Fastermind:
Using a SAT-solver to play Mastermind more efficiently

Natalie Collina
Adviser: Zachary Kincaid

May 2018

Abstract

Mastermind is a popular code-breaking game in which one player constructs a hidden ordered sequence of n pegs, each taking one of m colors. The second player attempts to determine this code through playing their own sequences and receiving information in return. Brute-force implementations of Mastermind-playing algorithms scale poorly with n and m. Even genetic algorithms designed to improve runtime fail to efficiently solve large games. To create an efficient and successful Mastermind player, this paper turns to the Mastermind Satisfiability Problem (MSP), an NP-complete problem which determines if there exists a valid query given a Mastermind game history. This paper presents the first reduction from MSP to SAT and uses an incremental SAT-solver to determine effective queries. With large values of n and m, we find that the MSP-to-SAT Mastermind player wins games in a small fraction of the time of any previously existing algorithm, and does so with only minor sacrifices in expected turn count. Turn-by-turn runtime comparisons and best- and worst- case inputs were also examined.

1. Introduction

1.1. The Game

Mastermind is a turn-based, limited-information game involving two players. In the original game, players have access to pegs of 6 possible colors. Player 1 begins by selecting a secret ordered combination of length 4 using these pegs. The code may include multiple pegs of the same color.

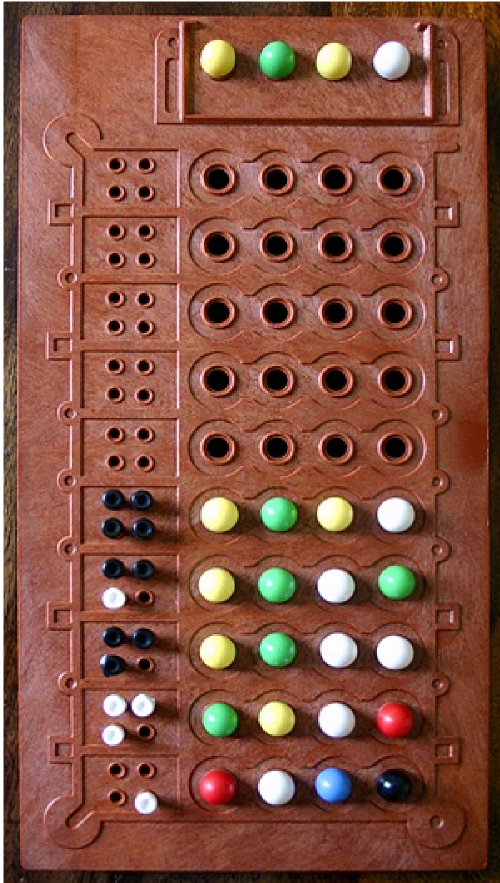

Figure 1: A Mastermind Board after player 2 has won in five turns. Player 2's queries begin at the bottom row and progress upwards. The revealed secret code is at top of the board.

After the secret code is established, turns will proceed as follows:

1) Player 2 plays an ordered combination of colored pegs on the board

2) If player 2's combination is the same as the secret code, player 2 wins the game. Otherwise, player 1 places b black and w white pegs next to the query code. This indicates that $b+w$ of the pegs in the query are in the secret code in some order, and b of these are in the correct position.

Player 2 plays combinations until they win. If they do not win within 10 turns, player 1 wins.

While the original Mastermind game is played with six possible colors and a code length of four, the MSP-to-SAT algorithm will be tested using games with various code lengths and color options. For the remainder of this paper, n will refer to the number of pegs in the secret code and m will refer to the number of possible colors in the game. Furthermore, a game with n pegs in the code and m colors will be referred to as an (n,m) game.

1.2. Goal

The goal of this project is twofold: specific to game of Mastermind, it was to build a Mastermind player that could complete games in a reasonable time while still retaining a low expected turn count. More generally, this paper serves to enrichen the literature on SAT reductions. While playing Mastermind efficiently may not have direct real-world implications, the techniques used in this new reduction may serve to aid future reductions in fields such as cryptography.

1.3. Summary of Project

Much previous work on the topic of Mastermind has focused on minimizing the number of turns for player 2 to win. However, these algorithms have high computational complexity and are impractical to use on games with larger n and m. The approach taken in this paper retains a low expected turn count and reduces the time needed to win large games by an order of magnitude. MSP-to-SAT takes inspiration from Swaszek's Random algorithm, which simply plays random valid moves each turn and garners a surprisingly low expected turn count. [9] However, MSP-to-SAT finds a valid move through a reduction to SAT and the use of a SAT-solver. The solver used in this paper was SAT4J. We hypothesize that playing valid moves found by the SAT-solver will be strategically similar to playing random valid moves, and therefore that MSP-to-SAT will retain reasonably low expected turn count.

Overall, MSP-to-SAT was found to be slower than brute-force algorithms when tested against small n and m, including the actual game values of $(4,6)$. However, it is faster than these algorithms it was tested against when n and m are large. Furthermore, MSP-to-SAT has a turn count equivalent to the random valid move strategy on almost all inputs.

MSP-to-SAT was also compared to two genetic Mastermind algorithms, which were created with the intent of sacrificing an optimally low turn count for a better runtime, similar to the goal of MSP-to-SAT. MSP-to-SAT has comparable turn count to these algorithms and is much faster. For example, it outperforms the fastest examined algorithm by a factor of 10 on $(8,12)$ games.

In addition to determining MSP-to-SAT's efficiency, this paper analyses its runtime turn-by-turn, showing that it avoids the Random algorithm's exponential increase in runtime each turn of the game. Finally, we consider best- and worst-case inputs, showing that codes with more colors garner poorer results and codes of all one color are easy to solve.

2. Related Work

In all previous work, and in this paper, player 1 is assumed to be playing uniformly random code combinations during testing.

