Utrecht University

Bachelor Thesis
7.5 ec

Genetic Algorithms Playing Mastermind

Vivian van Oijen, 5681235

Supervisor:
dr. ir. Dirk Thierens

Second reviewer:
dr. Johannes Korbmacher

BSc Artificial Intelligence
Utrecht University
Faculty of Humanities
The Netherlands
July 4, 2018

Abstract

This thesis discusses the game Mastermind and a number of strategies that can be implemented to solve this NP-complete problem. First, four different types of algorithms that can be used will be discussed and examples will be given for each one. These algorithms will then be compared with each other. One of these algorithms, the GA presented by Berghman et al. (2009), will be discussed in detail and tested for its scalability, along with a few variations on said algorithm. This algorithm scales relatively well; one of its variations scales even better and is therefore suited for solving Mastermind even for larger problem parameters. In order to obtain a better understanding of the performance of the algorithms tested here, in terms of the number of guesses needed to solve the problem, a follow-up study is required. It is clear however, that in terms of computation time and scalability this GA and its variations perform well compared to many others.

Contents

1 Introduction **2**

1.1 The game Mastermind . . . 2

1.2 Relevance in artificial intelligence . . . 3

1.3 Research question . . . 3

2 Literature review **4**

2.1 Mastermind Satisfiability Problem (MSP) . . . 4

2.2 NP-completeness . . . 4

2.3 Types of algorithms . . . 4

2.3.1 Full enumeration . . . 5

2.3.2 Heuristics . . . 6

2.3.3 Local search . . . 7

2.3.4 Genetic algorithms . . . 8

2.4 Comparison of algorithms . . . 10

2.5 GA by Berghman et al. (2009) . . . 11

2.5.1 The original algorithm . . . 11

3 Experiment **14**

3.1 Tested algorithms . . . 14

3.2 Research questions revisited . . . 15

3.3 Design . . . 15

4 Results **17**

4.1 Original algorithm: fitness proportionate selection . . . 17

4.2 First variation: family competition . . . 19

4.3 Second variation: genetic local search . . . 21

4.4 Multi-start local search . . . 22

4.5 Summary . . . 25

5 Discussion **26**

6 Conclusion **28**

7 References **29**

Chapter 1

Introduction

Mastermind is a game of logic that was invented in 1970 by Mordechai Meirowitz. It is played by two players, one of whom is the codemaker and the other is the codebreaker. The codemaker chooses a secret code, a sequence of colours. The codebreaker's goal is to figure out what that code is, using the feedback given by the codemaker. Mastermind can be seen as a search problem with constraints and is therefore interesting from the algorithmic point of view. Constraint satisfaction problems are typically solved with some sort of search algorithm; this is the case with Mastermind as well. It is possible to investigate the optimal strategy for the game and ways to tweak this strategy in order to improve it in terms of performance. As such, many algorithms have been presented, tested and compared in the past.

First, Mastermind will be discussed in more detail. In the next chapter (2), several different types of algorithms will be reviewed and examples of existing algorithms will be presented for each type. After that, the algorithms will be compared to each other and finally, one interesting algorithm, and a few variants of it, will be further examined by performing an experiment.

1.1 The game Mastermind

The secret code is a sequence that consists of P coloured pegs. There exist N possible colours. There are many variants of the game, but the most common and well known one is Classic Mastermind (often simply called Mastermind), where $P = 4$, $N = 6$ and repetition is allowed, meaning that one colour may occur more than once. When the game is implemented as a computer program, these colours can easily be represented by digits. So for $N = 6$ the set of possible 'colours' would be $\{1, 2, 3, 4, 5, 6\}$.

In order to figure out what the code is, the codebreaker makes guesses to which the codemaker responds. The number of guesses the codebreaker is allowed to make varies. It may even be infinite, though there is only a finite number of possible codes: N^P. The response consists of black and white pegs; the number of black pegs equals the number of pegs in the guess that are correct in both their colour and their position and the number of white pegs equals the number of pegs in the guess that are correct in colour, but not in position.

1.2 Relevance in artificial intelligence

As stated earlier, Mastermind is a search problem with constraints. Search algorithms are a relevant subject in the field of AI, and so are agents that are able to examine possible solutions to a problem in a smart way. Since Mastermind is NP-complete (Section 2.2), deterministic algorithms that will evaluate each possibility do not scale well for this problem. Other types of algorithms that work with a limited amount of information are better suited. In particular genetic algorithms, which will be examined in the most detail here, are interesting for AI.

1.3 Research question

For this experiment, it will be evaluated how well certain algorithms scale with regard to problem parameters. An algorithm might perform well for the classic version of Mastermind, but due to the nature of the problem, that does not mean it is equally suited for a more complex version. Therefore it is important to consider not only performance in terms of the number of guesses necessary to solve the problem, but also time-complexity when assessing an algorithm. Four different algorithms will be tested (Chapter 3) and several more will be discussed in the literature review (Chapter 2). After that, the research question will be revisited and discussed in more detail (Section 3.2).

Chapter 2

Literature review

2.1 Mastermind Satisfiability Problem (MSP)

The game of Mastermind can be approached as a decision problem, a question with a yes-or-no answer depending on input variables. In this case that would be: ”Given previous guesses and their corresponding responses, is there at least one possible code that is valid?” A valid or eligible code is a possible code that, if it were the secret code, would generate exactly those responses that have been observed so far.

Stuckman, J., & Zhang, G.-Q. (2005) refer to this decision problem as the Mastermind Satisfiability Problem (MSP). By approaching the problem as such, Stuckman and Zhang were able to prove it is NP-complete.

2.2 NP-completeness

When a decision problem is NP-complete, that means that any solution can be verified in polynomial time; however, there is no known way to efficiently find a solution. That is, there is no way to deterministically find the correct solution in polynomial time. Mastermind is a combinatorial problem and as the problem parameters grow, the time needed to solve the problem deterministically increases exponentially. There are no efficient algorithms that are guaranteed to find a solution in few guesses, but verifying a given solution can be done by simply checking wether all its pegs match the pegs of the secret code.

Solving Mastermind in as few steps as possible is time-intensive, since all possible codes need to be evaluated. In order to improve the computation time needed to find the secret code, many techniques have been tested. It proves to be quite challenging though, if possible at all, to construct a strategy that is both optimal and efficient.

2.3 Types of algorithms

Although Mastermind is NP-complete and there is no known way to efficiently and deterministically find the solution (Section 2.2), there are several types of algorithms that can be used to solve the game. When choosing a strategy, one must make a trade-off between optimum and performance.

An optimal strategy is one that guarantees that the solution will be found in as few guesses as possible. So, for each guess that means all possible codes need to be evaluated and the best one will be played. Unfortunelately, applying this strategy requires a lot of computation time. On the other hand, it is possible to create an algorithm that can find the secret code more quickly, but it will not be strategically optimal and therefore it will not be guaranteed to find a solution within a certain number of guesses. It might occur for the algorithm to only find a solution after many guesses.

