Universiteit Leiden
The Netherlands

Opleiding Informatica

Cracking the Mastermind Code

Sylvester de Graaf

Supervisors:
Rudy van Vliet & Jan van Rijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

31/07/2019

Abstract

Mastermind is a game where one or multiple code breakers try to crack a four-colour code in a certain number of guesses. More than forty years ago, Knuth described an algorithm that needs at most five guesses to crack all codes in a *Mastermind* game with six different colours and the option to use the same colour more than once in the code. Nowadays, *Mastermind* is also played with more than six different colours, as well as the rule to use each colour at most once in the code. In this thesis we try to find a better algorithm to solve this variety of *Mastermind* games, and examine what the maximum number of guesses would be to solve all the possible codes for these games. Different algorithms will also be compared with each other, to see how much they differ in performance.

Contents

1 Introduction **1**

2 Algorithms **4**
2.1 Five-Guess Algorithm 4
2.2 Random Algorithm 6
2.3 MiniMax Only Knuth Codes Algorithm 7
2.4 Secondary MiniMax Knuth Codes Algorithm 8
2.5 Every Possible Branch Only Knuth Algorithm 9
2.6 Every Possible Branch Algorithm 11
2.7 Random Knuth Codes Algorithm 12
2.8 Other Algorithms 12

3 Experiments **13**
3.1 Possible Best Algorithms 13
3.2 Walking Down Every Possible Branch 14
3.3 Worst Case Scenario 15
3.4 Other Algorithms 18

4 Best Algorithm Results **19**

5 Conclusions and Further Research **23**

References **24**

1 Introduction

Mastermind is a code-breaking game for two or more players. One player, the code maker, makes a code consisting of m pins, while all the other players are code breakers. Code breakers take turns to try to crack the *Mastermind* code, which is made by the code maker [Wik].

The first board game version of *Mastermind* was made in 1970 by Mordecai Meirowitz [Mei].
In the physical game, the *Mastermind* code is formed using $m = 4$ pins, and the feedback for this guess is represented by two sliders, a red one and a white one. The red one indicates the correct colour in the correct place, while the white one tells us the number of correct colours in the wrong place. For the ease of notation, we use digits $1, 2, ..., n-1, n$ to denote the colours. A possible notation for a code could be 1123.

Most versions of *Mastermind* have a limit of eleven rounds to crack the code. After every guess, the code maker will tell us two things:

- The number of correct colours in the correct place (*cp*).
- The number of colours which are in the code, but are in the wrong place (*wp*).

Below is an example gameplay for a *Mastermind* game with two code breakers and where the *Mastermind* code is 1214:

1. The first code breaker starts with guess 5111.
2. The code maker gives the following feedback: 1 *cp* and 1 *wp*. The code breaker gets back 1 *cp*, because the second 1 in his guess is in the correct place. There is one other 1 in the code, but since the code breaker did not guess it in the correct place, he gets back 1 *wp*.
3. The second code breaker follows up with guess 1313.
4. The second code breaker receives the feedback: 2 *cp* and 0 *wp*.

After these steps the first code breaker makes a guess again, then the second code breaker, etc. This continues until either the *Mastermind* code has been cracked or the limit number of guesses has been reached.

In this thesis we study algorithms for the case with one code breaker, trying to crack the code in as few guesses as possible.
When there is only one code breaker, he wins when he cracks the code within the specified number of guesses (this is when we get m *cp* back as feedback).
If there are multiple code breakers, the first code breaker who guesses the code within the specified number of guesses wins.

In case of two or more code breakers, every guess, no matter by which code breakers, counts as one guess towards the specified number of guesses.
If the code is not cracked within the specified number of guesses, the code maker wins the game.

Figure 1: Mastermind game with six colours.
Source: https://en.wikipedia.org/wiki/Mastermind_(board_game)#/media/File:Mastermind.jpg.

Mastermind is a game that uses a minimum of six different colours, and the following rules can be applied for a game:

1. It is allowed to use the same colour multiple times in the *Mastermind* code. This game variant will be referred to as ***same colour allowed (sca)***.

2. It is not allowed to use the same colour multiple times in the *Mastermind* code. This game variant will be referred to as ***same colour not allowed (scna)***.

Originally, *Mastermind* was played with six colours with the *sca* variant.
Nowadays the physical board game that is sold, has eight different colours and is played with both variants (*sca* and *scna*). To represent what sort of *Mastermind* game we play, we will write *Mastermind(n, m, a/na)*, where *n* is the number of colours we can use in the code, *m* is the length of the code we need to crack, and *a/na* tells us if it is allowed to use the same colour multiple times in the code.

