Scaling Temporel & Kovacs' Stochastic Hill Climbing Algorithm for Mastermind

Mark Pallone (`markpa1@umbc.edu`) Max Spector (`mspecto1@umbc.edu`)

December 17, 2011

1 Abstract

The strategy game *Mastermind* requires that one player guess another player's secret code based on limited feedback in between guesses until the code is discovered. Many computer scientists have presented different algorithms for playing *Mastermind*, including Alexandre Temporel and Tim Kovacs' stochastic hill climbing algorithm. Here, we present changes to this approach that allow the algorithm to work in larger problem spaces while still maintaining the effectiveness of its heuristics.

2 Introduction

Mastermind, invented in 1970-1971 by Mordecai Meirowitz, specifies two roles—a code maker and a code breaker.

The code maker must create a sequence G of n_c colors from a set containing k colors: $C = \{c_1, c_2, ..., c_k\}$. Duplicate colors are allowed and order matters (*e.g.*, $G_1 = \{c_1, c_1, c_2, c_2\} \neq G_2 = \{c_2, c_2, c_1, c_1\}$, but both are valid guesses).

The code breaker is aware of the length of the sequence n_c as well as the pool of possible colors C. She guesses until the code maker indicates that she has broken the code. After each guess, the code maker reveals how many elements of the sequence are the correct color and in the correct position, as well as how many elements are the correct color but in the incorrect position (these two amounts are hereafter referred to as *reds* and *whites*, respectively).

A common game size is $n_c = 4$ and $k = 6$ (4x6). For reasons soon to be revealed, we consider game sizes of 4x6, 5x7, 6x8, 7x8, and 8x8.

3 Background

We read several articles on different approaches to playing *Mastermind:*

- Investigations into the Master Mind Board Game[1]
- Solving Mastermind Using Genetic Algorithms[2]
- Efficient solutions for Mastermind using genetic algorithms[3]
- Mastermind Game[4]
- A heuristic hill climbing algorithm for Mastermind[5]

This reading in addition to an informal survey of University of Maryland, Baltimore County computer science students indicated that genetic approaches to *Mastermind* had already been done with fair success on larger-than-standard problem sizes.

Temporel and Kovacs' heuristic hill climbing algorithm did not indicate how well the algorithm scaled in extremely large problem sizes.

[1] `http://www.tnelson.demon.co.uk/mastermind/`

[2] `http://www.springerlink.com/content/2acvhpxa9cuvde6y/`

[3] `http://www.scribd.com/doc/36268449/Efficient-Solutions-for-Mastermind-Using-Genetic-Algorithms-2008`

[4] `http://delphiforfun.org/Programs/mastermind.htm`

[5] `http://www.cs.bris.ac.uk/Publications/pub_master.jsp?id= 2000067`

The unexplored nature of their approach as well as their interesting and novel algorithm led us to choose their stochastic hill climbing solution as the basis for our exploration.

4 Mathematical Background

There are several technical details guiding both our approach and the analysis of our results.

The number of possible secret codes given length n_c and the number of potential colors k is k^{n_c}. Hence, the standard 4x6 game has $1,296$ possible codes and an 8x8 game has $16,777,216$ possible codes.

Temporel and Kovacs' suggest a scoring function based on *reds* and *whites*:

- $reds = 0, whites = 0 \Rightarrow score = 0$
- $reds = 0, whites = 1 \Rightarrow score = 1$
- $reds = 1, whites = 0 \Rightarrow score = 2$
- $reds = 0, whites = 2 \Rightarrow score = 3$
- $reds = 1, whites = 1 \Rightarrow score = 4$
- etc.

Constructing a table of scores based on *reds* and *whites* reveals a formula for this calculation:

	$reds =$ **0**	**1**	**2**	**3**	**4**
$whites =$ **0**	0	2	5	9	14
1	1	4	8	13	19
2	3	7	12	18	25
3	6	11	17	24	32
4	10	16	23	31	40

Noting that:

- The numbers in column $reds = 0$ are triangular numbers,
- the difference between two horizontally adjacent cells increases by 1 as one moves to the right across the table, and
- the nth triangular number is equal to $\frac{n^2+n}{2}$,

we found that the score based on *reds* and *whites* can be calculated in constant time based on the formula

$$score = \frac{whites^2 + whites}{2} + (reds)(whites) + \frac{reds^2 + reds}{2} + reds$$

5 Temporel and Kovacs' Algorithm

5.1 Commutative Requirement

Scores in *Mastermind* are *commutative;* a guess scored against a secret code will receive the same number of *reds* and *whites* as the secret code scored against that guess.