When evaluating the performance of an algorithm, the average number of guesses needed to find the solution is typically looked at. As the exact computation time depends on other factors as well, like the computer and the implementation, one might also look at the average number of combinations that are evaluated by the algorithm, and what "evaluating a combination" entails for that algorithm. For some algorithms that entails computing exactly how many guesses would still be eligible after receiving each possible response, but another algorithm might evaluate a code by calculating a fitness value with a simpler function.

Below, several different types of algorithms that can be used to solve a search problem like Mastermind will be discussed, and examples will be given. In the section after that (2.4), the algorithms will be compared.

2.3.1 Full enumeration

These are the algorithms that examine every possible or eligible code and compute which one has the best score according to some function before choosing the next guess. By doing so, full enumeration algorithms are able to find the secret code in few guesses each time, however they are inefficient. Not only that, the time required to solve the problem increases quickly as P and N grow.

Knuth (1976) presented an algorithm that could solve the game in five moves or less; it finds a solution in 4.478 moves on average. In order to do so, it uses the technique minimax, which minimises the loss that will occur in a worst case scenerio, i.e., it minimises the maximum number of guesses needed after the next guess. This is the worst case strategy. This algorithm is guaranteed to find the solution in few moves, however it is time-intensive. For every guess, it examines all possible unused codes, not only the eligible ones, which makes it inefficient. Especially for a variant of the problem with a longer sequence and more possible colours to choose from, finding the solution using this algorithm will be problematic. Still, Knuth's algorithm is a very influential one and others have used it as a base for their own strategies.

Irving (1979) is one of the people who have come up with an algortihm similar to that of Knuth. Whereas Knuth uses a worst case strategy however, Irving uses an expected size strategy. He does this by playing the guess that minimises the expected number of remaining codes, rather than the maximum number of remaining codes. By doing so, Irving reduces the average number of guesses slightly with respect to Knuth, Irving needs 4.369 guesses on average. However, in the worst case scenario his algorithm needs six moves, as opposed to Knuth's five.

Another example is Norvig and Berkeley's (1984) algorithm, which they presented in 1984 and described as "different from the others in that it is specifically designed to be an approximation to the optimal strategy". The strategy they use is to play the guess that minimises the maximum number of remaining possible target sequences. Which is similar to Knuth's, but with the added heuristic that the number of guesses needed to find the secret code is directly proportional to the number of eligible guesses left. Norvig and Berkeley's algorithm looks at all eligible codes only. This makes it less time-intensive than Knuth's algorithm. It needs 4.47 guesses on average.

Kooi (2005) uses a strategy that almost seems to be the opposite of what many others do. He proposes an algorithm that solves the problem by applying the most parts strategy. According to this strategy, one should make the guess that leads to the highest number of possible answers. This is based on the following principle: A guess may be considered a partition, as it splits the set of codes that are eligible at the time of choosing a new guess into smaller sets of codes. One of those subsets will be the set of eligible guesses after the guess has been played and a new response has been given, which subset depends on what the response is. Kooi reasons that in order to maximise the number of combinations for which one would win in a certain round, one should maximise the number of parts into which the set of all combinations is partitioned in each round, i.e., the more possible answers one might receive after playing a certain guess, the more parts the set of possible codes can be split into. By doing so, there will be more possible codes for which one would win in a certain round. Kooi needs 4.373 guesses on average.

2.3.2 Heuristics

This is a technique that can be used in order to solve a problem more quickly than one would be able to when using full enumeration. Because of that, heuristics are often used for problems that are NP-complete, as is the case here. These algorithms achieve their speed by trading optimality, completeness or accuracy. A heuristic function is used in order to rank possible solutions based on available information in order to estimate how good a possible solution is.

Perhaps the simplest example is the algorithm proposed by Shapiro (1983). It initialises the set of all possible codes and then chooses a random order in which to examine those codes. As soon as a code is found which is still consistent with previously obtained information, it is played. The algorithm does not consider every possibility before making a decision, its heuristic is to simply play the first eligible code it finds. After that the algorithm picks up where it left off and continues to iterate over the set of codes. Shapiro (1983) claims his algorithm performs quite well compared with experienced players, as it needs 4 to 6 guesses on average. This makes it slightly worse than Knuth's algorithm, up to 5 guesses, but it does not examine as many codes, making it faster. This algorithm uses greedy search, since it plays an eligible code as soon as it can find one. Shapiro's algorithm needs an average of 4.758 guesses.

Singley (2005) has developed several algorithms, one of which also uses greedy search. This is a good example of a simple heuristic algorithm. It follows the heuristic of playing the guess which is logically optimal at each step, as Singley states. Singley starts with perhaps the simplest strategy possible: determine the colour of each peg separately, one at a time. By doing so, the algorithm does not examine or compare any codes at all, making it fast in terms of computation time. But in the worst case scenario, the number of guesses needed to find the solution is $P * N$, which is 24 for the classic variant. This makes it far from optimal. In order to improve this algorithm, Singley makes a few observations. First, only complete guesses may be submitted. This means that a colour does not have to be explicitly tested for each peg. He further improves the algorithm after observing a strategy often used by human players: first determine what the colours of the secret code are, then find their permutation. Instead of playing guesses consisting of a single colour (1111, 2222, etc.) even when those are known to be wrong, previously determined colours are kept in the guess. After finding the colours, the algorithm iterates through that set in order to find the correct permutation of those colours. Singley's algorithm can solve the problem in $N + \frac{P(P+1)}{2} - 2$ guesses or less, for Classic Mastermind with $P = 4$ and $N = 6$ that equals 14 guesses or less. This is not a particularly good score; however, keep in mind that this is achieved without examining any codes before playing a guess.

2.3.3 Local search

A subtype of heuristic algorithms is local search. These algorithms use sampling and neighbourhoods in order to examine only part of the total search space. They typically initialise a solution randomly, or use some kind of heuristic to find one, and then look at nearby solutions. These nearby solutions are called the neighbourhood. This way, only a predefined number of possible solutions need to be compared to each other at each step of te algorithm. In the case of Mastermind, this generally means that at the start, a fixed or random guess will be played. After that, this current solution is compared with its neighbours. This goes on until the correct code is found. By doing so, these algorithms are not guaranteed to find the solution in few guesses, but they are significantly faster than full enumeration algorithms in terms of computation time. This means they scale better and become increasingly more useful as the size of the problem grows. Examples of local search algorithms include tabu search, simulated annealing and hill climbing.

Aside from the greedy search algorithm discussed in the previous section (2.3.2), Singley also developed a local search algorithm (Singley, 2005). Again, he considers a simple algorithm first and then discusses how it can be improved. He starts with a 2-swap algorithm, which creates an initial solution by determining the colours that occur in the secret code using the first few guesses, similar to his other algorithm. After that, the pegs of different colours are swapped in a methodical order and the resulting guess is submitted. When a better solution is found, it is kept. Singley develops a tabu search approach for solving the problem by choosing pairs at random rather than using a methodical order. The positions of the swapped pegs are added to the tabu list, which is used to determine whether a subsequent chosen pair will actually be swapped. Unlike the initial algorithm, this one does not submit a guess after each swap. Instead, it searches until it finds an eligible guess and submits that guess.

