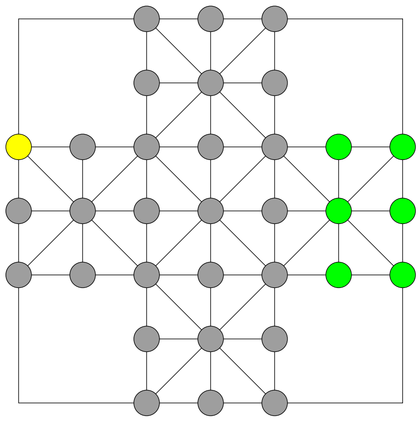


Figure 7.5: State of “Fox and Geese” that results in the minimum code

7.4.1.1 Calculation of the Minimum Code

As the state is a state of the geese player the player code is 0:

$$C_p = 0$$

The fox is placed on the field with the index 0, thus the fox code is 0:

$$C_f = 0$$

The binary sequence for the encoding of the geese is '00000000000000000000000000111111'. It has a length of 32 and includes 6 ones. We encode it using the algorithm described in Section 5.3. The starting point is $\binom{32}{6}$.

$$C_g =$$

$$0 * \binom{31}{6} + 0 * \binom{30}{6} + 0 * \binom{29}{6} + 0 * \binom{28}{6} +$$

$$0 * \binom{27}{6} + 0 * \binom{26}{6} + 0 * \binom{25}{6} + 0 * \binom{24}{6} +$$

$$0 * \binom{23}{6} + 0 * \binom{22}{6} + 0 * \binom{21}{6} + 0 * \binom{20}{6} +$$
$$0 * \binom{19}{6} + 0 * \binom{18}{6} + 0 * \binom{17}{6} + 0 * \binom{16}{6} +$$
$$0 * \binom{15}{6} + 0 * \binom{14}{6} + 0 * \binom{13}{6} + 0 * \binom{12}{6} +$$
$$0 * \binom{11}{6} + 0 * \binom{10}{6} + 0 * \binom{9}{6} + 0 * \binom{8}{6} +$$
$$0 * \binom{7}{6} + 0 * \binom{6}{6} + 1 * \binom{5}{6} + 1 * \binom{4}{5} +$$
$$1 * \binom{3}{4} + 1 * \binom{2}{3} + 1 * \binom{1}{2} + 1 * \binom{0}{1} =$$
$$1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 + 1 * 0 = 0$$

In order to add the information of how many ones the binary sequence consists of, we have to add the result of the following sum to the geese code:

$$\sum_{k=6}^{n_g-1} \binom{32}{k}$$

n_g is the number of geese, in this case 6. That means the value we have to add is 0.

As all parts of the code are 0, the resulting code is also 0.

7.4.2 Maximum Code for “Fox and Geese”

In order to get the maximum code for the game “Fox and Geese” we have to find the maximum code for the player, the geese and the fox.

The maximum code for the player is one. That means the state has to be a state of the fox player.

The maximum fox code is 19. Thus we have to place the fox on the field with index 19. We see the fields and their indices in Figure 5.1.

To get the maximum geese code we have to place the maximum number of geese - 15 geese - on the board. To get the largest code we have to create a binary sequence that has all its ones directly at the beginning. We use the algorithm described in Section 5.3 to encode it. In addition we have to add the information of how many ones the sequence consists of. Since we use the maximum number of geese that can be on the board, this value is also the highest value that is possible. We see the respective calculation in the following subsection.

We see the state with the maximum code in Figure 7.6. It is a state of the fox player.

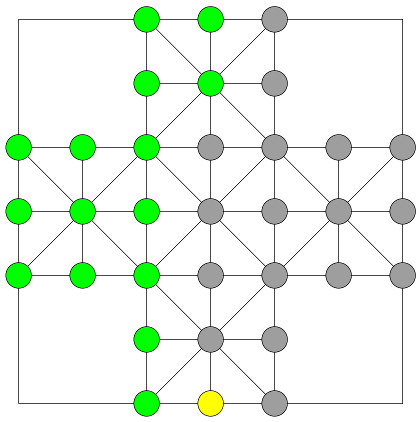


Figure 7.6: State of “Fox and Geese” that results in the maximum code

7.4.2.1 Calculation of the Maximum Code

As already mentioned above, the player code for this state is 1, because it is the fox player’s turn:

$$C_p = 1$$

The fox is placed on the field with the index 19. Thus the fox code is as follows:

$$C_f = 19$$

The sequence for the encoding of the geese is ’11111111111111100000000000000000’. It has 15 geese and a length of 32. We encode it with the algorithm described in Section 5.3. The starting point is $\binom{32}{15}$.