An essential part of Temporel and Kovacs' algorithm is using this observation to ensure future guesses are consistent with old guesses. Each new guess that is generated is scored against each previous guess. If the *reds* and *whites* received does not match what the old guess received when it was scored against the actual secret code, then the new guess is thrown away and another one is generated.

This consistency check reduces the number of guesses that can be submitted, especially as the number of previous guesses grows. However, it can take a very long time to generate a guess that is consistent with every single previous guess, because the search space does not shrink.

5.2 Algorithm Overview

Any time a guess is created, it is checked against a hash table of previously generated guesses to ensure that duplicate guesses are not wastefully considered. If a newly generated guess is not in the hash table, then it is added.

If a guess receives a score of 0, then the colors in that guess are not used in any future guesses.

The Temporel and Kovacs algorithm works according to the following pseudocode:

```
submit a guess to the code maker, and
consider it the Current Favorite
Guess (CFG)
```

```
while the guess is not correct:

  create a new guess by calling
  GENERATE-VALID-GUESS (which is
  detailed in the following section)

  submit that guess to the code
  maker

  if this guess scores at least as
  good as the CFG, then consider
  it the new CFG
```

5.3 The GENERATE-VALID-GUESS Function

`GENERATE-VALID-GUESS` creates guesses by rearranging and probabilistically assigning colors based on the CFG and its score:

```
create a copy of the CFG

for each red the CFG received,
choose an element to keep the
same

for each white the CFG received,
choose a non-red element and
change its location

for each remaining peg,
probabilistically assign a new
color based on the CFG (a
color in the CFG has a higher
probability of being chosen)
```

5.4 Color Probability Calculation

Temporel and Kovacs present a system for modifying color probability based on the current favorite guess and the colors that are already present in the new generated guess after the red and white indices have been picked as follows:

1. Set all color probabilities to 0.

2. Add 100 for all colors in the current favorite guess.

3. Subtract $100 - 45$ for all colors that are in the red list.

4. Subtract all of the probabilities from 100 and set any negative results to 1.

The value 45 was the same value Temporel and Kovacs' paper presents, and after testing, we also confirmed that a value around 45 was optimal. Furthermore, we found that this value works well on larger problem sizes, including 8x8.

6 System Evaluation

Our goal was to improve Temporel and Kovacs' algorithm so that it would, at the very least, solve an 8x8 code in a small number of guesses. In addition to the average number of guesses scored and the internal CPU time required to crack the secret code, two additional metrics are considered:

- *guesses-evaluated* : the number of times potential guesses are checked against all previously scored guesses to see if the commutative requirement is satisfied.

- *guesses-generated* : the number of times a potential guess is generated by mutating the CFG.

6.1 Performance of Temporel and Kovacs' Algorithm

The number of tests run for each problem size is shown below.[6]

[6]Due to the very long time it takes for this algorithm to solve problem sizes beyond 6x8, we could only run a few tests on larger problem sizes for the basic implementation of Temporel and Kovac's algorithm. We tried to run 500 tests for each algorithm detailed throughout this paper on each of the 5 problem sizes (4x6, 5x7, 6x8, 7x8, and 8x8), but after a virtual machine allocated 4 GB RAM and 6 2.8 GHz cores running on Mark Pallone's home computer ran for 40 hours, it then gave a SEGMENTATION FAULT error, corrupting the entire virtual machine and leaving no data. Fortunately, we feel that this still shows how incredibly long it takes Temporel and Kovacs' algorithm to solve anything beyond 6x8 and justifies attempts to improve it.

Note that these numbers are specific to the basic implementation of the algorithm.