Bernier et al. (1996) proposed an algorithm using another type of local search, simulated annealing. Simulated annealing starts with an initial solution and then selects a random alternative solution from the neighbourhood at each step. If this solution is better than the previous one, which is calculated according to some function, it is kept. If not, there is a certain chance it will be kept anyway. The chance of this happening will decrease over time, this is controlled by a parameter called temperature. Bernier uses this strategy in order to find a code that meets the maximum number of constraints within a limited time. A cost function that counts the differences in black and white pegs between the response of a guess and the target response (P black pegs), is used to compare different codes with each other. Bernier uses two operations to find a code's neighbours, permutation and mutation. Bernier et al. tested this algorithm for $P = 6$ and $N = 6$, instead of $P = 4$, and with these parameters, the algorithm needs an average of 5.45 guesses.

Temporel and Kovacs (2003) proposed yet another local search algorithm, derived from the algorithm presented by Baum et al. (1995), who presented Random Mutation Hill Climbing, which Temporel and Kovacs (2003) adapted to Mastermind. This algorithm uses a hill climbing strategy and it also uses a heuristic. It keeps track of the "current favourite guess", or CFG, and starts by playing a random initial guess, which will be the first CFG. Using the CFG and its corresponding response, a new code is derived stochasticly. If this new code is eligible, it is played. Otherwise this step is repeated to derive another code. As a heuristic, this algorithm uses the distance between each possible response and the goal, four black pegs, to establish how good a guess is. If a submitted guess is as good as or better than the CFG according to this heuristic, it becomes the new CFG. After this step, the algorithm derives a new guess again and keeps following this same process until it finds the secret code. This algorithm needs 4.621 guesses on average, but is interesting mostly because it needs to evaluate an average of only 85 codes, which is less than 7% of all possible guesses to achieve this feat. Temporel and Kovacs discuss two possible alterations two this algorithm: a code tracker and Fitness Proportionate Selection (FPS) of pegs to mutate. A code tracker is an array that keeps track of how many times each possible code has been evaluated. This can be used to lower the number of evaluations needed in total, since codes that have already been examined before are now avoided. FPS selection means that a fitness function is used to select pegs that will be mutated instead of choosing pegs randomly. When both of these techniques are used, the algorithm needs 4.625 guesses on averages and it needs to examine only 41.9 codes on average, which is less than 4% of all possible codes.

2.3.4 Genetic algorithms

A genetic algorithm, or GA, is a subtype of local search, as it also uses sampling to examine only part of the search space. It mimicks natural selection; the idea behind this technique is to start with a random population of solutions and improve it over time by creating subsequent generations by selecting individuals from the current population and using them as parents for crossover. Crossover combines the information from the parent solutions to stochasticly create new solutions.

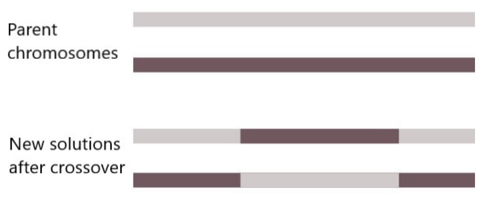


Figure 2.1: Illustration of crossover

In order to apply this strategy, the possible solutions to a problem must be represented as what we call a chromosome, e.g. a binary string or a string of characters, some kind of sequence. In the case of Mastermind this is quite easy, we can simply represent solutions as a string of digits like we have been doing so far. When a GA is used to solve Mastermind, it typically starts with playing a fixed initial guess and receiving its corresponding response. As long as this response does not consist of four black pegs, a new set of eligible guesses will be determined by first creating a population of random solutions and then, using the responses found so far, improving that population and adding eligible guesses from each generation of the population to a seperate set. From this set of eligible guesses, the next guess that will be played is then chosen randomly or according to some function.

Bento et al. (1999) presented one of the first GAs for Mastermind. It starts by initialising a population of a fixed size. All the codes in this first population are random and without repetition in colours. The first guess is chosen randomly from this population. For all other guesses the current generation's element with the best fitness is chosen. The fitness function uses previous responses to evaluate whether a code satisfies all the constraints found so far. The selection of parent codes used to produce the next generation is proportional to the fitness computed with the same fitness function. A new generation is created with crossover and it replaces the previous one, but a predefined number of the best codes from the parent generation will substitute an equal number of the worst codes from the new generation. This is called elitism. A random code from the resulting population is selected for mutation. This mutation happens by changing the colour of a randomly chosen peg. A heuristic is also added to this algorithm: when there are neither white nor black pegs in the response to a certain code, all the colours in that code are removed from the population and will no longer be applied after mutation. This algorithm needs 7.538 guesses on average.

Bernier et al. (1996) proposed another GA, along with their local search algorithm. They noted that, since a new constraint is added with every guess played, the problem is to maintain diversity as that way the changing fitness landscape can be explored efficiently. Because of this and the fact that only one possible code is correct, Bernier et al. (1996) present an algorithm that plays a guess as soon as a suitalbe one is found. A population is initialised, the size of which depends on the length of the codes. A proportion of each generation is kept in the next one, the generation is not replaced entirely. Three different methods are used to generate new codes. The first is by applying crossover to two parent codes. In order to increase diversity, there is a chance of random mutation occuring. The second way to create new codes is by cloning with mutation. Here, only one parent code is used and a new code is made by mutating it. The third method also uses only a single parent, but this time inversion is used. This operator chooses two positions of pegs from the parent code and reverses the pegs between those two positions. The last two methods are only used on codes whose fitness is not among the best. As mentioned before, as soon as a code is found that is eligible, it is played. The response it receives is then used to update the constraints. Bernier et al. tested this algorithm for $P = 6$ and $N = 6$, instead of $P = 4$, and with these parameters, the algorithm needs an average of 5.62 guesses.

A more recent GA for Mastermind was presented by Berghman et al. (2009). They claim their algorithm requires low running times and needs a low number of guesses on average. The idea behind is to use several generations of a population to create a large set of eligible guesses, one of which is then chosen as the guess that will be played next. The first guess is fixed. For each guess after that, a set of eligible guesses is found by initialising a random population and following the pattern of a GA with crossover, mutation, permutation and inversion. For each code, the probability of being chosen as parent is proportional to its fitness, which is calculated by determining the response the code would result in, if the previous guesses were the secret code. Essentially, a GA is run for every individual guess, rather than running it a single time and selecting a code from each generation. A maximum number of generations and a maximum size for the set of eligible codes are defined and when either of those criteria is met, the algorithm terminates and one of the codes in the resulting set is chosen as the next guess. This algorithm finds the solution in 4.47 guesses on average. It also seems to be fast in terms of computation time, as Berghman et al. (2009) report an average of 0.762 seconds, when random selection is being used to choose a code from the set of eligible guesses.

2.4 Comparison of algorithms

Now that different types of algorithms have been reviewed, and several examples for each type, it is time to discuss and compare the results of them all. In Table 2.1 an overview of the performance of each algorithm is shown.