2.1. Turn-minimization algorithms

The overwhelming majority of previous work on Mastermind has involved minimizing the expected number of turns for player 2 to win. The earliest known Mastermind algorithm, presented by Donald Knuth in 1977, did just this. In Knuth's strategy, each turn player 2 considers all possible moves and all possible consistent responses by player 1. They score these responses based upon how many secret code possibilities are eliminated and select the move with the best elimination score in the worst case (the maximin). This algorithm wins in an expected 4.478 turns in the $(4,6)$ case. However, it takes exponential time to complete and has a large search space. [4] A turn-optimal solution for the $(4,6)$ case was found in 1993 by Koyama and Lai. Using an exhaustive depth-first search on a supercomputer, they were able to find an algorithm that wins in an expected 4.340 turns. [5]

Other algorithms, such as two suggested by Peter Swaszek in 2000, have sacrificed complete optimality of turn count for simplicity. One of Swaszek's algorithms, which we will refer to as the Random algorithm, selects a random valid query each turn. We define a valid query as one that could still be the secret code, given the information received by player 1. Swaszek found that the Random algorithm won in 4.638 turns on average in the $(4,6)$ case. A second algorithm, his Iterative algorithm, treats codes as n-digit numbers and plays the lowest valid query each turn. The Iterative algorithm wins in 4.758 turns on average in the $(4,6)$ case. [9]

While these algorithms have lower computational complexity than Knuth's and Koyama and Lai's, they are still exponential. The Iterative algorithm must consider on average half the number of possible secret codes throughout the game. Near the end of the game when few valid code combinations remain, the Random algorithm may consider close to the entire search space in a single turn. A turn-count comparison of the previously discussed algorithms in the case of a typical Mastermind game of $(4,6)$ can be seen below.

Algorithm	**Expected Turns, (4, 6)**
Koyama and Lai	4.340
Knuth	4.478
Swaszek: Random	4.638
Swaszek: Iterative	4.758

Table 1: Comparison of Turn-minimization algorithms: (4, 6)

2.2. Time-efficient algorithms

While the majority of Mastermind algorithms have focused on simply reducing turn count, some work has been done on reducing computational complexity using genetic algorithms. In the context of Mastermind, genetic algorithms consider an initial subset of potential queries each turn and score these moves according to a fitness function. High-scoring queries are then "mutated," or slightly altered, to attempt to find an optimal move.

Kalisker and Camens (2003) present a genetic algorithm where values are assigned to potential queries by comparing these queries to previous queries. Each previous guess is treated as a secret code, and the b and w that the potential new query would have elicited for that secret code are input into the fitness function. [3] Through this method, Kalisker and Camens optimize the validity of their next query in a manner similar to Swaszek's algorithms.

Another genetic algorithm was proposed by Berghman et. al. Instead of comparing the potential new queries to previous plays, this algorithm compares them to other codes in the sample space. [6] In doing this, it favors algorithms that will reduce the number of valid moves in a manner similar to Knuth's algorithm. A comparison between the two algorithms can be seen below.

Algorithm:	**Turns, Runtime/Turn: (4, 6)**	**Turns, Runtime/Turn: (6, 9)**
Kalisker	4.75 turns, 0.26 sec	7.27 turns, 2.31 sec
Berghman	4.39 turns, 0.762 sec	6.475 turns, 1.284 sec

Table 2: Comparison of Genetic Algorithms; all values are expectations

2.3. The Mastermind Satisfiability Problem and NP-completeness

A separate but related topic in Mastermind research is the Mastermind Satisfiability Problem, or MSP. This decision problem takes in a Mastermind game history and determines if there is a secret code consistent with all the information provided. More technically, it takes as input a set of query-response pairs $(q_t, (b_t, w_t))$, each of which sets certain constraints on the secret code, and determines if a valid solution exists that satisfies all constraints.

Stuckman and Zhang (2005) proved that MSP is NP-complete by reducing from the NP-complete problem Vertex-Cover to MSP. Vertex-Cover takes as input a graph G and determines if there exists a set of vertices of size n such that each edge in the graph is adjacent to at least one of these vertices. The general idea of the reduction is that each distinct vertex and edge in G is represented by a color. Queries are constructed for each edge, encoding the idea that at least one of the vertices that edge is connected to must be in the vertex cover. A final query encodes the constraint that n of these vertices total must be in the vertex cover. A solution to MSP, excluding the control colors, represents the set of n vertices in the vertex cover. [8]

Given that MSP is NP-complete, it is in the same complexity class as SAT. SAT is a decision problem which takes as input a conjunction of disjunctions of boolean variables and asks whether there is an assignment of values to the variables such that the expression evaluates to true. The relationship between MSP and SAT is key to the new algorithm.

3. Approach

3.1. Key novel idea

In this project, we developed and implemented the first poly-time reduction from MSP to SAT. We then created a Mastermind player that feeds the output of this reduction into an incremental SAT-solver to find a new valid move each turn. This project combines two key ideas from the literature:

1) **Playing a valid code each turn works surprisingly well for turn minimization**, as demon-

strated by Swaszek's algorithms. [9] However, no previous work has developed a speedy way to find this satisfying move, opting instead to check all possible queries randomly or incrementally against the game history.

2) **Finding a valid code (MSP) is NP-complete**. This means that a poly-time reduction certainly exists from MSP to SAT, and this reduction does not make the problem more computationally complex. SAT is a well-researched problem and there are many SAT-solvers that can efficiently find satisfying instances to huge SAT inputs. Therefore this project takes advantage of SAT-solver efficiency through reducing MSP to SAT.

4. Implementation

4.1. Reduction Setup

From the outset, there is a clear relationship between MSP and SAT. MSP asks whether, given a game history, there is a valid possibility remaining for the secret code. SAT asks whether, given a set of constraints, there is a valid assignment of boolean values. Broadly, our reduction transforms pegs into variables and a Mastermind game history into constraints.

4.1.1. Overview: Variables and Definitions Let us define the secret code constructed by player 1 as s. Next, we define $m*n$ boolean variables in the following way:

$$x_{i,j} = \begin{cases} 1 \text{ if } s[i] = j \\ 0 \text{ otherwise} \end{cases}$$

Conceptually, for each of n positions in the code we will have m variables, each representing the event that a specific color is at that position. Each query by player 2 can be viewed in terms of these variables. For example, in a game with $n = 4$, $m = 6$, a query of 1550 is the correct answer iff the variables $x_{0,1}$, $x_{1,5}$, $x_{2,5}$ and $x_{3,0}$ are all 1.