	$k = \mathbf{6}$	**7**	**8**
$n_c = \mathbf{4}$	200		
5		200	
6			50
7			70
8			30

Tests Run

The basic implementation of this algorithm works well in terms of the average number of guesses to reach a solution, *guesses-evaluated*, and *guesses-generated*:

	$k = \mathbf{6}$	**7**	**8**
$n_c = \mathbf{4}$	4.59		
5		5.49	
6			5.98
7			6.32
8			7.26

Average Number of Guesses to Reach Solution

	$k = \mathbf{6}$	**7**	**8**
$n_c = \mathbf{4}$	11.19		
5		24.84	
6			28.58
7			32.16
8			67.67

guesses-evaluated

	$k = \mathbf{6}$	**7**	**8**
$n_c = \mathbf{4}$	117.42		
5		1,054.6	
6			4,418.86
7			57,384.36
8			327,141.94

guesses-generated

However, the algorithm became impractical[7] to use as n_c and k grew, due to the very large number of seconds it took, on average, to solve 8x8 codes[8]:

	$k = \mathbf{6}$	**7**	**8**
$n_c = \mathbf{4}$	.04		
5		.37	
6			8.39
7			32.29
8			146.20

Average Seconds Per Solution [9]

6.2 Relaxing the Commutative Requirement

Temporel and Kovacs' algorithm performs a large amount of precomputation before submitting a guess. While this ensures a lower number of guesses on average, it takes a very long time once the search space gets reasonably large—for example, on average, an 8x8 code took 146.20 seconds to solve[10].

We decided that the best way to limit the time our algorithm takes to run, while still utilizing the commutative requirement to produce good guesses (and therefore quick solutions to problems) was to limit the number of guesses we check for commutative validity before submitting a guess. This is necessary because the size of possible-guess-space on the generation side does not shrink, but the number of guesses that are all commutatively valid shrinks dramatically. Thus, the probability of one of the generated guesses falling into the commutatively valid guess list approaches zero with even 10 previous guesses. However, simply submitting the last guess we generated at the time limit would not yield good results since this guess may still be very far from

[7]On one 8x8 test it took 3,600 seconds to find a secret code.

[8]In particular, we were aiming to solve 8x8 secret codes in 20 seconds, as this is a requirement to compete in the Fall 2011 CMSC 471 Mastermind Tournament.

[9]As measured on the UMBC GL Machines in Fall 2011 (`gl.umbc.edu`). These machines have Quad-Core AMD Opteron Processors running at 2.593.646 MHz each, and 13.38 GB RAM (according to their `/proc/cpuinfo` and `/proc/meminfo` files).

[10]Because of the 20 second time limit for the aforementioned CMSC 471 Fall 2011 Mastermind tournament, this large running time would have disqualified us for even 7x8 problems.

optimal, and might even be worse than a guess previously checked for commutativity.

The alternative was to create a heuristic function that can be used to pick the closest matching guess. We created a formula to define this heuristic based on the "distance" from previous guesses. Let d be the distance, bh be the basic heuristic function described in section 4, gr and gw be the reds and whites (respectively) generated by comparing the current guess to a previous guess, and pgr and pgw be the reds and whites (respectively) generated by scoring the previous guess against the secret code. Our formula is then:

$$d = d + |bh(gr, gw) - bh(pgr, pgw)|$$

with the final distance being equivalent to the sum of the distances of each previously scored guess from the current generated guess.

We also attempted to use a weight function based on the actual heuristic of the guess, in order to encourage "hill climbing behavior," but found that this actually hurt our scores, since it devalued some of the information that we received from the low valued guesses.

We next ran the code on various sizes and found the optimal number of maximum commutative checks, as summarized here:

k	**Best Constant**
4	1000
5	1400
6	1900
7	1900
8	2200
9	2200
10	2200
11	200

Note the sharp reduction from $k = 10$ to $k = 11$. Even with this change, problems larger than 10x10 still took a very long time to solve, and so the upper limit for the commutativity test was reduced.

This dramatically reduced the amount of time it took to guess an 8x8 puzzle—going from 146.2 average seconds per guess to only 3.6 average seconds per guess[11]. The average number of guesses to crack the code only went up a small amount: from 7.62 on an 8x8 puzzle to 8.2. While this is a 7.1% increase in the average number of guesses to reach a solution, it is a *97.5% decrease* in the time taken to generate each guess.

[11]Which allowed us to compete in the tournament!