Back in 1977, Donald Knuth described an algorithm that can crack each of the $6^4 = 1296$ possible codes of a *Mastermind(6, 4, a)* game, in five guesses or less [Knu77]. We can also prove that it is not possible to crack every code for this game in four guesses or less:

1. We start with the first guess. Knuth describes a worst case guess, which leaves at most 256 codes that can still be the *Mastermind* code after this guess.

2. We can theoretically separate 255 of these codes equally into 13 different subsets based of the feedback (there is one code that returns feedback 4 *cp* and 0 *wp* in the next guess).

3. After the second guess, there will be at least one subset that has at least $\lceil (255/13) \rceil = 20$ possible codes that can still be the *Mastermind* code.

4. After the third guess, there will be at least one subset with at least $\lceil (20-1)/13 \rceil = 2$ possible codes left, that can still be the *Mastermind* code.

5. If we guess the wrong code in guess four, we need to take an extra guess. Therefore we cannot crack every code in four guesses or less.

In the sense of using the smallest number of guesses possible, we can say that Knuth's algorithm is optimal. For games with more than six colours, it is not known what the smallest number of guesses is to crack any possible code. It has been proven that the *Mastermind* satisfiability problem is NP-complete by Jeff Stuckman and Quo-Qiang Zhang in 2005 [SZ05]. Some earlier research also shows the speed and effectiveness of a genetic algorithm, a simulated annealing algorithm and a random search algorithm to solve a game of *Mastermind* [BHG+96].

Nowadays *Mastermind* is played with more than six colours. For these games, we do not know what the best algorithm is to solve all possible codes, and what the maximum number of guesses would be if we tried to crack one of these codes.

In this thesis we try to find the answer to the following two questions:

1. What is the best algorithm to solve an *sca Mastermind* game with six colours?

2. How many guesses would this algorithm need for *Mastermind* games of both *sca* and *scna* variants, with six up till and including ten different colours?

In principle it is possible to make codes of less than or more than four pins, as well as to play games with less than six or more than ten different colours. However, these games are not played in practice. That is why we only look at the variants that actually appear in practice.

In Chapter 2 we describe the algorithms we test to solve the first question of our thesis.
In Chapter 3 we test the performance of our algorithms for *sca Mastermind* games with six colours.
In Chapter 4 we answer the second question of our thesis.
Finally, in Chapter 5 we discuss the conclusions of this thesis and the future work that can still be done.

2 Algorithms

For this thesis, we decided to use a variety of different algorithms, to see which are the best. In the description of these algorithms, we use the following terms:

- ***Knuth codes***: The codes that may still be the secret code, given the guesses made so far and the feedback for these guesses.
- ***Possible codes***: The codes that have not been guessed yet.
- ***Total codes***: Total possible codes before the algorithm starts (e.g. 1296 for an *sca Mastermind* game with six colours and four pins).
- ***Guess codes***: The current possible codes with the lowest MiniMax value (explained in Section 2.1).

We decided that, whenever a *Mastermind* code has been made, it cannot be changed anymore.
Bestavros & Belal have done their own research, where they let the code maker change the *Mastermind* code after every guess. They describe two methods of how the code maker changes the *Mastermind* code [BB86].

2.1 Five-Guess Algorithm

In 1977, Donald E. Knuth described an algorithm that can solve *Mastermind(6, 4, a)* games in five guesses or less. As this could be the algorithm we are looking for to solve our first thesis question, we decided to include it into our research. His algorithm uses an auxiliary function MiniMax, which is described below:

```
# This program gives a list of guesses that are the best to use as next guess.
# This list is sorted from lowest number to highest number,
# e.g. 1122 comes before 1133.

function MiniMax(knuth_codes, possible_codes){
    FOR each code in possible_codes:
        FOR each code_to_crack in knuth_codes:
            feedback = GuessCode(code_to_crack, code)
            times_found[feedback]++
        ENDFOR
        maximum = GetMaxValue(times_found)
        scores[code] = maximum
    ENDFOR

    minimum = GetMinValue(scores)
```

```
    FOR each code in possible_codes:
        IF scores[code] == minimum:
            add code to guess_codes
        ENDIF
    ENDFOR

    return guess_codes
}

function GuessCode(code_to_crack, code){
    compare code_to_crack and code

    return number of cp and wp
}
```