Let us also define the tth query in the game history as q_t and the tth response pair as $\{b_t, w_t\}$.

In the following sections, we will detail transformation of the game history input into a SAT input.

There are three constraints:

1) Each position in the secret code must contain exactly one color

2) For each $\{q_t, \{b_t, w_t\}\}$, q_t and s share b_t positions containing pegs of the same color

3) For each $\{q_t, \{b_t, w_t\}\}$, there are $b + w$ pegs in common between the q_t and s

These constraints will be defined mathematically in the following sections.

4.1.2. Constraint 1: Defining the Game Regardless of the input, the first clauses in our boolean expression must encode the fact that each position in the code must have exactly one color.

For each of position i in the code, we have a set of m booleans representing the presence of a certain color at that position. We can create a constraint that exactly one of these is true by selecting the first two of these booleans, let's call them x_{ia} and x_{ib}, and creating a new boolean $x'_1 = x_{i1} \vee x_{i2}$. Then we will create the constraint $\neg(x_{i1} \wedge x_{i2})$. For each following x_{ij}, we will set

$$x'_j = x'_{j-1} \vee x_{i(j+1)}$$

and also create the constraint

$$\neg(x'_{j-1} \wedge x_{i(j+1)})$$

At the end, we confirm that the final $x'_{i(m-1)}$ is true. This ensures that at least one boolean in the original set was true. However, the constraints formed along the way ensure that no more than one of the booleans is true, so exactly one must be true.

4.1.3. Constraint 2: B Constraint Next, we must encode information about each query-reply pair, beginning with the B constraint. For $1 \leq x \leq n$, there are exactly b_t different values of x s.t. $s[x] = q_t[x]$. q_t can be seen as a sequence $x_{1c_1} x_{2c_2} \ldots x_{nc_n}$, where c_i represents the color of the particular element in the query. Therefore the query maps to n distinct x_{ij} variables. As each variable is either a 1 or a 0, this constraint can be represented by summing up the x variables representing q_t and checking that the sum matches b.

$$\sum_{a=1}^{n} x_{ac_a} = b_t$$

To do this, it is necessary to create new booleans representing the results of multiple summations. However, doing the additions this in sequence would lead to single bits being added to increasingly large numbers, which is not space-efficient. This addition is therefore encoded using a tree of adder circuits, which recursively splits the array of variables in half and then adds back up the tree.

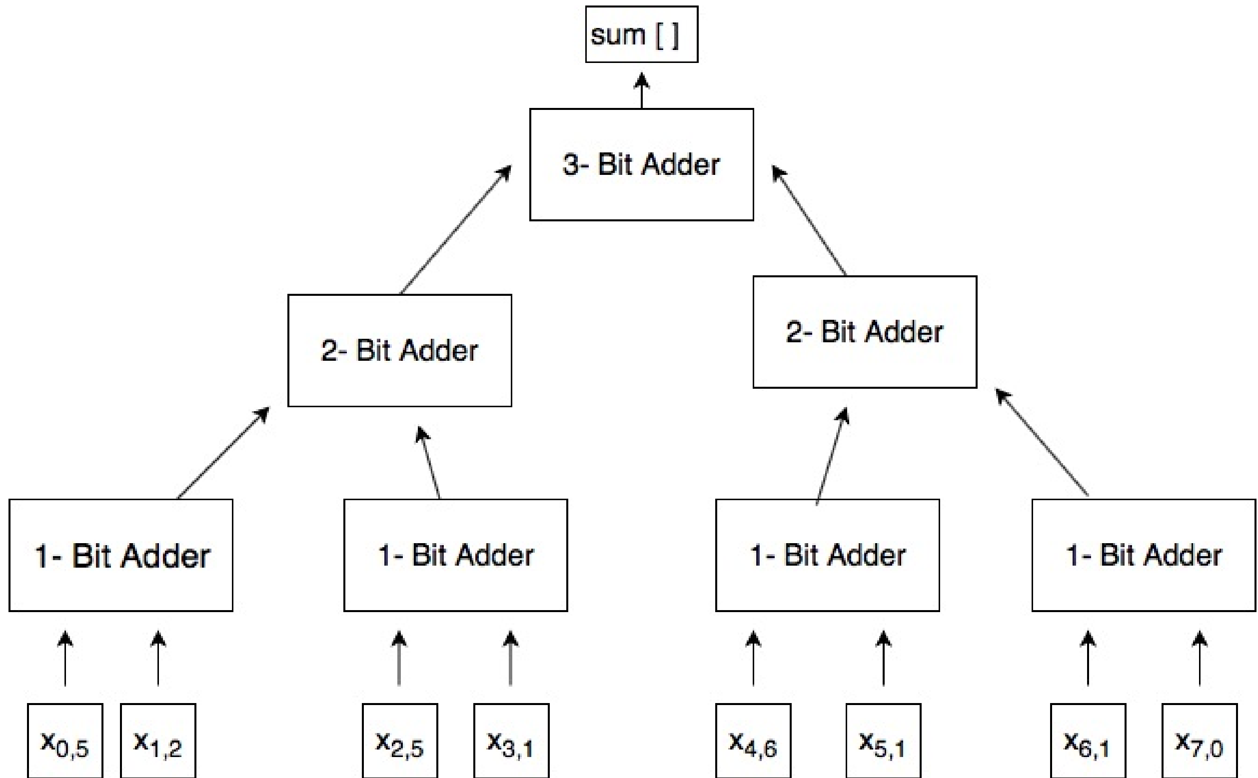

Figure 2: An example adder tree given a query of 52516110

4.1.4. Constraint 3: B+W Constraint We have previously described $b_t + w_t$ as providing information about the number of pegs in q_t which are present in s. For the purposes of this reduction, however, we will use an equivalent definition which is less intuitive but much more useful.