	$k =$ **6**	**7**	**8**
$n_c =$ **4**	200		
5		200	
6			50
7			50
8			50

Tests Run

	$k =$ **6**	**7**	**8**
$n_c =$ **4**	4.675		
5		5.51	
6			6.57
7			7.6
8			8.2

Average Number of Guesses to Reach Solution

	$k =$ **6**	**7**	**8**
$n_c =$ **4**	69.75		
5		456.92	
6			1067.84
7			1526.57
8			1607.65

guesses-evaluated

	$k =$ **6**	**7**	**8**
$n_c =$ **4**	88.49		
5		1159.99	
6			2709.93
7			1257
8			3962

guesses-generated

	$k =$ **6**	**7**	**8**
$n_c =$ **4**	.052		
5		.34	
6			1.038
7			1.76
8			3.61

Average Seconds Per Solution

Since this approach provides a much shorter run time than the basic implementation of Temporel and Kovacs' algorithm, modifications described in sections 6.3 and 6.4 are made to this approach.

6.3 Punishing Colors that Score Poorly

Temporel and Kovacs suggest reducing the probabilities of colors in initial or consecutive guesses that score poorly. We generalized this suggestion by punishing any guess that scores less than 14% of the perfect score for that particular problem size.

The changes were minimal. Very miniscule improvements to the number of guesses guesses to reach a solution and *guesses-evaluated* were observed, whereas a sharp increase in the number of seconds to produce a guess resulted.

	$k =$ **6**	**7**	**8**
$n_c =$ **4**	200		
5		200	
6			100
7			50
8			50

Tests Run

	$k =$ **6**	**7**	**8**
$n_c =$ **4**	4.67		
5		5.4	
6			6.9
7			7.58
8			8.18

Average Number of Guesses to Reach Solution

	$k =$ **6**	**7**	**8**
$n_c =$ **4**	67.78		
5		413.79	
6			1121.79
7			1405.1
8			1495.56

guesses-evaluated

	$k =$ **6**	**7**	**8**
$n_c =$ **4**	90.56		
5		1034.74	
6			3099.71
7			2068.72
8			6408.2

guesses-generated

	$k =$ **6**	**7**	**8**
$n_c =$ **4**	.049		
5		.54	
6			2.91
7			5.00
8			9.32

Average Seconds Per Solution

6.4 Punishing Guesses that do not Hill Climb

While punishing colors that score poorly did not give a large guess number advantage, we felt that punishing scores that did did not hill climb (*i.e.,* did worse by at least 25% than the previous guess) would help the guesses converge faster.

The goal of this "anti-hill climbing" punishing system was to have a behavior similar to genetic algorithms in that guesses close to "bad" guesses are discouraged. However, unlike the color probability changing system presented in 6.3, anti hill climbing punishing does not decrease the search space, it merely attempts to increase the probability of a better guess.

We calculate how close a guess was to the bad guesses by maintaining a bad-guess table, and recording a bad-guess heuristic equivalent to the sum of the basic scoring function of the guess with all of the bad guesses. This gave us an unweighted sum of the closeness of the guess to all the bad guesses, with 0 being the farthest away. We then added this bad guess value, multiplied by a weighting constant (we chose 10%), and added it to the distance value used in part 6.2. While this did help the guess number slightly, the increased cost of the calculation of the bad guess value made our algorithm take many more seconds to arrive at a solution.[12]

[12] We actually omitted this change from the class tournament, as we would have timed out on at least half of

	$k = \mathbf{6}$	**7**	**8**
$n_c = \mathbf{200}$	x		
5		200	
6			100
7			50
8			50

Tests Run

	$k = \mathbf{6}$	**7**	**8**
$n_c = \mathbf{4}$	4.63		
5		5.56	
6			6.77
7			7.56
8			8.64

Average Number of Guesses to Reach Solution

	$k = \mathbf{6}$	**7**	**8**
$n_c = \mathbf{4}$	58.96		
5		380.67	
6			951.06
7			1291.28
8			1647.56

guesses-evaluated

	$k = \mathbf{6}$	**7**	**8**
$n_c = \mathbf{4}$	106.28		
5		672.36	
6			1921.87
7			1271.82
8			5082.82

guesses-generated

	$k = \mathbf{6}$	**7**	**8**
$n_c = \mathbf{4}$	.071		
5		.72	
6			4.03
7			9.11
8			20.92

Average Seconds Per Solution

7 Conclusion

Despite our improvements, the algorithm still took a very long time to solve problem sizes greater than 8x8.

The strength and weakness of this algorithm is the commutative requirement. Satisfying it, while expensive in terms of time, yields very good guesses.

Future approaches may wish to find quicker ways to satisfy this requirement. For example, a systematic, rather than stochastic, approach to generating guesses based off of the current favorite guess may yield a guess satisfying the commutativity requirement more quickly.

the guesses beyond 8x8.