$$C_g =$$

$$1 * \binom{31}{15} + 1 * \binom{30}{14} + 1 * \binom{29}{13} + 1 * \binom{28}{12} +$$

$$1*\binom{27}{11}+1*\binom{26}{10}+1*\binom{25}{9}+1*\binom{24}{8}+$$
$$1*\binom{23}{7}+1*\binom{22}{6}+1*\binom{21}{5}+1*\binom{20}{4}+$$
$$1*\binom{19}{3}+1*\binom{18}{2}+1*\binom{17}{1}=$$
$$300\,540\,195+145\,422\,675+67\,863\,915+30\,421\,755+$$
$$13\,037\,895+5\,311\,735+2\,042\,975+735\,471+$$
$$245\,157+74\,613+20\,349+4\,845+$$
$$969+153+17=565\,722\,719$$

We have to add the information of how many ones are included in the sequence. Therefore we have to add the following sum to the geese code:

$$\sum_{k=6}^{n_g-1}\binom{32}{k}$$

n_g is the number of ones in the sequence, in this case 15. The value we have to add is 1 280 977 908.

$$C_g = 565\,722\,719 + 1\,280\,977\,908 = 1\,846\,700\,627$$

To get the full code we have to combine the three values:

$$C = (C_g * num(C_f) + C_f) * 2 + C_p$$
$$C = (1\,846\,700\,627 * 20 + 19) * 2 + 1 = 73\,868\,025\,119$$

The maximum code for "Fox and Geese" is 73 868 025 119. We need 37 bits to store this information. If we compare these values with the numbers we've calculated for "Fuchs und Henne" in Section 5.5.2.1, we see that the the values for "Fox and Geese" are considerably lower.

To sum up, in this chapter we created an encoding method for states of "Fox and Geese". We created unique codes for the current player, the position of the fox and the positions of the geese. We combined these three codes to get the unique code for the state. At the end of this state we encoded an example state of “Fox and Geese” and decoded the result again. Finally, we determined the minimum and the maximum code for the game “Fox and Geese”.

In the next chapter, we will look at some variations of the games “Fuchs und Henne” and “Fox and Geese”.

8 Variations of the Games

In this chapter we look at some variations of the games “Fuchs und Henne” and “Fox and Geese”. We strongly solve this variations and compare the results.

As already mentioned above, we sometimes perceive asymmetric games as unfair, due to unequal opportunities and conditions. To overcome this we compare some variations of the games by solving them strongly. Our goal is to find a variation that is as fair as possible. Of course, this is possible only to a limited extent, because the computer program lacks human aspects. Thus, we just compare the number of possible winning states for the players and use this as a meassure for the fairness of the game.

8.1 Variations of “Fuchs und Henne”

In this section we look at variations of the game “Fuchs und Henne”. As mentioned in Section 2.2, there are many different variations of this game. In this section we vary the number of hens in the initial state from 9, which is the minimum value for this game, to 20. In Figure 8.1 we see the initial states for this variations. In Chapter 4 we found an upper bound for the number of different states for this game. We use the same approach for the variations of this game. We strongly solve the variations and compare the results. In Section 3.4 we described the game framework that we use. The encoding is the same as described in Chapter 5.

In order to get the upper bound of states of a variation of “Fuchs und Henne” we can use the following formula:

$$(\sum_{i=9}^{x} \binom{31}{i} * 281 + \sum_{i=9}^{x} \binom{32}{i} * 20) * 2$$

The parameter x is the number of hens of the variation. We start the index of this sums at 9, because this is the game’s minimum number of hens. The first sum represents the situations with two foxes. We multiply this value by 281 which is the number of different positions for two foxes. The second sum represents the situations with just one fox. We multiply it by 20, because this is the number of possible positions for this one fox. We then multiply this sums by 2, because each state can be the state of the fox player or of the hen player. We described this formula for the main version of this game in Section 4.5.2.3.