Let us define $f(q_t, c)$ as a function that takes in a query q_t and a number $1 \leq c \leq m$ and outputs the number of times that this number (representing a color) appears in q_t. Then:

$$b_t + w_t = (\sum_{c=1}^{m} \min\{f(q_t, c), f(s, c)\})$$

The values for $f(q_t, c)$ is already known, because we are examining a past guess to gain this information. Therefore our encoding must only find $f(s, c)$. We can do this as follows:

$$f(s, c) = \sum_{i=1}^{n} x_{ic}$$

Therefore we can describe this expression in the following way:

$$b_t + w_t = (\sum_{c=1}^{m} \min\{f(q_t, c), \sum_{i=0}^{n} x_{ic}\})]$$

Determining the minimum of two values is also something that can be efficiently encoded by comparing the corresponding bits of the numbers. The first step involves finding which number is smaller for each $c \in m$: the pre-calculated frequency of this color in the query code q_t, or the frequency of this color in out satisfying solution s. The frequency in s of color c, found using adder circuits, is represented by an array of booleans. Let's call this b_c. The known frequency of color c in q_t is represented in its binary form. Let's call this $known_c$. We would like to define a boolean g_c that is 1 if the $known_c \geq b_c$, and 0 otherwise. Let us define a recursive function knownMax(index, known, b) and set g_c = knownMax(0, $known_c$, b_c). The function works as follows:

```
knownMax(index) {
    if (index >= known.length) return TRUE;
    else if (known[index] == 1) (NOT b[index]) OR knownMax(index+1);
    else if (known[index] == 0) (NOT b[index]) AND knownMax(index+1);
}
```

Our intuition is that when we compare an unknown binary number to a known number (with equal lengths), and our known number begins with a 1, then the known number is certainly bigger if

the unknown number begins with a 0. If the unknown number begins with a 1, we can compare further down the binary string.

On the other hand, if our known number begins with a 0, then the unknown number must begin with a 0 AND the known number must have a larger value further down in the binary string.

Finally, we can use this g_c to set $\min\{f(q_t, c), \sum_{i=0}^{n} x_{ic}\}$ for each c.

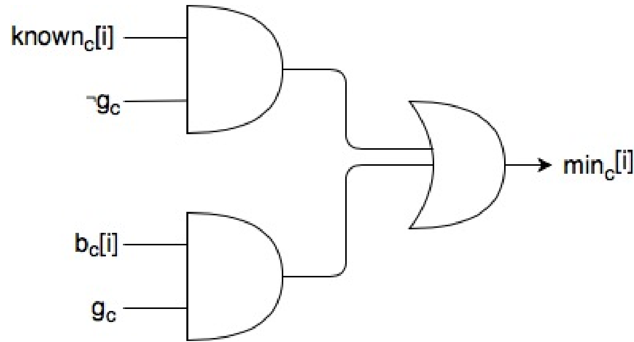

Figure 3: The process for setting the *i*th bit of the minimum frequency of color c

g_c acts as a "control switch" that maps a new set of booleans to b_c if $g_c = 0$ and to $known_c$ if $g_c = 1$. The general logic for setting the ith index in the minimum number is shown in Figure 3.

4.1.5. Summary We now have a top-level structure for the three constraints needed for each query-response pair in the MSP. A top-level encoding of an entire MSP input, given T previous moves, is therefore:

$$\textit{Defining the Game} \wedge \bigwedge_{t=1}^{T} (\bigwedge_{i=1}^{n} (\sum_{j=1}^{m} x_{ij} = 1) \wedge \sum_{a=1}^{n} x_{ac_a} = b_t \wedge (\sum_{j=1}^{m} \min\{f(q_i, j), \sum_{i=0}^{n} x_{ij}\}) = b_t + w_t)$$

4.2. Coding Implementation

Once the reduction was fully theorized, code was created that took in a representation of a Mastermind game board and translated it into a SAT problem in dimacs-CNF format. This is the input format that all SAT-solvers accept. It was then input into Minisat [2], a popular SAT-solver that is used as the underlying framework of many other SAT solvers. This was a useful way to test each constraint as the reduction was being implemented.

4.2.1. Incremental SAT-solving As a Mastermind game goes on, more constraints are added each turn, but old constraints are never removed. In other words if a previous turn provided a player with information that information still must hold true in the current round. Therefore re-computing MSP each turn on the same game board with just one new turn is inefficient. Incremental SAT-solvers deal with this issue by saving their computational state after being queried. New constraints can be added to the solver at any time, leading to incrementally more refined solutions.

The initial implementation was incapable of saving the previous turn information, as it was using Minisat from the command line. It was necessary to re-run the entire MiniSAT program each time a turn was completed. To solve this issue, a SAT-solver was embedded into the code itself using SAT4J, a Java wrapper for MiniSAT. [1] This allows the Mastermind player to avoid re-computing previous constraints each turn.

5. Testing setup

5.1. Constructing Player One

To test MSP-to-SAT against other Mastermind algorithms, a program was created that generates a random secret code and communicates with a player 2, providing $b+w$ feedback on moves each time. This player 1 code announces when player 2 wins, along with the number of turns taken. This player 1 code was able to interact with various implementations of player 2. This allowed for easy testing of both runtime and turn count across various algorithms.

To allow MSP-to-SAT algorithm to play, another function was created that translated the SAT4J output back into the format that Mastermind player 1 accepts.

5.2. Brute-force strategies

In order to determine if MSP-to-SAT performed better than brute-force approaches, it was compared with both of Swaszek's algorithms. Limited data was available on the average turn count and runtime of these algorithms outside of the $(4,6)$ case. To compare MSP-to-SAT with these turn-optimal algorithms, Swaszek's algorithms, Incremental and Random, were both implemented in code. To

check that the implementations were accurate, the average turn counts for $(4,6)$ were compared with the published results, and the differences were found to be statistically insignificant.

Once these algorithms were implemented, it was possible to directly compare their runtimes and expected turn counts by running many trials with varying n and m.

6. Results

Testing was performed by playing the algorithms against secret codes generated uniformly at random by our player 1 implementation. All data is averaged over 100 trials.