Algorithm by	Type	Guesses	Computation time
Irving	full enum.	4.369	-
Kooi	full enum.	4.373	-
Berghman et al.	GA	4.47	0.762 sec
Norvig & Berkeley	full enum.	4.47	-
Knuth	full enum.	4.478	-
Temporel & Kovacs	local search	4.625	$\sim$ 1 sec / 41.2 codes evaluated
Shapiro	heuristic	4.758	1.087 sec[1]
Singley (Tabu)	local search	5.736[1]	-
Bento et al.	GA	7.538	$\sim$ 1.5% - 18% codes evaluated[2]
Singley (Greedy)	heuristic	$\sim$ 9.5	-
Bernier et al. (SA)	local search	5.45	tens of ms to 10 sec
Bernier et al. (GA)	GA	5.62	seconds to hours

[1] Copied from the work of Berghman et al. (2009) as it was not reported in the work where the algorithm was introduced.
[2] Average depends on population size used by the algorithm.

Table 2.1: Average number of guesses needed and any reported information about the average computation time of each algorithm, for $P = 4$ and $N = 6$. Bernier et al. (1996) report their results for $P = 6$ instead.

Note that while Singley's (2005) algorithm using greedy search needs more guesses on average, he reported that tabu search ran for a significantly longer time than greedy search and that it does not scale well at all. Similarly, full enumeration algorithms perform relatively well, but as discussed in Section 2.3.1, they are significantly less efficient than the other algorithms and scale badly.

Table 2.1 makes it clear that, in general, full enumeration algorithms need the lowest number of guesses. With the GA by Berghman et al. (2009) being the only exception, the full enumeration algorithms achieve the best results in terms of guesses needed. GAs do not seem to perform the best in general. The GA by Bernier et al. (1996) also performs slightly worse than their local search algorithm, even though they report that during their experiment their GA takes significantly longer to run.

However, Table 2.1 only considers the results for small problem parameters. As those parameters grow, the computation time becomes more important and other algorithm types than full enumeration will be preferred. For the classic variant of the game there are $6^4 = 1296$ possible codes that will be examined, but for another variant of the game with $P = 6$ and $N = 9$ there are $9^6 = 531441$ possible codes, over 410 times as many. Since Mastermind is a combinatorial problem with a search space that grows exponentially with respect to it parameters P and N, the computation time of a full enumeration algorithm grows exponentially as well. This is why other types of algorithms that trade accuracy for speed become increasingly more useful as the problem grows and why the seemingly less successful algorithms in Table 2.1 are useful after all.

Singley's (2005) algorithms are a good example of the fact that an algorithm is not necessarily better when it needs fewer guesses. As mentioned above, his greedy search does not perform as well as his tabu search; however, running the entire tabu search algorithm for $P = 7$ and $N = 8$ was inferred to take about 117 days, while greedy search took less than one second for all testcases.

In short, for a small search space full enumeration algorithms may be the best option as they tend to perform best and their time-complexity is not a problem. For larger values of P and N other algorithms are more useful.

2.5 GA by Berghman et al. (2009)

A variety of algorithms has been discussed and compared over the previous sections (2.3 and 2.4). Now, one of those algorithms will be studied more closely, namely the GA by Berghman et al. (2009).

The algorithm performs relatively well for an algorithm that does not use full enumeration. Because it does not use full enumeration, it is also relatively efficient. It is for these two reasons this algorithm will be further explored. Throughout the following sections the algorithm will be discussed in detail (2.5.1) and it has also been tested (Chapters 3 and 4). To sum it up, it will be investigated how well this algorithm scales and therefore how useful it is when larger problem parameters are used.

Aside from that, two variations on this algorithm and one different algorithm will be tested as well. The GA will be modified so that the next generation of a population will be created in a different way; the search space the GA explores will effectively change.

2.5.1 The original algorithm

First, an in-depth description of the algorithm as presented by Berghman et al. (2009). As stated earlier, this algorithm chooses each next guess after finding a of set eligible codes. It finds this set by applying a GA. The first guess is fixed, but it is dependent on the problem parameters of course. The initial guess g_1 for different values of P and N is stated below. The initial guesses presented in the work of Berghman et al. (2009) were used:

$$\begin{aligned}
P = 4,\ N = 6 &: \ g_1 = \{1,1,2,3\} \\
P = 6,\ N = 9 &: \ g_1 = \{1,1,2,2,3,4\} \\
P = 8,\ N = 12 &: g_1 = \{1,1,2,2,3,3,4,5\}
\end{aligned}$$

Every guess after the first one will then be generated by the actual algorithm. As long the number of black pegs in the response corresponding to the last played guess is not equal to P, meaning the correct solution was not found, a next guess will be found and played. Each guess and the response it receives are added to the constraints the solution needs to satisfy and will be used to select the next guess.

In order to choose the next guess, a GA is used. It initialises a set of eligible code which starts off empty and a random population of predefined size, though duplicates are not allowed. Subsequent generations are created from parent codes by means of crossover, mutation, permutation and inversion. The probability of a code being chosen as a parent is proportionate to its fitness.

Either one-point or two-point crossover is used, both with a probability of 0.5. When mutation is applied, one peg of the code is randomly selected and is changed to a different colour, also chosen randomly. For permutation, two different positions are randomly chosen and the colour of their corresponding pegs are swapped. For inversion, two different positions are also randomly chosen, but this time the sequence between them is reversed. Each of these operations happens with a certain chance, except crossover. Crossover is always applied.

After each generation is created, every eligible code in that generation is added to the set of eligible codes. This continues until either the set reaches its maximum size or the population has existed for the maximum number of generations, both of these values are chosen at the start of the program. If at this point the set of eligible codes is still empty, a new population is initialised and the process starts over. From the set of eligible guesses that is found this way, one code is chosen to be the next guess. This is done randomly, by selecting the code that is the most similar to the other codes in the set or by selecting the one that is the least similar to the rest. These methods of selecting a code as the next guess were chosen, because they are less time-intensive than calculating which code will lead to the smallest number of eligible codes left after playing the guess. For the experiment described in the next chapter (3), the code was selected randomly.

Berghman et al. (2009) use the following values for parameters:

Population size	= 150
Maximum number of generations	= 100
Maximum size of the eligible set	= 60
Chance of mutation	= 0.03
Chance of permutation	= 0.03
Chance of inversion	= 0.02

And the fitness function they use to calculate the fitness of code c as guess i:

$$fitness(c,i) = a(\sum_{q=1}^{i} |X'_q - X_q|) + \sum_{q=1}^{i}(|Y'_q - Y_q| + bP(i-1)$$

Here, X_q and Y_q are the number of black and white pegs respectively that correspond to previously played guess q. X'_q and Y'_q are the number of black and white pegs respectively that code c would receive if guess q were the secret code. The sum of the differences between the number of pegs is computed; the difference between the number of black pegs is being multiplied by parameter a, which Berghman et al. (2009) set to 1, since they only play eligible codes. If the algorithm is modified such that it may play ineligible guesses as well, choosing a larger value may result in better scores. Finally a constant is added to smoothe the fitness values: a parameter b, Berghman et al. (2009) use value 2, multiplied by P, multiplied by the number of previous guesses minus 1. In this experiment, the values 1 and 2 for a and b respectively were used as well.