No. of hens	Calc. upper bound of states	Exact number of states	States hens win	States fox wins	DB size
9	12 451 914 150				
10	39 958 320 480				
11	92 705 140 710				
12	181 046 589 360				
13	310 855 761 510				
14	478 745 764 560				
15	670 278 262 950				
16	863 225 068 140				
17	1 034 886 555 990				
18	1 169 658 208 140				
19	1 262 862 887 190				
20	1 319 480 441 820				

Table 8.1: States of "Fuchs und Henne" with different numbers of hens

We do not have results for the variations of "Fuchs und Henne" so far. Due to the complexity of this game, the computer program we developed for solving "Fuchs und Henne" strongly is still running at the time of this writing, even for the variation with the minimum number of 9 hens. This is why Table 8.1 is not filled with data so far, except for the values we calculated for the upper bound of states.

8.2 Variations of "Fox and Geese"

In this section, we look at variations of the game "Fox and Geese". We vary the number of geese at the initial state from 4 to 15. We show the arrangements of the initial states for this variations in Figure 8.2. The method to calculate the upper bound of states for this variations is described in Chapter 6 for the main version of this game. Thus, we strongly solve these variations and compare the results. We use the game framework which is described in Section 3.4 and the encoding we developed in Chapter 7.

In order to find the upper bound of states of a variation of "Fox and Geese" we use the following formula:

$$\sum_{i=4}^{x} \binom{32}{i} * 20 * 2$$

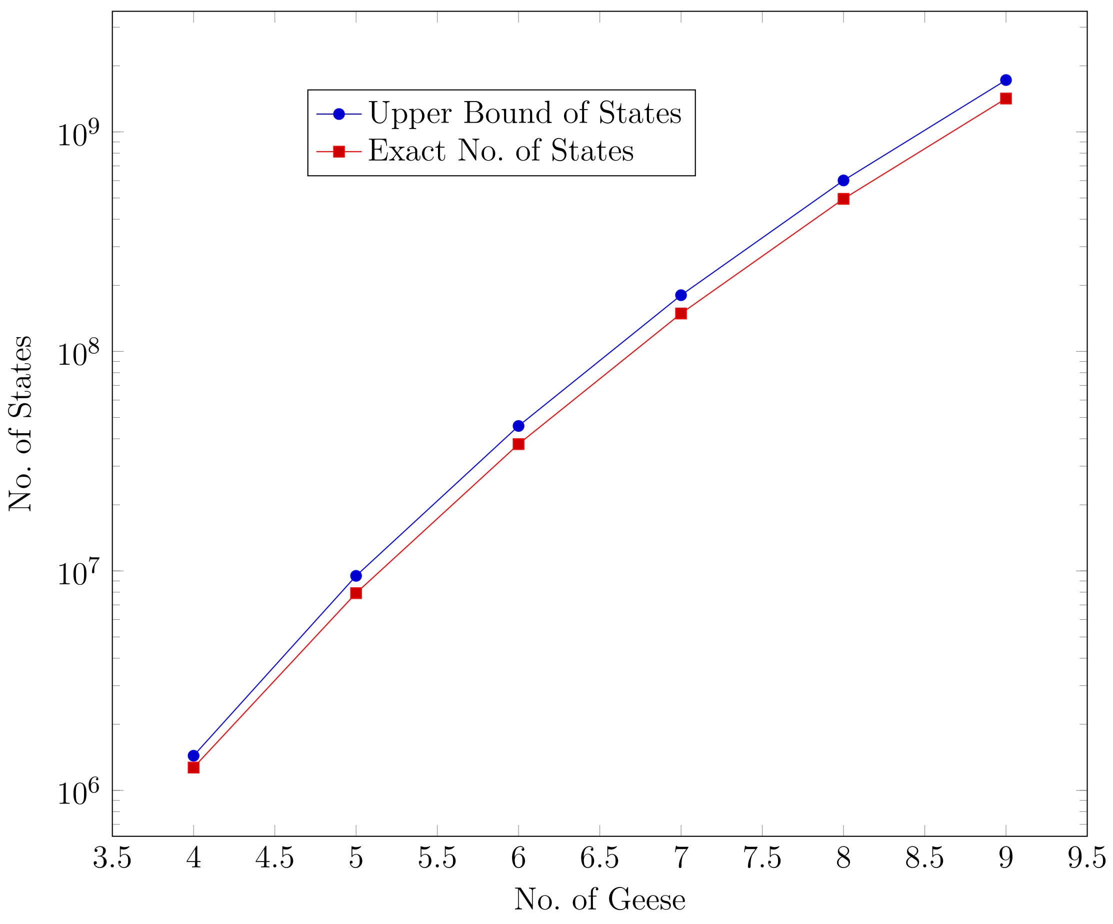
The parameter x is the number of geese of the variation. The index of the sum is starting at 4, because this is the minimum number of geese for this game. We multiply the whole sum by 20 as this is the number of different positions for the fox. Then we multiply this sums by 2, because each state can occur during either player’s turn. We described this formula in detail in Section 6.6.