6.1. Comparison with Brute-force Algorithms

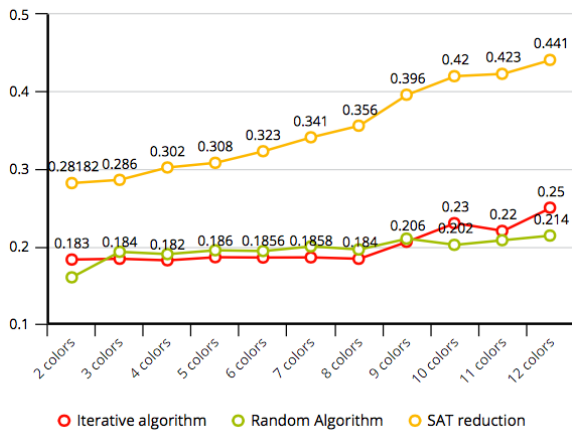

Figure 4

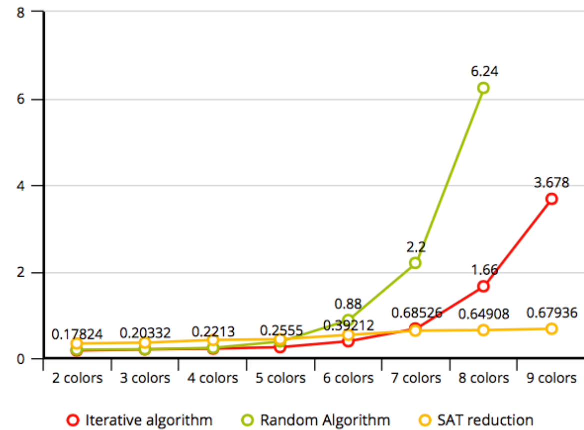

Figure 5

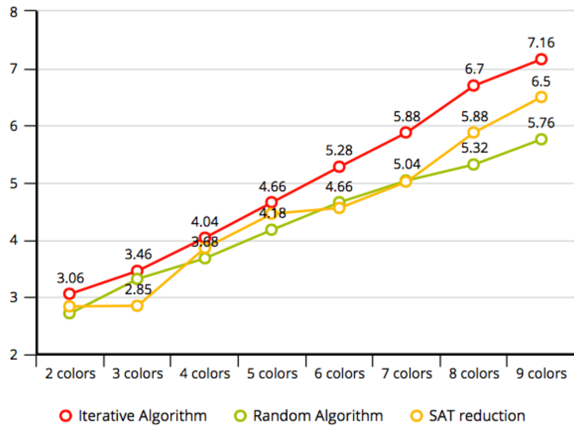

Figure 6

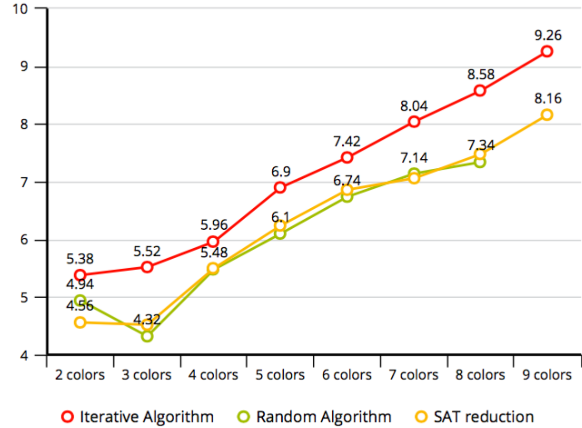

Figure 7

6.2. Runtime per Turn comparison

In addition to overall times, the average time taken to select a query each turn was measured. These values are conditioned on the algorithm having not yet terminated by the given turn. Furthermore, data is only included for turns where greater than 50% of the instances of each of the algorithms had not yet terminated. This was done to reduce the problem of small sample size and the uncertainty that comes with almost-completed games. Random was not included in the $(10,12)$ graph because its runtime was too slow to measure for such large n.

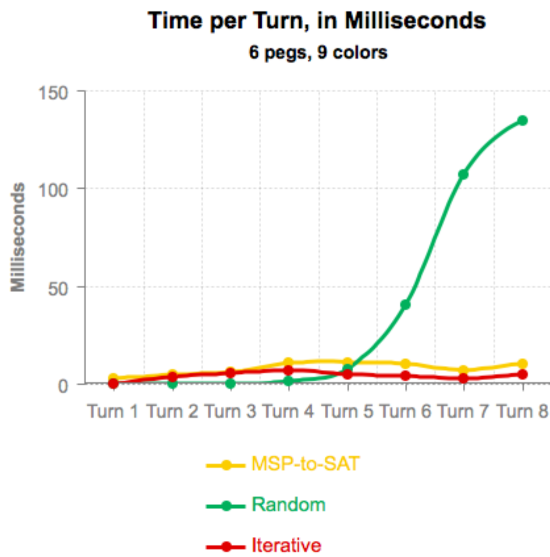

Figure 8

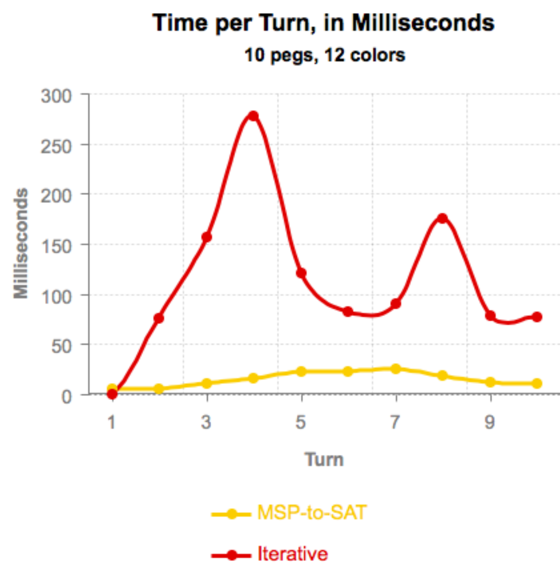

Figure 9

6.3. Comparison to Genetic Algorithms

The source code is unavailable for both the Berghman and Kalisker algorithms. However, average time and turn count numbers were available for various (n,m) combinations, and MSP-to-SAT was run on these values to provide comparison.