Below is the pseudocode of the Five-Guess Algorithm:

```
# This program describes the Five–Guess Algorithm made by Knuth

function Mastermind(n, m, a/na){
    total_codes = GetTotalCodes(n, m, a/na)
    knuth_codes = total_codes
    possible_codes = total_codes
    mastermind_code = GetMastermindCode(total_codes)

    WHILE mastermind_code has not been cracked:
        code = GetCode(knuth_codes, possible_codes)
        feedback = GuessCode(mastermind_code, code)

        IF feedback == (m cp and 0 wp):
           mastermind_code has been cracked
        ELSE:
            PruneList(code, feedback, knuth_codes)
        ENDIF
    ENDWHILE

    return number of guesses needed to crack the code
}

function PruneList(last_guess, feedback, knuth_codes){
    FOR each code in knuth_codes:
        retrieved_feedback = GuessCode(code, last_guess)
```

```
        IF retrieved_feedback != feedback:
            remove code from knuth_codes
        ENDIF
    ENDFOR
}

function GetMastermindCode(total_codes){
    return a random code from total_codes
}

function GetTotalCodes(n, m, a/na){
    return list of all codes that can be made
    with the rules specified by the user.
}

function GetCode(knuth_codes, possible_codes){
    guess_codes = MiniMax(knuth_codes, possible_codes)
    code = GetGuessCodeFromList(knuth_codes, guess_codes)

    remove code from possible_codes

    return code
}

function GetGuessCodeFromList(knuth_codes, guess_codes){
    return the first knuth_code we can find in guess_codes
    return the first code in guess_codes if we could not find a knuth_code
}
```

2.2 Random Algorithm

We made an algorithm that randomly tries codes, until the game has been won. We made this algorithm, because we wanted to see what happens if we just go through all the codes without making use of the code maker's feedback.

```
# This program describes an algorithm that solves a game of Mastermind.
# It randomly goes down all possible codes,
# until the Mastermind code has been cracked.

function Mastermind(n, m, a/na){
    total_codes = GetTotalCodes(n, m, a/na)
```

```
    possible_codes = total_codes
    mastermind_code = GetMastermindCode(total_codes)

    WHILE mastermind_code has not been cracked:
        code = GetCode(possible_codes)
        feedback = GuessCode(mastermind_code, code)

        IF feedback == (m cp and 0 wp):
            mastermind_code has been cracked
        ENDIF
    ENDWHILE

    return number of guesses needed
}

function GetCode(possible_codes){
    get one random code from possible_codes
    remove this code from possible_codes
    return code
}

# The functions GuessCode, GetTotalCodes and GetMasterMindCode
# have been described in Section 2.1
```

2.3 MiniMax Only Knuth Codes Algorithm

An interesting aspect of Knuth's algorithm is, that it sometimes takes a code as guess that is not in *Knuth codes*. We were interested what would happen if we modified this algorithm, so that the MiniMax function only stores the *Knuth codes* in *guess codes*.

```
# This program solves a game of Mastermind by using MiniMax and pruning.
# The difference between this algorithm and the Five-Guess Algorithm is
# that this algorithm does not store the possible_codes in guess_codes
# that are not knuth_codes

function MiniMax(knuth_codes){
    FOR each code in knuth_codes:
        FOR each code_to_crack in knuth_codes:
            feedback = GuessCode(code_to_crack, code)
            times_found[feedback]++
        ENDFOR
```

```
            maximum = GetMaxValue(times_found)
            scores[code] = maximum
        ENDFOR

        minimum = GetMinValue(scores)
        FOR each code in knuth_codes:
            IF scores[code] == minimum:
                add code to guess_codes
            ENDIF
        ENDFOR

        return guess_codes
    }

    # The function Mastermind and all the other function it uses,
    # are the same as the ones described in Section 2.1
```

The *Mastermind* function of this algorithm works the same as the Five-Guess Algorithm. We can see in the pseudocode above, that MiniMax only iterates over the *Knuth codes*, instead of over all *possible codes*. This makes sure our *guess codes* only consist of *Knuth codes*.

2.4 Secondary MiniMax Knuth Codes Algorithm

Knuth's Five-Guess Algorithm only looks at the minimum value. We wondered if there would be an improvement if the algorithm would try to use a *Knuth code* (if it exists) as guess with the second lowest value, when there was no *Knuth code* for the lowest value in *guess codes*.

```
    # This program solves a game of Mastermind by using MiniMax and pruning.
    # The difference between this algorithm and the Five-Guess Algorithm is
    # that this algorithm also looks for knuth_codes which have the second
    # best minimum value, when knuth_codes lack in guess_codes for the
    # best minimum value.

    function MiniMax(knuth_codes, possible_codes){
        FOR each code in possible_codes:
            FOR each code_to_crack in knuth_codes:
                feedback = GuessCode(code_to_crack, code)
                times_found[feedback]++
            ENDFOR
            maximum = GetMaxValue(times_found)
            scores[code] = maximum
```

```
    ENDFOR

    minimum = GetMinValue(scores)
    minimum2 = GetSecondMinValue(scores, minimum)
    FOR each code in possible_codes:
        IF scores[code] == minimum:
            add code to guess_codes
        ELSE IF scores[code] == minimum2:
            add code to guess_codes
        ENDIF
    ENDFOR

    return guess_codes
}

function GetSecondMinValue(scores, minimum){
    return the lowest value in list scores that is not equal to minimum
}

# The function Mastermind and all the other function it uses,
#  are the same as the ones described in Section 2.1

# Note that first the possible_codes with the
# lowest minimum value are stored in guess_codes,
# followed by those with the second lowest minimum value.

# In case there is no knuth_code in guess_codes, the algorithm will
# still take the first possible_code that has the lowest minimum value.
```