Chapter 3

Experiment

The algorithm that was described in the previous section (2.5) has been tested, together with two variants and a different algorithm. In this chapter, the experiment that was performed will be described. First, the algorithms that were tested are discussed.

3.1 Tested algorithms

The first algorithm that will be tested is the original GA by Berghman et al. (2009) (Section 2.5), using fitness proportionate selection. The second is a variation on it that differs in the way a new generation of a population is found, as it uses family competition instead. The algorithm iterates over an unsorted population to divide it into pairs. Every pair will then be used as two parent codes in order to generate two new codes, using crossover, mutation, permutation and inversion. All four codes, the parents and the children, are ranked by their fitness and the best two are added to the new population, regardless of whether the same code already occurs in that population. This is illustrated in Figure 3.1. The rest of the algorithm remains the same.

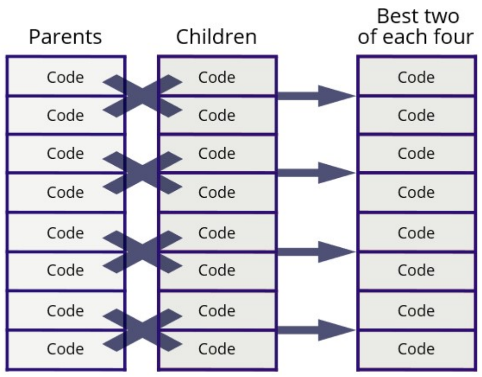


Figure 3.1: Illustration of family competition

The third algorithm being tested is another variation, this one uses genetic local search with family competition. It works similarly to the first variation, the difference being that this one applies local search as well to make sure every code in a population is a local optimum. The algorithm iterates over each code that is generated and for each peg tries out each possible colour. If a different colour leads to a better fitness, the change is kept. The algorithm keeps iterating until no more improvements can be found. This method makes mutation, permutation and inversion unnecessary; they are no longer used, only crossover. After local search has been applied to the children codes, they and the parent codes are ranked by their fitness and the best two are added to the new population, regardless of whether the same code already occurs in that population. Since local search is used, the search space of the genetic algorithm consists of only local optima. This way the algorithm is able to search for the correct solution in a much more effective way and it works with a smaller population size, which compensates for the extra computation time that goes into applying local search so many times. Another effect of exploring local optima is that more eligible codes are found. This results in the maximum size of the eligible set being reached quickly. For this reason, this parameter and the parameter indicating the maximum number of generations are no longer used. Instead, a new stop criterium is being used. Namely, for the algorithm to quit after five generations without improvement in terms of both highest and lowest fitness occuring in the population.

The fourth and final algorithm being tested uses multi-start local search. This is not a GA; it is tested in order to determine how much the GA in the genetic local search algorithm actually contributes to the results the algorithm produces. Since there is no GA involved, the population size simply indicates how many codes are generated each time the algorithm searches for a new guess. All of these codes are initially random. Local search will be applied to each code and then the unique eligible codes will be added to a set of eligible guesses like usual. Since no stop criterium is necessary, the size of the eligible set is not limited, like the genetic local search algorithm. A random code from the resulting set is chosen as the next guess.

3.2 Research questions revisited

The main research question is: how well does the GA presented by Berghman et al. (2009) scale with respect to problem parameters? This question has been divided into the following sub-questions:

- How well does the original algorithm scale?
- How well does the algorithm perform and scale when family competition is used instead of fitness proportionate selection?
- How well does the algorithm perform and scale when genetic local search with family competition is used?
- Does the GA in the genetic local search algorithm contribute significantly to the performance of the algorithm, i.e., how does genetic local search compare to multi-start local search?
- What is the effect of the values of the population size and the maximum number of generations on the performance and on the scalability of each of the algorithms described above (Section 3.1)?

3.3 Design

The experiment was conducted by implementing each of the algorithms described above in C#. While solving the problem, the program kept track of several values used to measure performance, namely the number of guesses needed, the number of guesses that were evaluated in total (this includes the fixed initial guess and every guess that was at one point generated as part of a population), the number of eligible guesses that were found in total (this includes the fixed initial guess and each guess that was added to a set of eligible codes) and the computation time in milliseconds.

Each of the four algorithms was tested for $P = 4$ and $N = 6$, $P = 6$ and $N = 9$, and $P = 8$ and $N = 12$. And for each of these parameters, they were tested for different values of the population size and, if applicable, the maximum number of generations. Each algorithm was run 200 times for each set of parameters.

There was one exception, namely the GA using family competition for $P = 8$ and $N = 12$. This was run only for the largest tested values of population size and the maximum number of generations, and only 30 times. This was due to the long computation time and to the fact that for the smaller values of the population size and maximum number of generations, the algorithm tended to end up in a seemingly infinite loop, as it was unable to find any eligible guesses and therefore the GA did not terminate after a reasonable amount of time.

Chapter 4

Results

While the original algorithm proposed by Berghman et al. (2009) is one of the faster algorithms that was discussed during the literature review (Chapter 2), it is not the algorithm that scales the best out of the ones tested here. The algorithm that scales the best is genetic local search. The results will be discussed in more detail throughout the rest of this chapter.

4.1 Original algorithm: fitness proportionate selection

population size, max. generations	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
75, 50	4.32 ± 1.06	4758.07 ± 5191.39	212.61 ± 65.69	51.00 ± 82.70
150, 100	4.56 ± 0.85	10601.50 ± 14061.34	247.21 ± 64.99	131.02 ± 187.05
300, 200	4.67 ± 0.95	23366.50 ± 46017.30	295.40 ± 121.12	367.04 ± 832.60

Table 4.1: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for fitness proportionate selection with $P = 4$ and $N = 6$. Standard deviations are stated as well.

population size, max. generations	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
75, 50	6.61 ± 1.26	52564.90 ± 58019.50	282.11 ± 77.04	1168.23 ± 1302.27
150, 100	6.55 ± 1.22	85633.75 ± 94297.20	327.87 ± 86.84	2229.54 ± 2701.58
300, 200	6.68 ± 1.09	187661.50 ± 241455.10	361.13 ± 80.27	5278.69 ± 6888.17

Table 4.2: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for fitness proportionate selection with $P = 6$ and $N = 9$. Standard deviations are stated as well.

population size, max. generations	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
75, 50	8.58 ± 1.54	1139040.76 ± 1396362.00	267.50 ± 77.38	51433.37 ± 67502.94
150, 100	8.81 ± 1.46	855194.50 ± 1457472.00	395.44 ± 91.98	45943.87 ± 87081.66
300, 200	8.73 ± 1.55	1420093.00 ± 2058111.00	438.10 ± 99.30	80602.03 ± 118075.80

Table 4.3: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for fitness proportionate selection with $P = 8$ and $N = 12$. Standard deviations are stated as well.

In Tables 4.1, 4.2 and 4.3 it can be seen that while larger values for the population size and the maximum number of generations do not necessarily result in fewer guesses needed to solve the problem, they do seem to significantly slow down the algorithm as more guesses are being evaluated. This difference in computation time is relatively larger for smaller values of P and N.