Algorithm:	**Berghman**	**Kalisker**	**MSP-to-SAT**
4 pegs, 6 colors	0.762 seconds, 4.39 turns	0.26 seconds, 4.75 turns	0.323 seconds and 4.62 turns
6 pegs, 9 colors	1.284 seconds, 6.475 turns	2.31 seconds, 7.27 turns	0.530 seconds, 6.98 turns
7 pegs, 10 colors	no data	6.72 seconds, 8.62 turns	0.638 seconds, 7.8 turns
8 pegs, 12 colors	9.711 seconds, 8.366 turns	no data	0.933 seconds, 9.56 turns

Table 3: Comparison between Genetic Algorithms and MSP-to-SAT

6.4. Larger Numbers

As MSP-to-SAT is able to efficently solve much larger Mastermind games than existing algorithms, tests were created with larger numbers to examine when the game size became impractical to solve.

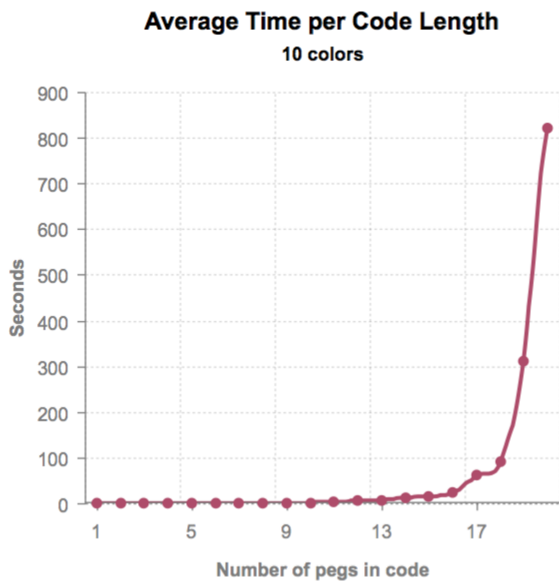

Figure 10

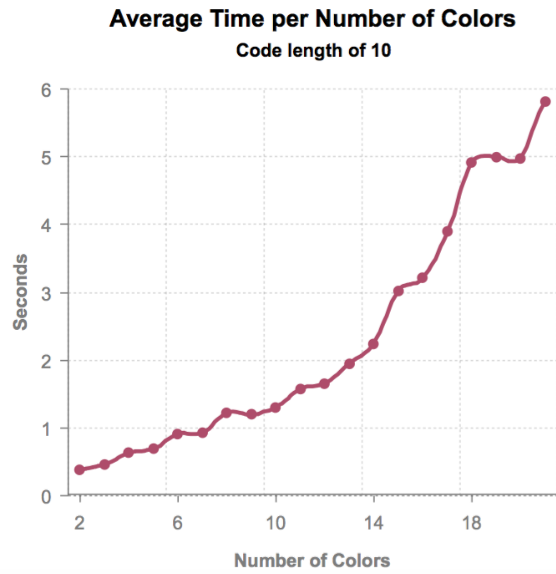

Figure 11

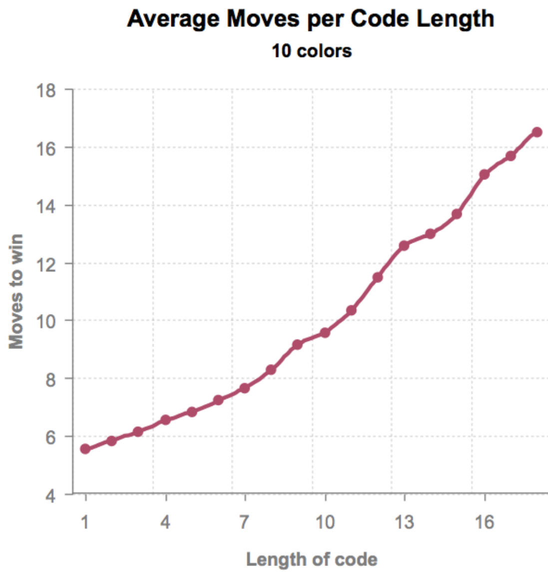

Figure 12

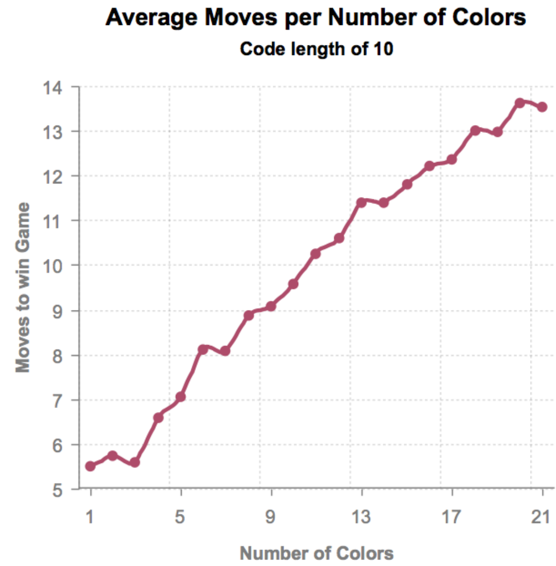

Figure 13

6.5. Best- and Worst-case inputs

Finally, we examined under which conditions the algorithm performed particularly well or poorly. Various secret codes were tested, and the best and worst of these were recorded.

Structure of secret code	Seconds	Turns
Maximizing # of colors	3.21	13
Uniformly at random	1.64	10.4
All one color	0.616	6

Table 4: Comparison of MSP-to-SAT performances for different inputs, 10 pegs and 12 colors

7. Discussion

When examining games with $n = 4$ and m ranging from 2 to 9 in figures 4 and 6, MSP-to-SAT is consistently slower than both Random and Iterative. MSP-to-SAT takes a similar number of turns as Random, though this number is slightly increased at higher values of m. [1] Analysis of $n = 4$ shows us, most significantly, that MSP-to-SAT is not the optimal choice for the actual Mastermind game of $(4,6)$. The algorithm's relatively high runtimes at lower input values are likely explained by its computational overhead. Unlike Random and Iterative, MSP-to-SAT must translate player 1's feedback to constraints in Conjunctive Normal Form (CNF) format at each turn. At small input sizes, this likely represents a significant portion of MSP-to-SAT's runtime.