2.5 Every Possible Branch Only Knuth Algorithm

In Knuth's Five-Guess Algorithm, he uses the first *Knuth code* (or just the first code by lack of *Knuth codes* in *guess codes*) in *guess codes*. We wondered how important it is for the algorithm to actually use the first *Knuth code* in *guess codes* as next guess, instead of an arbitrary *Knuth code* in *guess codes*. Rather than running the algorithm a number of times with random *Knuth codes* from *guess codes*, we haven chosen to try each of these Knuth codes as a next guess, and see how many guesses we need to finish the game in each of the cases. That is, we try all possible branches in the search tree. In other words, we walk down every possible path in the tree.

```
# This program walks down every knuth_code for a guess if
```

```
# list guess_codes contains codes that are in knuth_codes.
# Otherwise it takes the first code in guess_codes.

function Mastermind(n, m, a/na){
    total_codes = GetTotalCodes(n, m, a/na)
    FOR each mastermind_code in total_codes:
        knuth_codes = total_codes
        possible_codes = total_codes
        code = GetCode(knuth_codes, possible_codes)
        feedback = GuessCode(mastermind_code, code)

        IF feedback == (m cp and 0 wp):
            program needed one guess to crack the code
        ELSE:
            PruneList(code, feedback, knuth_codes)
            LoopOverCodes(knuth_codes, possible_codes, mastermind_code)
        ENDIF
    ENDFOR
}

function LoopOverCodes(knuth_codes, possible_codes, mastermind_code){
    codes = GetCodeList(knuth_codes, possible_codes)
    FOR each code in codes:
        For this iteration, remove code from possible_codes
        feedback = GuessCode(mastermind_code, code)
        IF feedback == (m cp and 0 wp):
            print the number of guesses we made
            until we cracked the mastermind_code
        ELSE:
            PruneList(code, feedback, knuth_codes)
            LoopOverCodes(knuth_codes, possible_codes, mastermind_code)
        ENDIF
    ENDFOR
}

function GetCodeList(knuth_codes, possible_codes){
    guess_codes = MiniMax(knuth_codes, possible_codes)
    codes= GetGuessCodeFromList(knuth_codes, guess_codes)

    return codes
}

function GetGuessCodeList(knuth_codes, guess_codes){
```

```
    return all the knuth_codes we can find in guess_codes
    return the first code in guess_codes if we could not find a knuth_code
}

# The functions GuessCode, GetTotalCodes, GetCode, and PruneList
# have been described in Section 2.1
# In PruneList possible_codes and knuth_codes
# are only removed from the local list
```

We used a fixed code to make the first guess. This is, because a *Mastermind(6, 4, a)* game has 90 first best guesses, which all consist of two times two different colours (1122, 1212, 1221, 1133, etc.). If we walk over all *total codes* as *Mastermind* code, we would get the same results for each of these 90 codes, only at different paths. Since getting 90 times the same results is not useful, we decided to use 1122 as our fixed first guess.

2.6 Every Possible Branch Algorithm

In Section 2.5 we described that we were interested in what would happen if we walked down all *Knuth codes* entries contained in *guess codes*. We are, however, also curious what happens if we walk down all entries in *guess codes*, if it does not contain a code that exists in *Knuth codes*. With this algorithm and the one described in Section 2.5, we try to find out if it matters that Knuth always takes the first (*Knuth code*) entry of *guess codes*, instead of an arbitrary one.

```
# This program walks down every knuth_code for a guess if
# list guess_codes contains codes that are in knuth_codes.
# Otherwise it walks down all codes in guess_codes.

# The function Mastermind and all the functions it uses, are the same as
# the ones described in Section 2.5

# The only difference is the following function:

function GetGuessCodeList(knuth_codes, guess_codes){
    return all the knuth_codes we can find in guess_codes
    return guess_codes if we could not find a knuth_code
}
```

2.7 Random Knuth Codes Algorithm

Knuth uses the MiniMax function to determine promising guess codes. We wanted to see how important this function is for solving the *Mastermind* code. That is why we made an algorithm that instead of using the MiniMax function to get the best next guess, we take a random code from *Knuth codes* and use that code as our next guess.

```
# This program describes an algorithm that solves a game of Mastermind.
# It randomly selects knuth codes for its guesses,
# until the Mastermind code has been cracked.
# It also makes use of pruning.

function GetCode(knuth_codes, possible_codes){
    get one random code from knuth_codes
    remove this code from possible_codes
    return code
}

# The functions Mastermind, GuessCode, GetTotalCodes,
# PruneList and GetMasterMindCode
# have been described in Section 2.1
```

2.8 Other Algorithms

In the past, there have been several other people beside Knuth, who came up with an algorithm to solve a game of *Mastermind*. One of these algorithms is a genetic algorithm which is based of the creation of a large set of eligible guesses collected throughout the different generations of the genetic algorithm [BGL09]. Another algorithm that has been described uses depth-first backtracking to solve a *Mastermind* game [KL93].