Large values for the population size and the maximum number of generations do not seem to be needed for this algorithm to perform well. For smaller values of these parameters this algorithm is able to solve Mastermind for a larger P and N in a noticeably shorter amount of time than for larger values. For $P = 4$ and $N = 6$ the algorithm takes 51 milliseconds to run on average and evaluates an average of 4758 codes. For $P = 8$ and $N = 12$ it takes 51433 milliseconds and evaluates 113904 codes; the search space is 331776 as large and it takes approximately 1008 times as long to compute, meaning the algorithm scales much better than a full enumeration algorithm. However, the algorithm does scale relatively badly compared to the other tested algorithms with respect to problem parameters as can be seen in Figure 4.1. As the problem keeps growing, the computation time will grow increasingly quickly as well.

4.2 First variation: family competition

population size, max. generations	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
75, 50	4.58 ± 0.99	16596.76 ± 8442.16	72.27 ± 21.44	132.63 ± 97.65
150, 100	4.54 ± 0.89	40358.50 ± 14763.53	127.46 ± 28.46	311.72 ± 143.93
300, 200	4.54 ± 1.00	110351.50 ± 53540.42	194.66 ± 50.09	948.36 ± 559.47

Table 4.4: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for family competition with $P = 4$ and $N = 6$. Standard deviations are stated as well.

population size, max. generations	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
75, 50	6.47 ± 1.34	772473.54 ± 806578.60	67.15 ± 21.34	15994.99 ± 17615.53
150, 100	6.42 ± 1.38	543213.25 ± 584665.80	112.10 ± 30.09	10828.54 ± 12890.66
300, 200	6.75 ± 1.02	478292.50 ± 288903.90	178.42 ± 39.66	9593.60 ± 6848.10

Table 4.5: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for family competition with $P = 6$ and $N = 9$. Standard deviations are stated as well.

population size, max. generations	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
75, 50	- ± -	- ± -	- ± -	- ± -
150, 100	- ± -	- ± -	- ± -	- ± -
300, 200	8.43 ± 1.63	35377550.00 ± 36197770.00	170.13 ± 41.97	1389076.00 ± 1532076.00

Table 4.6: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for family competition with $P = 8$ and $N = 12$. Standard deviations are stated as well.

In Tables 4.4, 4.5 and 4.6 it can be seen that, again, larger values for the population size and the maximum number of generations do not necessarily result in fewer guesses needed to solve the problem. For this algorithm however, larger values for those parameters actually seem to lower the computation time for larger problem parameters. This might be the case because for a larger search space, it is difficult for the algorithm to locate the solution when it only gets to investigate a small part of the search space. Family competition makes for less diversity than fitness proportionate selection. Therefore, larger values for the population size and the maximum number of generations seem to result in better performance in this case. Still, as can be seen in Figure 4.1 the time the algorithm needs to run, seems to increase much more quickly as the problem grows than it does for fitness proportionate selection (Section 4.1).

4.3 Second variation: genetic local search

population size	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
15	4.48 ± 1.05	460.04 ± 156.75	43.18 ± 13.34	41.75 ± 22.36
30	4.72 ± 0.94	1044.61 ± 283.76	91.23 ± 25.00	93.22 ± 48.71
60	4.58 ± 1.00	1980.72 ± 549.54	173.02 ± 49.28	169.95 ± 84.46

Table 4.7: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for genetic local search with $P = 4$ and $N = 6$. Standard deviations are stated as well.

population size	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
15	6.46 ± 1.62	907.41 ± 334.59	56.15 ± 15.83	530.45 ± 296.34
30	6.74 ± 1.11	1923.94 ± 472.48	117.66 ± 24.86	1073.41 ± 452.05
60	6.67 ± 1.14	3741.67 ± 817.86	231.98 ± 50.87	1997.12 ± 742.72

Table 4.8: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for genetic local search with $P = 6$ and $N = 9$. Standard deviations are stated as well.

population size	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
15	8.40 ± 2.05	1903.58 ± 1315.38	66.77 ± 17.52	5048.98 ± 5060.79
30	8.86 ± 1.61	3446.92 ± 1550.41	136.23 ± 27.39	8543.44 ± 6514.91
60	8.84 ± 1.45	10553.56 ± 60683.98	270.77 ± 47.78	30761.02 ± 237728.50

Table 4.9: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for genetic local search with $P = 8$ and $N = 12$. Standard deviations are stated as well.

In Tables 4.7, 4.8 and 4.9 it can be seen that, once again, a larger value for the population size does not necessarily result in fewer guesses needed to solve the problem.

It can also be seen that unlike the previous variant with family competition (Section 4.2), this algorithm scales better for a smaller population size. This is due to the fact that, while the sample of the search space that the algorithm explores is again very small, the algorithm only considers local optima.

As can be seen in Figure 4.1, this algorithm scales better than both previous algorithms (sections 4.1 and 4.2).

4.4 Multi-start local search

population size	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
150	4.64 ± 0.97	1056.27 ± 293.14	335.58 ± 99.77	3234.70 ± 26008.19
300	4.68 ± 0.82	2126.38 ± 482.06	672.43 ± 194.02	218.12 ± 96.60
600	4.62 ± 0.89	4190.16 ± 1050.34	1324.62 ± 381.59	457.02 ± 207.65

Table 4.10: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for multi-start local search with $P = 4$ and $N = 6$. Standard deviations are stated as well.

population size	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
150	6.74 ± 1.03	1715.19 ± 322.87	378.52 ± 92.71	1135.60 ± 409.75
300	6.82 ± 1.08	3443.87 ± 650.42	775.45 ± 195.93	2296.07 ± 903.76
600	6.87 ± 0.94	7147.87 ± 2803.28	1551.51 ± 378.65	4577.34 ± 2660.66

Table 4.11: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for multi-start local search with $P = 6$ and $N = 9$. Standard deviations are stated as well.

population size	number of guesses	evaluated codes	eligible codes considered	computation time (ms)
150	8.88 ± 1.25	2858.10 ± 1803.62	391.64 ± 72.95	7902.18 ± 8633.18
300	8.87 ± 1.18	4991.44 ± 1353.74	813.78 ± 148.14	13449.63 ± 6528.34
600	8.92 ± 1.11	9669.44 ± 1720.43	1604.93 ± 292.67	24287.67 ± 8764.11