However, when $n = 8$, MSP-to-SAT emerges as the clearly preferable algorithm, as seen in figures 5 and 7. It remains roughly equivalent to Random in expected turn count and avoids the exponential-time blowup experienced by Random and Iterative at $(8,7)$ and beyond.

It is important to note the similarity between MSP-to-SAT and Random in expected turn count. SAT-solvers are not typically uniformly random, and therefore using a SAT-solver was not guaranteed to produce the same results as the Random brute-force algorithm. The fact that their average turn counts are similar suggests that MSP-to-SAT is not significantly negatively impacted by selecting valid moves in a non-uniformly random fashion.

However, Random does seem to have a better turn count for exactly two (n,m) pairs tested: $(4,7)$

[1] Iterative consistently has the highest average turn count across all values of n and m tested. This may be because, though all algorithms play valid moves each turn, Iterative is playing the lowest possible valid move. This may include more colors in common with previously invalidated moves on average than a random valid move, and therefore provide less information on each round.

and $(4,8)$. This may simply be due to the unpredictability of the Random algorithm. Despite the fact that the values were averaged over many trials, the Random algorithm experiences so much variance that four or five particularly lucky or unlucky games could impact the expected turn count. On the other hand, it may be the case that at smaller values of n a random valid code is preferable to the satisfying solution found by SAT4J. In this case, utilizing a randomized SAT-solver may provide an improvement. Some initial investigatory work was done using Quicksampler, a SAT-solver which generates thousands of potentially correct assignments and checks them, quickly finding a satisfying assumption among the samples in a near-uniformly random manner. [7] Small trials suggested no significant difference between Quicksampler and SAT4J in terms of average moves. Furthermore, Quicksampler showed a slower solution time for individual turns than the SAT4J implementation. However, due to time constraints and the incompatibility between Quicksampler and all previously-written Java code, no definite conclusions can be drawn at this time on whether randomized SAT-solvers provide an improvement on expected turn count.

Figures 8 and 9 provide more information on why MSP-to-SAT has a lower runtime than Swaszek's algorithms. When examining the time taken on a turn-by-turn basis at $(6,9)$, Random is found to be the fastest on early turns but quickly slows. This makes intuitive sense, as this algorithm is trying random queries each time until one is found to be valid. This works quickly at the beginning when most codes remain valid, but could involve testing the majority of all possible color combinations during later turns. The Iterative algorithm avoids this exponential increase in $(6,9)$, performing similarly to MSP-to-SAT. When tested on $(10,12)$, however, it is consistently slower than MSP-to-SAT at all turns but the first. An explanation for the shape of Iterative's turn length graph is not immediately obvious. This may be an interesting area for future exploration.

The comparison of MSP-to-SAT with Swaszek's algorithms confirms the validity of the SAT reduction, demonstrates that using SAT-solvers to choose valid moves provides an improvement over the brute-force approach at higher values of n, and allows for a thorough turn-by-turn analysis of where MSP-to-SAT saves time. However, Swaszek's algorithms were designed to minimize turns at the expense of time efficiency, and therefore would be expected to scale poorly. MSP-to-SAT's

true test in the area of timing comes in the form of the two genetic algorithms examined in section 2.2, both of which were built with the goal of reducing the time complexity of Mastermind strategies while retaining a low turn count. [3] [6]

Table 3 shows that MSP-to-SAT outspeeds both algorithms as n and m increase. MSP-to-SAT provides a lower average turn count than every data point provided in the Kalisker paper. Furthermore, it provides a lower runtime in all cases but $(4,6)$. As stated previously, poor performance at lower input values may be explained by transformations to CNF format instead of the actual SAT-solver. Aside from this data point, MSP-to-SAT appears to completely outclass Kalisker's algorithm. Berghman's algorithm, on the other hand, consistently has lower expected turn count than MSP-to-SAT. However, it is significantly slower than MSP-to-SAT in all examples. While Berghman's turn count is roughly 90% of MSP-to-SAT's in the cases examined, MSP-to-SAT's runtime is less than 10% of Berghman's at large inputs.

Once MSP-to-SAT was established to be the algorithm most capable of solving large instances of Mastermind, it was stress tested to examine when it became impractical (see figures 10, 11, 12 and 13). The algorithm performs reasonably well as m increases and n is set at 10. Both time and turn count increase in a manner that appears roughly linear. When setting m to 10 and increasing n, however, though average moves continues to increase linearly, the runtime increases exponentially. At $(20,10)$, MSP-to-SAT takes more than 13 minutes on average to complete a game. As the number of possible codes in Mastermind is m^n, the fact that MSP-to-SAT scales more effectively with m is to be expected.

Finally, any set algorithm must consider worst-case inputs. If player 1 knows that player 2 is using MSP-to-SAT, then what could they play to punish player 1? While this analysis did not cover every single possible input, many dozens of possible inputs were tried. It seems that MSP-to-SAT's performance is tied to the number of distinct colors represented in the secret code. This improvement is partly due to the lower number of turns needed to win, but the ratio of time improvement when the number of colors is decreased is larger than the ratio of turn improvement, suggesting that each turn is in fact faster on average with less colors represented.

This may seem intuitive; playing a secret code of all one color seems easy to guess. When considering implementation of the algorithm, however, it seems less obvious. The reduction to SAT transforms the presence of a color c at each peg position into n distinct boolean variables. Therefore the direct relationship between same-colored pegs that is intuitive to a human Mastermind player is partially lost in the reduction. Despite this, it is still easy for MSP-to-SAT to solve instances of all the same color. A player constructing a secret code to play against MSP-to-SAT would be wise to maximize the number of distinct colors in their code.