3 Experiments

We tested the algorithms described in Chapter 2, to see which algorithm was the best. There are two ways to define an algorithm for *Mastermind* as the best algorithm:

1. The best algorithm is an algorithm that requires as little guesses as possible in the worst case.

2. The best algorithm is an algorithm that requires as little guesses as possible on average for all codes.

We decided to define the best algorithm in terms of how many guesses it needs in worst case to crack a *Mastermind* code. The lower the number of guess, the better. If there is a tie between two algorithms, we will look at the average number of guesses needed to crack all codes (the lower the better).
We decided to do the testing on a *Mastermind(6, 4, a)* game, since both Knuth [Knu77] and Koyoma & Lai [KL93] use this variant for their research.
In contrast to our research, a research has been done by Barteld Kooi [Koo05] in 2005, where he compares different algorithms and determines the average number of guesses needed by these algorithms to solve a *Mastermind(6, 4, a)* game, given a certain first guess.

In Chapter 4 we test *sca* and *scna Mastermind* games with n = (6, 7, 8, 9, 10) for the algorithm that proves to be the best.

Since the random algorithm (Section 2.2) takes $\frac{1}{2} * 1296 = 648$ guesses on average, and 1296 guesses at most, we decided we will not look further into this algorithm, since it will certainly not be the best algorithm.

3.1 Possible Best Algorithms

We start by looking at the algorithms described in Sections 2.1, 2.3 and 2.4.
We tested these algorithms against all *total codes*, which is 1296 in this case, and counted how many times it took the algorithm $1, 2, ..., r-1, r$ guesses to crack the *Mastermind* code, where r is the biggest number of guesses we needed for these algorithms.

Guesses	FG	MMOKC	2MMKC
1	1	1	1
2	6	12	6
3	62	99	63
4	533	468	528
5	694	662	697
6	0	54	1
Total guesses	5,801	5,828	5,805
Average guesses	4.476	4.497	4.479

Table 1: Number of times the Five-Guess Algorithm (FG), MiniMax Only Knuth Codes Algorithm (MMOKC) and Secondary MiniMax Knuth Codes Algorithm (2MMKC) needed a certain number of guesses to crack a code, as well as the total and average number of guesses to crack all codes.

When we look at table 1, we see a few interesting things.

- All three algorithms have one code that needs only one guess to crack the *Mastermind* code. This is code 1122, since that is the code these algorithms use as their first guess.
- There is an increasing line between the number of guesses needed and the number of times that we have a code that needs that number of guesses.
- The Five-Guess Algorithm and the Secondary MiniMax Knuth Codes Algorithm require almost the same total number of guesses.

We can conclude, that sometimes taking a code that is not in *Knuth codes* as guess, gives us better results. Apparently, in these cases, such a guess yields more, useful information for later steps.
Overall, MMOKC, which only uses codes that are left in *Knuth codes* as guess, performs less good than the others, which sometimes take a code that is not in *Knuth codes* as guess. We can therefore conclude that MMOKC is not the best algorithm we have found so far.
Since the *2MMKC Algorithm* has one code that needs six guesses (this is code 2462), we conclude that the Five-Guess Algorithm is the best algorithm we have found so far.

3.2 Walking Down Every Possible Branch

The algorithms described in Sections 2.5 and 2.6 were not intended to become the best algorithm possible. Instead, we made these algorithms to see what would happen if instead of taking the first (*Knuth*) code in *guess codes*, we walk down all possible *knuth codes* in *guess codes*. We made two variants, one where we try all codes in *guess codes* when there

Guesses	EPB	EPBOK
1	1	1
2	305	305
3	17,424	10,352
4	2,035,536	166,579
5	9,359,424	380,110
6	128,448	16
Total guesses	55,762,835	2,598,629
Average guesses	4.832	4.662

Table 2: Number of times the Every Possible Branch Algorithm (EPB) and Every Possible Branch Only Knuth Algorithm (EPBOK) needed a certain number of guesses to crack the *Mastermind* code over all possible paths, as well as the total number and average number of guesses to crack all codes with all paths. A path is a sequence of guesses the algorithm takes, until it cracks the *Mastermind* code.

are no *Knuth codes* in *guess codes*, and one where, if this happens, we just take the first code in *guess codes*.
When we look at table 2, there are a few things we need to mention:

- The first guess we do is always fixed (code 1122 in this case). That is the single path that needs one guess to crack the *Mastermind* code.
- We walk approximately 20 times more paths if we always check every code in *guess codes* when it does not contain any *Knuth code*.
- Not only do we sometimes need six guesses to crack the *Mastermind* code, we also need a lot more guesses on average (4.832 or 4.662 in comparison to 4.476 of the Five-Guess Algorithm)
- For EPBOK, only $2.87 * 10^{-3}$% of the total paths need six guesses to crack the *Mastermind* code, while 1.11% of the total paths need six guesses for EPB.