No. of geese	**Calc. upper bound of states**	**Exact number of states**	**States geese win**	**States fox wins**	**DB size**
4	1 438 400	1 270 717	0	666 912	13 MB
5	9 493 440	7 919 632	0	1 644 004	83 MB
6	45 741 120	37 820 778	16	3 478 337	431 MB
7	180 375 360	148 821 122	416	6 975 888	1.3 GB
8	601 107 360	495 481 299	5 224	12 936 253	4.7 GB
9	1 723 059 360	1 419 189 540	52 859	21 781 681	15 GB
10	4 303 548 960				
11	9 464 528 160				
12	18 496 241 760				
13	32 391 185 760				
14	51 248 609 760				
15	73 877 518 560				

Table 8.2: States of “Fox and Geese” with different numbers of geese

We do not have the results for all variations of this game so far at the time of this writing. Due to the complexity of this game, the computer program we developed for solving “Fox and Geese” strongly is still running at the time of this writing for the remaining variations.

Upper bound and exact number of states
of different variations of “Fox and Geese”



8.3 Results

In this work we developed an encoding for the games “Fuchs und Henne” and “Fox and Geese”. We used a method that is as flexible as possible and thus can also be applied for the different variations of these games.

We have already solved a few variations of “Fox and Geese” while writing this thesis. Since the number of states increases exponentially as we increase the number of hens or geese in the initial state, the computation time of the program also increases exponentially.

For the variation of “Fuchs und Henne” with the minimum number of 9 hens computing the creation of all states without a classification of them already took nearly one month. The classification process for these states is still running. This is why we do not have results for this game at the time of this writing.

For “Fox and Geese” we already have results for the variations from 4 to 9 geese. In this variations the fox player has clearly better chances to win the game.

The tables above show that a different number of hens or respectively geese changes the result significantly. If we compare the results of the winning states for both players, we can identify the variant with the smallest difference between the values. That might be the fairest version for the game. However, since the computer behaves differently than a human being, we can only assume that this variant will be perceived as fair.

Additionally we can use the databases of the solved variations, for example to play the games against the computer.

8.4 Future Work

The main goal of this work was to develop a method to solve different variations of asymmetric two-player games, especially the games “Fuchs und Henne” and “Fox and Geese”. Due to the high complexity of these games, we were able to strongly solve only a few of these variations completely.

We can use the idea of the presented encoding also for other asymmetric two-player games. The databases of the already solved versions can be used for playing against the computer. As further work it would be interesting to create a user interface for playing the games against the computer, because so far it is just possible to play the games on the command line.

We were not able to solve all variations we described in this work strongly, thus it is still unknown which variation could be considered the fairest. The goal of the thesis was to develop the framework and the implementations, so this task can be fulfilled in the near future, but after the time of finishing this thesis.