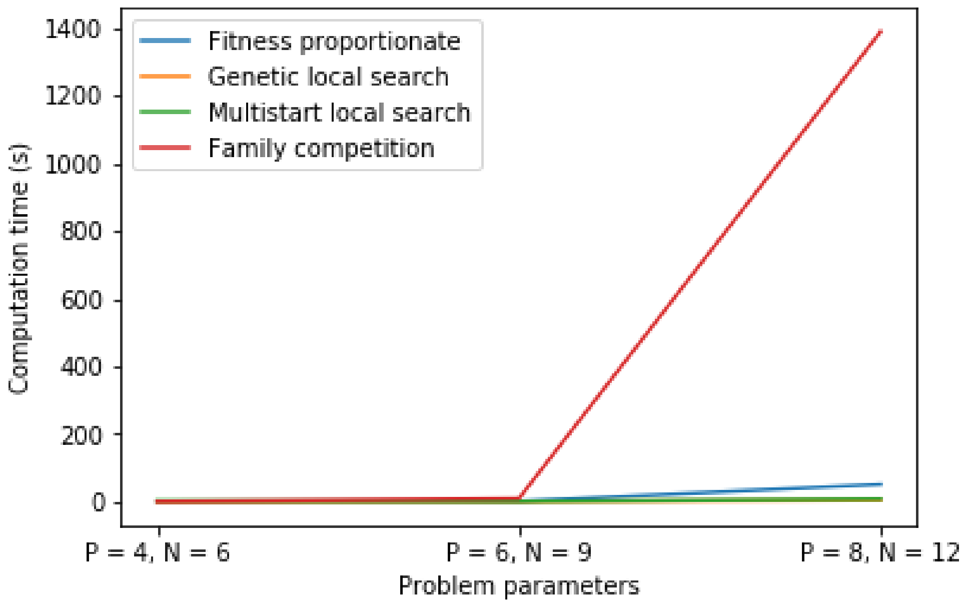
Table 4.12: Average number of guesses, number of evaluated codes, number of eligible codes found and computation time in milliseconds for multi-start local search with $P = 8$ and $N = 12$. Standard deviations are stated as well.

In Tables 4.10, 4.11 and 4.12 it can be seen that for multi-start local search, there is also no clear effect of a larger value for the population size when it comes to the average number of guesses needed to solve the problem.

Something strange that can be seen in Table 4.10 is that both the average computation time and its standard deviation are much higher for the smallest value of the population size than for the other two values. This is due to three extreme outliers in the results of this part of the experiment, though the number of guesses that were evaluated during these runs is not unusually high. Perhaps the extreme values were caused by the algorithm taking a long time to apply local search to one or more of the evaluated codes. If these three outliers are disregarded, the average computation time is 281.50 ± 191.03.

The multi-start local search algorithm seems to scale best with respect to problem parameters for the smallest value of the population size. In Figure 4.1 it can be seen that multi-start local search scales better than both the original algorithm and the variant with family competition, but it cannot beat genetic local search. Meaning that while multi-start local search does not seem to perform worse than genetic local search in terms of the average number of guesses, the GA in genetic local search does seem to contribute to the algorithm's scalabitlity. This is probably because multi-start local search evaluates more codes than genetic local search, which can also be seen in the relevant tables (4.7, 4.8, 4.9, 4.10, 4.11 and 4.12). Genetic local search is more effective at exploring the search space than multi-start local search.

In order to gain better insight into whether or not the difference in scalability between these two algorithms is significant, an unpaired t-test was performed, with $\alpha = 0.1$ and a confidence interval of 95%, comparing the computation time for genetic local search and multi-start local search, each for the smallest tested population size, for $P = 8$ and $N = 12$. The results of this t-test show that there is indeed a significant difference in the computation time, t(200) = -1.7362, p = 0.08408, 95% CI.



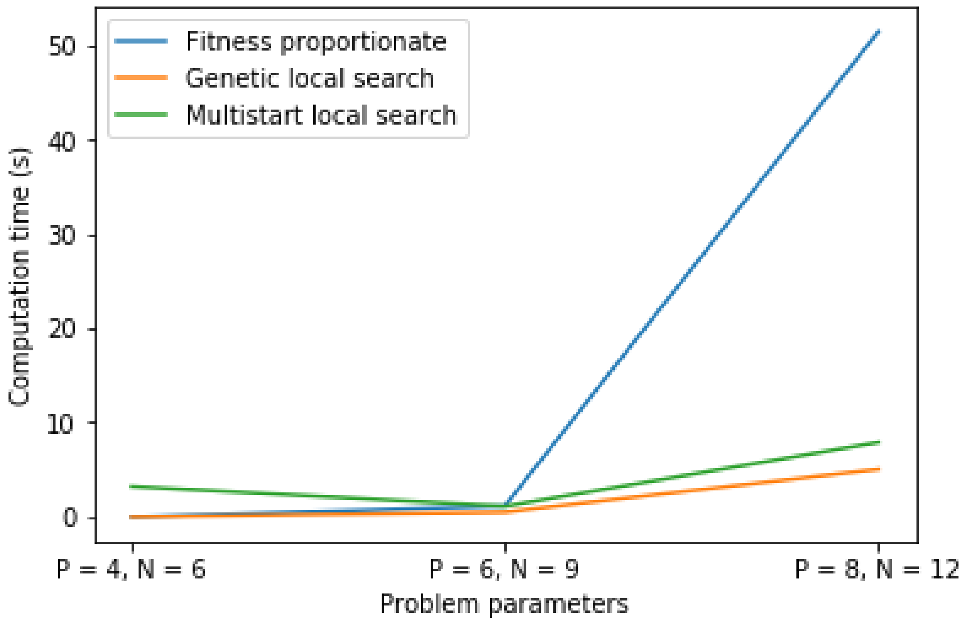


Figure 4.1: Illustration of the scalability of all four algorithms. For family competition the largest tested parameter values were used, for the rest the smallest tested values. In the bottom image, family competition is left out, so the others can be compared more easily.

4.5 Summary

Now that the results of the experiment have been presented, first the sub-questions as formulated in Section 3.2 and then the main research question will be answered.

- How well does the original algorithm scale?

  The original algorithm scales much better than a full enumeration algorithm and has no problem solving Mastermind in a reasonable amount of time for $P = 8$ and $N = 12$. It does not come close to scaling linearly however, and as problem parameters grow further, the computation time of the algorithm will grow increasingly quickly.

- How well does the algorithm perform and scale when family competition is used instead of fitness proportionate selection?

  The algorithm scales significantly worse when family competition is being used instead of the original fitness proportionate selection.

- How well does the algorithm perform and scale when genetic local search with family competition is used?

  Out of the variants of the original GA tested, genetic local search scales the best by far. Though it does not scale linearly with respect to problem parameters, it would be fit for solving Mastermind with values for P and N greater than the ones used during this experiment.

- Does the GA in the genetic local search algorithm contribute significantly to the performance of the algorithm, i.e., how does genetic local search compare to multi-start local search?

  While multi-start local search seems to perform nearly as well as genetic local search, the difference in computation time is significant. It cannot be said on the basis of this experiment alone whether or not they perform equally well in terms of the number of guesses needed on average, but it has been determined that the GA in the genetic local search algorithm does contribute to the algorithm's scalability.

- What is the effect of the values of the population size and the maximum number of generations on the performance and on the scalability of each of the algorithms tested (Section 3.1)?

  During this experiment, no significant difference in performance was found in terms of the number of guesses needed on average. Those parameters seem to mostly influence the computation time of the algorithm. Family competition being the exception, the algorithms scale better for smaller values of those parameters.

No significant difference in performance was noticed between the algorithms. Now for the main research question: how well does the GA presented by Berghman et al. (2009) scale with respect to problem parameters?

While the algorithm scales well compared to many other algorithms that were discussed in Chapter 2, a modification of it using genetic local search scales significantly better still and would be an appropriate solution for solving the problem, even for larger problem parameters.