We can conclude, that it is relevant that we always take the first *Knuth code* in *guess codes*, or the first code in *guess codes* if it does not contain any codes that are in *Knuth codes*. This was not mentioned in [Knu77] and [Wik].

3.3 Worst Case Scenario

In Section 2.7 we describe an algorithm that chooses a random *Knuth code* as its next guess. To determine the worst case scenario, we wanted to measure how many paths we would walk, when we are only taking *Knuth codes* as viable guesses, and what the average number

of guesses is that we would need to solve all paths. This, however, would take a long time.

That is why we instead tried to calculate how long it would take, if we would test this. To simulate the worst case scenario, we modified the MiniMax Only Knuth Codes Algorithm (Section 2.3), so that it would take the highest value instead of the lowest value:

```
function MiniMax(knuth_codes){
    FOR each code in knuth_codes:
        FOR each code_to_crack in knuth_codes:
            feedback = GuessCode(code_to_crack, code)
            times_found[feedback]++
        ENDFOR

        maximum = GetMaxValue(times_found)
        scores[code] = maximum
    ENDFOR

    maximum2 = GetMaxValue(scores)
    FOR each code in knuth_codes:
        IF scores[code] == maximum2:
            add code to guess_codes
        ENDIF
    ENDFOR

    return guess_codes
}

function Mastermind(n, m, a/na){
    total_codes = GetTotalCodes(n, m, a/na)
    knuth_codes = total_codes
    possible_codes = total_codes
    mastermind_code = GetMastermindCode(total_codes)

    WHILE mastermind_code has not been cracked:
        print number of codes left in knuth_codes
        code = GetCode(knuth_codes, possible_codes)
        feedback = GuessCode(mastermind_code, code)

        IF feedback == (m cp and 0 wp):
            mastermind_code has been cracked
        ELSE:
            PruneList(code, feedback, knuth_codes)
```

```
            ENDIF
        ENDWHILE

        return number of guesses needed
    }

    # All the other functions that are used, are the same as
    # the ones described in Section 2.1
```

We ran this program a number of times, keeping track of the number of codes left in *Knuth codes*. In the end we found a code path in which we needed nine guesses to crack the *Mastermind* code. We multiplied the number of codes left in *Knuth codes* for every guess we did, which gave us a total of $1296 * 625 * 256 * 81 * 16 * 4 * 3 * 2 * 1 = 6.44972544 * 10^{12}$ paths we would need to walk in the worst case for a single *Mastermind* code. This would mean we would need to walk a total of $1296 * 6.44972544 * 10^{12} = 8.35884417 * 10^{15}$ paths.
It is worth mentioning that the actual number of paths is much lower than this, since almost all paths need less than nine guesses, which also reduces the number of paths.

In theory, we could walk all possible paths with the same methods as described in Section 2.6, and we ran this algorithm for several minutes. We wrote down how many paths we had walked. This gave us a result of 315 paths per second.

With this average, walking all paths would take $1296 * 1296 * 625 * 256 * 81 * 16 * 4 * 3 * 2 * 1/315 = 2.6536013 * 10^{13}$ seconds, or 840875.5 years.

Instead of walking all these paths, we decided to run the Random Knuth Codes Algorithm (Section 2.7) 10.000 times:

Guesses	Random Knuth Codes Algorithm
1	9
2	165
3	853
4	3,353
5	4,220
6	1,320
7	79
8	1
Total guesses	45,891
Average guesses	4.589

Table 3: Number of times Random Knuth Codes Algorithm needed a certain number of guesses to crack a code, as well as the total and average number of guesses to crack 10.000 random codes.

When we look at the results, we see that our average is surprisingly close to that of the Five-Guess Algorithm (4.476) and that it is better than that of Every Possible Branch Algorithm (4.832) and Every Possible Branch Only Knuth Algorithm (4.662). However, in some cases it needs seven or eight guesses to crack a code.

3.4 Other Algorithms

The algorithm made by Koyoma and Lai gives an average guesses of 4.34 [KL93]. They, however, sometimes need six guesses to crack a code. That is why we do not consider this algorithm the best algorithm.
The algorithm made by Berghman et al. [BGL09] uses 4.39 guesses on average. It was, however, not possible for us to reconstruct their algorithm. We therefore could not test how many guesses it would take at most.
If the maximum number of guesses were five, just like for Knuth's Five-Guess Algorithm, then the algorithm by Bergman et al. would be better, because of the lower average.
However, from the algorithms that we could test, Knuth's Five Guess Algorithm is the best, both in terms of maximum number of guesses, and in terms of the average. Therefore, we will use this algorithm for our experiments with the other variants of *Mastermind* in Chapter 4.