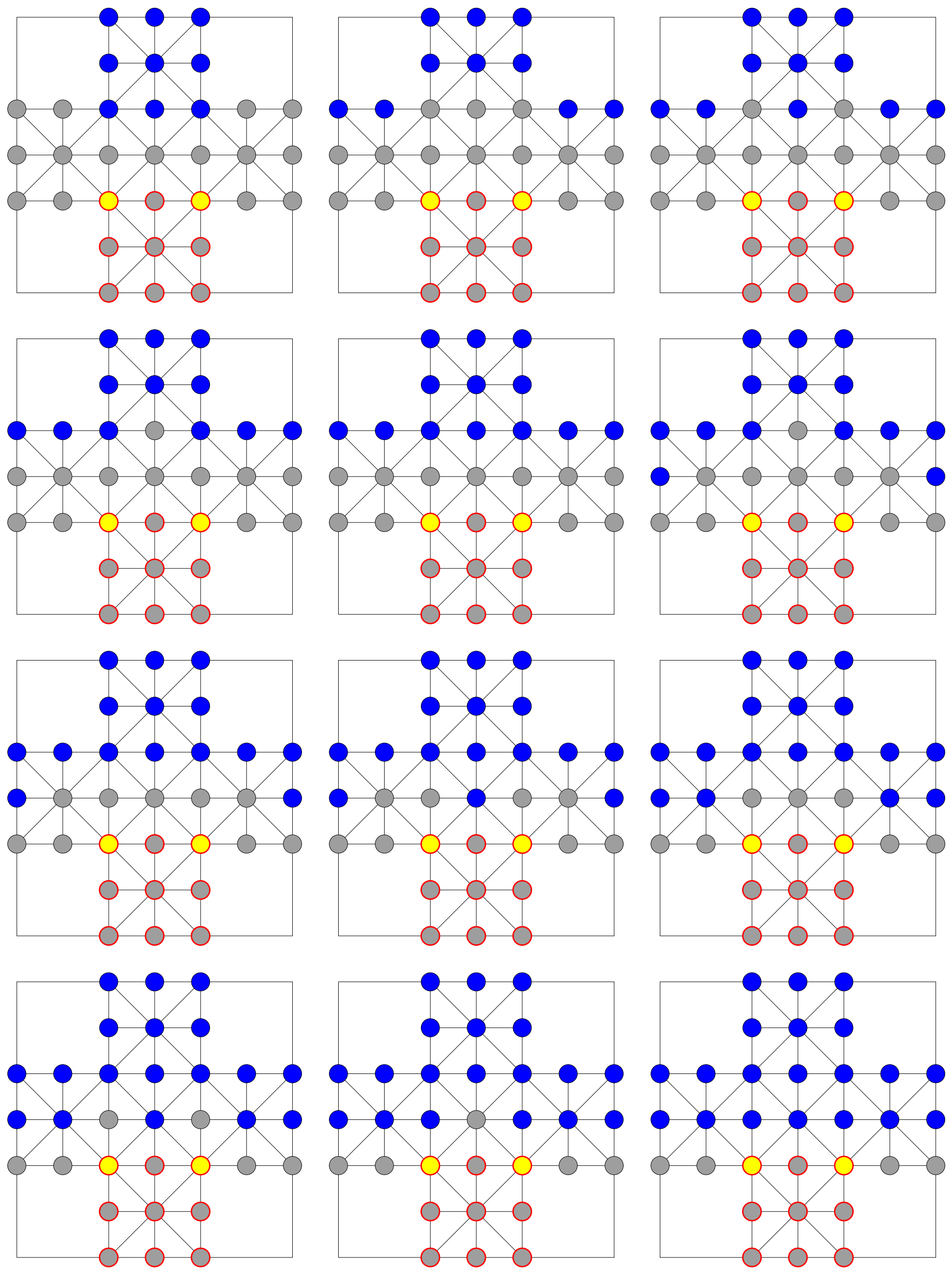


Figure 8.1: Initial states of variations of “Fuchs und Henne” with 9 to 20 hens

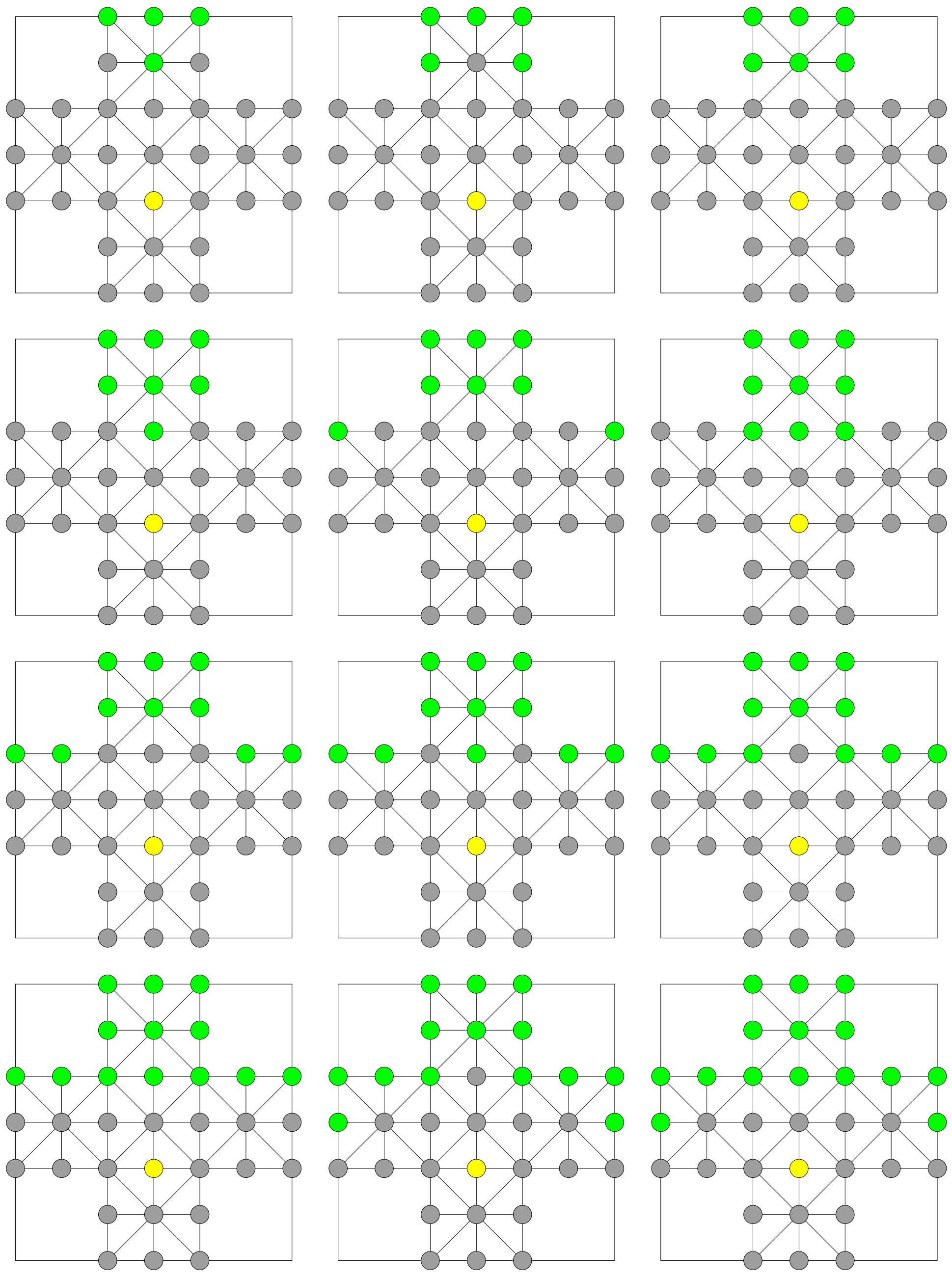


Figure 8.2: Initial states of variations of “Fox and Geese” with 4 to 15 geese

Bibliography

[1] E. Aarseth, S. M. Smedstad, and L. Sunnanå. “A multidimensional typology of games”. In: (2003). URL: http://www.digra.org/wp-content/uploads/digital-library/05163.52481.pdf (cit. on p. 1).

[2] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* 1994 (cit. on pp. 20, 21).

[3] R.C. Bell. *Board and Table Games from Many Civilizations.* Board and Table Games from Many Civilizations Bd. 1-2. Dover Publications, 1979. ISBN: 9780486238555. URL: https://books.google.at/books?id=5viitl9PvBoC (cit. on pp. 5, 6).