8. Conclusion and Future Work

By reducing MSP to SAT, we were able to demonstrate a significant improvement in runtime for larger values of n and m. Furthermore, this improvement came at minimal cost; at significantly large inputs, MSP-to-SAT solves problems an order of magnitude faster than any of the other algorithms, and in all turn-count comparisons it is never outperformed by significantly more than a turn. Ultimately, MSP-to-SAT provides a promising example of utilizing a SAT reductions to take advantage of the efficiency of SAT solvers.

One area for future work would be to further explore randomized SAT-solvers. This paper hypothesized that satisfying solutions returned by SAT solvers would be more similar to uniformly random valid solutions computed in Random than to the deterministic valid queries found in Iterative. This proved to be correct, as MSP-to-SAT demonstrated an expected turn count almost always as low as Random. However, at $n = 4$, the Random algorithm seems to slightly outperform MSP-to-SAT in expected turn count. Further analysis must be done to determine if this difference is meaningful, and whether it can be mitigated by a randomized SAT-solver.

Another potential avenue for improvement could be to provide only probabilistically valid queries through computing probabilistically valid assignments. For example, MSP-to-SAT could check the most recent query-response pair and then half of the remaining query-response pairs to construct constraints. Alternately, a probabilistic SAT-solver could be used. Allowing for false positives would likely reduce the time complexity of each turn. Furthermore, errors would likely cause

minimal damage. Queries that are somewhat close to valid would still provide useful information to player 2 and bring them closer to winning the game. In fact, we know for sure from the turn-optimal algorithm by Koyama and Lai that invalid moves are sometimes the optimal choice. [5]

9. Acknowledgements

I would like to thank Zachary Kincaid for being an extremely engaged, kind and helpful advisor.

10. Appendix

```
nat-oitwireless-inside-vapornet100-c-27112:IW nataliecollina$ time java -cp ".:org.sat4j.core.jar" toSAT 20 20
1 6 7 14 7 11 3 19 4 6 9 7 10 10 3 3 17 8 8 13
2 3 0 3 3 10 4 3 3 3 3 0 3 3 3 2 2 3 3 2
16 0 4 17 17 18 1 3 18 2 10 15 10 14 14 10 14 14 14 14
10 10 4 4 4 4 10 4 4 4 0 4 19 10 2 4 4 3 4 4
4 1 10 11 11 3 3 14 11 11 15 15 1 11 11 4 10 3 13 13
13 2 1 17 4 3 6 4 11 14 2 5 3 4 5 5 9 5 15 1
19 3 3 13 19 13 3 13 17 13 0 17 4 15 14 17 11 6 4 13
1 18 18 18 18 18 13 14 4 13 3 13 1 17 4 3 3 4 4 3
19 18 10 9 5 10 1 6 17 11 13 3 19 19 19 3 3 19 11 4
10 10 3 5 13 18 3 5 9 5 11 13 1 3 10 13 13 0 11 6
4 10 18 1 19 15 19 15 18 9 3 3 11 5 6 4 6 6 6 3
1 3 5 8 17 9 3 7 12 19 17 4 16 10 7 11 3 4 13 11
3 7 6 8 11 8 8 5 4 13 1 17 6 10 7 3 10 14 3 19
3 3 8 10 1 6 3 14 4 8 10 11 19 17 7 6 7 13 9 9
6 6 11 19 7 4 3 14 1 13 7 9 8 10 8 10 3 17 3 7
3 14 7 3 17 13 7 8 6 10 7 9 1 10 11 6 3 8 4 19
3 3 7 19 6 9 10 7 7 1 6 13 8 11 8 3 10 17 4 14
1 17 8 4 6 7 3 8 3 13 14 6 10 10 11 3 19 9 7 7
8 7 3 8 3 1 10 14 6 13 19 4 10 11 17 6 3 9 7 7
7 19 13 7 17 7 3 14 1 3 8 6 6 10 9 3 8 11 4 10
1 6 3 6 8 7 14 3 4 7 17 9 8 10 11 3 7 10 19 13
1 8 7 6 7 11 3 19 4 6 7 8 17 10 3 3 14 9 10 13
1 6 7 14 7 11 3 19 4 6 9 7 10 10 3 3 17 8 8 13
1 6 7 14 7 11 3 19 4 6 9 7 10 10 3 3 17 8 8 13
23

real    7m6.545s
user    7m12.578s
sys     0m3.896s
```

Figure 14: Typical example of the MSP-to-SAT algorithm playing a $(20,20)$ game. The first line represents the secret code, the following lines represent MSP-to-SAT's queries, and the final successful query is repeated at the end

References

[1] D. L. Berre and A. Parrain, "The sat4j library, release 2.2," pp. 59–64, 2010.

[2] N. Een and N. Sorensson, "An extensible sat-solver," *SAT 2003*, 2003. Available: https://github.com/niklasso/minisat

[3] T. Kalisker and D. Kamens, "Solving mastermind using genetic algorithms," in *Lecture Notes in Computer Science*, C.-P. E. et al., Ed., vol. 2427, Genetic and Evolutionary Computation. Springer, Berlin, Heidelberg, 2003.

[4] E. Knuth, "Computer as master mind," *Journal of Recreational Mathematics*, no. 9, pp. 1–6, 1977.

[5] K. Koyoma and W. Lai, "An optimal mastermind strategy," *Journal of Recreational Mathematics*, no. 25, pp. 251–256, 1994.
[6] D. G. L. Berghman and R. Leus, "Efficient solutions for mastermind using genetic algorithms," Katholieke Universiteit Leuven, Department of Decision Sciences and Information Management, Tech. Rep. 0806, 2006. Available: https://lirias.kuleuven.be/bitstream/123456789/184247/2/mastermind.pdf
[7] J. B. Rafael Dutra, Kevin Laeufer and K. Sen, "Efficient sampling of sat solutions for testing," International Conference on Software Engineering. University of California, Berekley, 2018.
[8] J. Stuckman and G. Zhang, "Mastermind is np-complete," *CoRR*, vol. abs/cs/0512049, 2005. Available: http://arxiv.org/abs/cs/0512049
[9] P. Swaszek, "The mastermind novice," *Journal of Recreational Mathematics*, no. 30, pp. 193–198, 2000.

I pledge my honor that this paper represents my original work in accordance with University regulations. /s/ Natalie Collina