4 Best Algorithm Results

When looking at all the results, we came to the conclusion that the Five-Guess Algorithm (Section 2.1) is the best algorithm we explored. We will therefore use the Five-Guess Algorithm to answer the second question of this thesis:
How many guesses would the best algorithm need for Mastermind games of both sca and scna variants, with six up till and including ten different colours?

We start with the six colours results:

Guesses	*sca* 6 colours	*scna* 6 colours
1	1	1
2	6	3
3	62	53
4	533	191
5	694	112
Total guesses	5,801	1,490
Average guesses	4.476	4.139

Table 4: Number of guesses needed for a certain code with the Five-Guess Algorithm for an *sca* and *scna Mastermind* game with six colours and the number of codes that need that number of guesses, as well as the total and average number of guesses to crack all codes.

When we count the *total codes*, we see that, instead of 1296 *total codes* with *sca*, we only have a 360 with *scna*. This reduces the computation time to crack all codes a lot.
Another thing we like to mention is, that the first guess of the *scna* variant is code 1234. This is, because of symmetry: at the start of a *Mastermind* game, all 360 guesses are equally good. That is why *guess codes* contains all 360 codes. The lowest numeric value of these codes is code 1234.
It is also not surprising that the average number of guesses of *scna* is lower than *sca*: if there are less codes to choose from, we need less guesses to crack the *Mastermind* code.

We now consider the *sca* and *scna Mastermind* games with seven and eight colours:

Guesses	*sca* 7 colours	*scna* 7 colours	*sca* 8 colours	*scna* 8 colours
1	1	1	1	1
2	2	2	1	2
3	64	59	56	56
4	538	378	500	463
5	1,512	379	2,169	931
6	28	21	1,369	227
Total guesses	10,077	3,715	21,230	8,042
Average guesses	4.698	4.423	5.183	4.787

Table 5: Number of guesses needed for a certain code with the Five-Guess Algorithm for an *sca* and *scna Mastermind* game with seven and eight colours and the number of codes that need that number of guesses, as well as the total and average number of guesses to crack all codes.

As we can see, when we have seven colours, we cannot crack every code within five guesses. Below are some interesting results we found by testing:

- We need at most six guesses to crack a random code for a *Mastermind* game with seven or eight colours.
- The first code it takes as a guess is 1234 for all four game variants (*sca* and *scna*, with both seven and eight colours).
- When we increase the number of colours, the number of times we guess a code in four guesses or less stays roughly the same, but the number of times we need five or six guesses to crack a code, increases a lot. We can explain this:
  - Every feedback creates a subset of *Knuth codes*.
  - Every guess separates the *Knuth codes* in a number of subsets. This number is more or less independent from the number of colours.
  - For most of these subsets, we use the same first few guesses.
  - After a few guesses, we are more likely to use codes which consist of the added colours as guesses.

  That is why the number of times we guess a code in four guesses or less stays roughly the same.

We finally consider the *sca* and *scna Mastermind* games with nine and ten colours.

Guesses	*sca* 9 colours	*scna* 9 colours	*sca* 10 colours	*scna* 10 colours
1	1	1	1	1
2	1	2	1	3
3	58	52	50	44
4	429	515	380	515
5	2,455	1,661	2,367	2,124
6	3,367	766	5,812	2,151
7	250	27	1,389	202
Total guesses	36,120	15,311	58,103	27,139
Average guesses	5.505	5.063	5.810	5.385

Table 6: Number of guesses needed for a certain code with the Five-Guess Algorithm for an *sca* and *scna Mastermind* game with nine and ten colours and the number of codes that need that number of guesses, as well as the total and average number of guesses to crack all codes.

As we can see, when we have nine colours, we cannot crack every code within six guesses. When we compare nine colours with ten colours, and also look at our previous games with six, seven and eight colours, there are quite some things worth mentioning:

- We need at most seven guesses to crack a random code for a *Mastermind* game with nine or ten colours.
- The first code it takes as a guess is 1234 for all four game variants (*sca* and *scna*, with both nine and ten colours).
- When we have the *sca* variant with eight or more colours, there is only one code that needs two guesses to be cracked. For nine and ten colours this is code 5566. For ten colours we can easily explain this. If the feedback we get to our first guess 1234 is 0 *cp* and 0 *wp*, we are left with 1296 *Knuth codes*. The situation we have now, looks like the beginning of the *sca Mastermind* variant with six colours, with the only difference, that *possible codes* contains 9999 codes, instead of 1296. It then behaves almost the same as the *sca Mastermind* variant with six colours.
- When we have the *scna* variant with ten colours, we suddenly have three codes that need only two guesses to be cracked.