[4] S. Cheng, D. M. Reeves, Y. Vorobeychik, and M. P. Wellman. “Notes on equilibria in symmetric games”. In: (2004), pp. 71–78 (cit. on p. 2).

[5] E. Demaine and R. Hearn. “Playing Games with Algorithms: Algorithmic Combinatorial Game Theory”. In: *CoRR* cs.CC/0106019 (2001). URL: http://arxiv.org/abs/cs.CC/0106019 (cit. on p. 2).

[6] Masters Traditional Games. *The Rules of Fox & Geese.* https://www.mastersofgames.com/rules/fox-geese-rules.htm. Accessed: 2018-03-08 (cit. on p. 13).

[7] E. Glonnegger. *Das Spiele-Buch: Brett- und Legespiele aus aller Welt; Herkunft, Regeln und Geschichte.* Drei-Magier-Verlag, 1999. ISBN: 9783980679206. URL: https://books.google.at/books?id=Lzo3YAAACAAJ (cit. on p. 5).

[8] Winkler Schulbedarf GmbH. *FUCHS und HENNE, WINKLER - Nr. 100685.* https://www.winklerschulbedarf.com/Documents/Anleitungen_Werkpackungen/PDF_Ger/100685.pdf. Accessed: 2018-03-08 (cit. on p. 9).

[9] S. Helmfrid. *Hnefatafl - the Strategic Board Game of the Vikings.* http://hem.bredband.net/b512479/Hnefatafl_by_Sten_Helmfrid.pdf. Published 2005-04-23, Accessed: 2018-03-21 (cit. on p. 5).

[10] H. Herik, J. Uiterwijk, and J. van Rijswijck. “Games solved: Now and in the future”. In: 134 (Jan. 2002), pp. 277–311 (cit. on pp. 17, 20, 21).

[11] E. Magnússon. *Grettis Saga. The Story of Grettir the Strong, translated from the Icelandic by E. Magnússon and W. Morris.* 1869. URL: https://books.google.at/books?id=GtdUAAAAcAAJ (cit. on p. 6).

[12] F. Mäyrä. *An Introduction to Game Studies.* SAGE Publications, 2008. ISBN: 9781412934459. URL: https://books.google.at/books?id=iI0kAQAAIAAJ (cit. on p. 5).

[13] L. Orbán. *Schach für Anfänger: Alles über das "königliche Spiel". Regeln, Strategien, Spielzüge. Leicht verständlich erklärt.* humboldt, 2010. ISBN: 9783869108179. URL: https://books.google.at/books?id=HTUuBQAAQBAJ (cit. on p. 2).

[14] S. L. Patterson. "Game on: Medieval players and their texts". dissertation. University of British Columbia, 2017. URL: https://open.library.ubc.ca/cIRcle/collections/ubctheses/24/items/1.0345627 (cit. on p. 5).

[15] O. Regenfelder. "Implementation of Games in a Generalized Framework for Solving Combinatorial Games using the State Space". Bachelor Thesis. Faculty of Computer Science, Graz University of Technology, May 2010 (cit. on pp. 22, 39).

[16] W. Roth. "Entwicklung einer KI für das Spiel Pentago". Bachelor Thesis. Faculty of Computer Science, Graz University of Technology, 2012 (cit. on pp. 18, 21).

[17] J. Schalkwijk. "An algorithm for source coding". In: *IEEE Transactions on Information Theory* 18.3 (May 1972), pp. 395–399. ISSN: 0018-9448. DOI: 10.1109/TIT.1972.1054832 (cit. on pp. 48, 49).

[18] P.D. Straffin. *Game Theory and Strategy.* Anneli Lax New Mathematical Library Bd. 36. Mathematical Association of America, 1993. ISBN: 9780883856376. URL: https://books.google.at/books?id=3TB3m3RvAlcC (cit. on p. 1).

[19] R. Winkler. *Fuchs und Henne nach Spielregeln spielen - so geht's.* http://www.helpster.de/fuchs-und-henne-nach-spielregeln-spielen-so-geht-s_81479. Accessed: 2018-03-08 (cit. on p. 9).