Chapter 5

Discussion

In previous work, the most focus was often put on the performance of an algorithm in terms of the average number of guesses it needs to solve Mastermind. Here, the required computation time and scalability are the aspects that were studied the most closely. Of course, an algorithm is more relevant when it needs fewer guesses to solve the game, but if it can only do it for a small search space, it is not interesting. The two aspects need to be balanced. In a large part of the work done previously on the subject of solving Mastermind with algorithms, scalability, often even computation time, were not discussed in detail. Sometimes not at all. Even if it was stated, it was not always done in the same manner and for the same size of the search space. This makes it more difficult to compare findings from the experiment performed here to previous findings; however, genetic local search seems to be an algorithm that scales very well compared to other algorithms. Though from this experiment, it is not evident how exactly it compares to other algorithms in terms of the average number of guesses it needs, it seems to perform quite well in that sense too.

From the results obtained with this experiment, it is not clear which of the tested algorithms performs best in terms of the average number of guesses needed. There is no evident line of how well each algorithm performs compared to the others. It is also not clear what the effect of different values for the parameters population size and, where applicable, the maximum number of generations is on the average number of guesses needed, though this experiment has shown the influence of those parameters on the algorithms' scalability. In Figure 5.1 examples are shown that for different values for the parameters of an algorithm, the differences in the results are not always the same. Often there seems to be nearly no difference, sometimes the spread of the results might become smaller as parameters grow. It might also occur that there is no clear pattern at all. It is possible that the algorithms were not run enough times to get an accurate indication of how well they perform, especially since they are not deterministic.

In a possible follow-up study, two or more of the algorithms tested here could be compared to each other in more detail when it comes to the number of guesses needed, or the size of the algorithms' parameters might be explored further, in order to gain a better understanding. Another possibility is to compare one or more algorithms from this experiment to one of the other algorithms mentioned in the literature review (Chapter 2). Perhaps a different type of algorithm, in order to gain more insight on how the scalability of the algorithms compare.

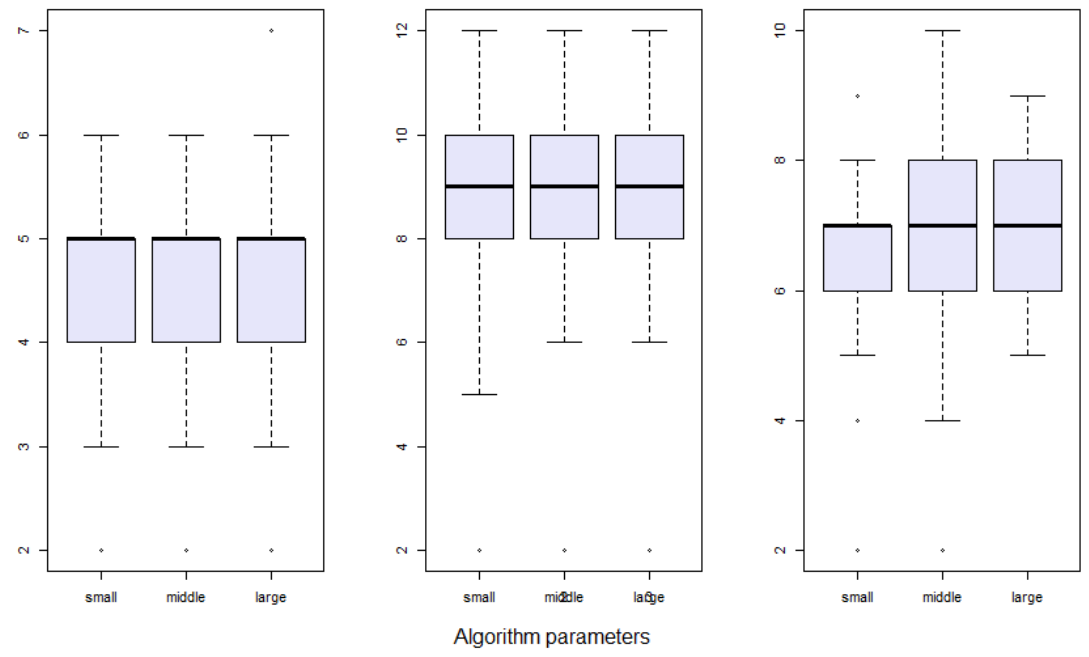


Figure 5.1: Boxplots displaying the number of guesses needed for multi-start local search for $P = 4$ and $N = 6$ (left), genetic local search for $P = 8$ and $N = 12$ (middle), and multi-start local search for $P = 6$ and $N = 9$ (right). These are examples of how varied differences between results are for different values for the parameters of the algorithms.

Chapter 6

Conclusion

The GA presented by Berghman et al. (2009) performs well both in terms of the number of guesses it needs and the computation time it requires, compared to other algorithms discussed in the literature review (Chapter 2). A variation on it using genetic local search scales even better however, and will therefore become increasingly more useful than the original as parameters grow. In order to compare the two in terms of the number of guesses they each need, a follow-up study is necessary.

7 References

Baum, E. B., Boneh, D., & Garrett, C. (1995). Where Genetic Algorithms Excel. *Proceedings COLT*, 230-239.

Bento, L., Pereira, L., & Rosa, A. (1999). Mastermind by Evolutionary Algorithms. *Proceedings of the 1999 ACM Symposium on Applied Computing*, SAC'99, 307-311.

Berghman, L., Goossens, D., & Leus, R. (2009). Efficient Solutions for Mastermind Using Genetic Algorithms. *Computers and Operations Research*, 36(6).

Bernier, J. L., Ilia Herrs, C., Merelo, J. J., Olmeda, S., & Prieto, A. (1996). Solving Mastermind Using GAs and Simulated Annealing: A Case of Dynamic Constraint Optimization. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1141, 554-563.

Irving, W. (1979). Towards an Optimum Mastermind Strategy. *Journal of Recreational Mathematics*, 11(2), 81-87.

Kalisker T., & Camens D. (2003). Solving Mastermind Using Genetic Algorithms. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2724, 1590-1591.

Knuth, D.E. (1976). The Computer as Master Mind. *Journal of Recreational Mathematics*, 9(1).

Kooi, B. (2005, March). Yet Another Mastermind Strategy. *ICGA Journal.*

Neuwirth, E. (1982). Some Strategies for Mastermind. *Zeitschrift Für Operations Research*, 26(1).

Norvig, P., & Berkeley, U. C. (1984). Playing Mastermind Optimally. *SIGART Bull.*, 90, 33-34.

Shapiro, E. (1983). Playing Mastermind Logically. *SIGART BULL.*, (1), 28-29.

Singley, A. M. (2005). Heuristic Solution Methods For The 1-Dimensional and 2- Dimensional Mastermind Problem. Master's thesis, University of Florida.

Stuckman, J., & Zhang, G.-Q. (2005, December). Mastermind is NP-Complete.

Temporel, A., & Kovacs, T. (2003). A Heuristic Hill Climbing Algorithm for Mastermind. *Proceedings of the 2003 UK Workshop on Computational Intelligence*, UKCI-03, 189-196.