When we were testing the Five-Guess Algorithm, we encountered a few other interesting things.

- When we increase the number of colours, two things happen:

1. Overall, the number of codes that need a low number of guesses gradually decreases.
2. The number of codes that need many guesses increases very fast.

- For an *sca Mastermind* game with six colours and 1296 *total codes*, we need five guesses or less to crack every code. When we have 4096 *total codes*, we need at most six guesses. We thought that there might be a strict relation between the number of total codes and the maximum number of guesses needed, regardless of the type of the game (*sca* or *scna*). This hypothesis, however, turned out to be false. An *scna Mastermind* game with nine colours has 3024 *total codes*, but has 27 codes that need seven guesses to crack the secret code.

- For an *sca Mastermind* game with six colours, the first guess took us from 1296 *Knuth codes* to 256 in the worst case. When we did the next guess, we had 46 *Knuth codes* left. After the third guess, we had six *Knuth codes* left. With the fourth guess, we had one *Knuth code* left. Then we needed our fifth guess to crack the secret code.
  We tried to find a relation between the number of *Knuth codes* left before pruning and the number of *Knuth codes* left after pruning, but we were unable to find it.

- The average number of guesses needed to crack all codes increased when we increased the colours. This is indeed what we expected to happen.

5 Conclusions and Further Research

In this thesis we investigated the following questions:

1. What is the best algorithm to solve a *Mastermind* game with six colours, where it is allowed to use the same colour multiple times in the code?

2. How many guesses would this algorithm need for *Mastermind* games, for the variants that the same colour is allowed or is not allowed multiple times in the code, with six up till and including ten different colours?

We tested several algorithms, and of these algorithms, the best algorithm to solve a *Mastermind* game with six colours, with multiple times the same colour allowed in the code, is the Five-Guess Algorithm that Donald E. Knuth made.

So, using the Five-Guess Algorithm to answer our second question, we saw that it takes at most five guesses for a *Mastermind* game with six colours, six guesses for a *Mastermind* game with seven or eight colours, and seven guesses for a *Mastermind* game with nine or ten colours, regardless of the variant of the game (multiple times the same colour allowed or not).

For future research it would be a good idea to see if the patterns we observed in the experiments, for example the gradually decreasing number of codes that can be solved in four guesses or less, stay the same when we use more colours.
The relationship between the maximum number of guesses, the number of colours and the variant of *Mastermind* is also worth looking into in the future.

We could also make another variant of the Five-Guess Algorithm, which takes a fixed entry out of all best next guesses, instead of the first entry.
Furthermore, we also had an idea for one more algorithm, which takes into account the previous guesses, in order to make an even better guess. In the limited time frame for this thesis project, it was not possible to work this out. It might be interesting to look at this in the future.

A whole different future project would be, to see if the Five-Guess Algorithm would still be optimal if we used codes of length five or more, instead of length four.
Another interesting research would be to see if our Secondary MiniMax Algorithm would still be worse than Knuth's algorithm, when we have seven colours instead of six.

References

[BB86] Azer Bestavros and Ahmed Belal. Mastermind: A game of diagnosis strategies. *Bulletin of the Faculty of Engineering, Alexandria University*, pages 1–10, 1986.

[BGL09] Lotte Berghman, Dries Goossens, and Roel Leus. Efficient solutions for Mastermind using genetic algorithms. *Computers & Operations Research*, 36(6):1880–1885, 2009.

[BHG+96] J. L. Bernier, C. I. Herrt'aiz, J. J. M. Guervt'os, S. Olmeda, and A. Prieto. Solving Mastermind using GAs and simulated annealing: A case of dynamic constraint optimization. *PPSN IV: Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, pages 554–563, 1996.

[KL93] Kenji Koyoma and Tony W. Lai. An optimal Mastermind strategy. *Journal of Recreational Mathematics*, 25(4):251–256, 1993.

[Knu77] Donald E. Knuth. The computer as Master Mind. *Journal of Recreational Mathematics*, 9(1):1–6, 1976–77.

[Koo05] Barteld Kooi. Yet another mastermind strategy. *ICGA Journal*, 28(1):13–20, 2005.

[Mei] Mordecai Meirowitz. https://en.wikipedia.org/wiki/Mordecai_Meirowitz. Retrieved 2019-05-06.

[SZ05] Jeff Stuckman and Guo-Qiang Zhang. Mastermind is NP-Complete. arXiv: cs/0512049, 2005.

[Wik] Wikipedia. Mastermind (board game). https://en.wikipedia.org/wiki/Mastermind_(board_game). Retrieved 2019-05-06.