A heuristic hill climbing algorithm for Mastermind

Alexandre Temporel
Department of Computer Science
University of Bristol
a_temporel@yahoo.fr

Tim Kovacs
Department of Computer Science
University of Bristol
kovacs@cs.bris.ac.uk

Abstract

The game of Mastermind is a constraint optimisation problem. There are two aspects which seem interesting to minimise. The first is the number of guesses needed to discover the secret combination and the second is how many combinations (potential guesses) we evaluate but do not use as guesses. This paper presents a new search algorithm for mastermind which combines hill climbing and heuristics. It makes a similar number of guesses to the two known genetic algorithm-based methods, but is more efficient in terms of the number of combinations evaluated. It may be applicable to related constraint optimisation problems.

1 Introduction

Mastermind (MM) is a two player game of logic and deduction, which was invented in 1970-71 by Mordecai Meirowitz [1]. One player is the code maker, and selects a combination consisting of four pegs, each of which can be either red, yellow, blue, green, black or white. All four positions must contain a peg, no blanks are allowed and a colour can be used one or any number of times in a combination. The other player is called the code breaker and has to find the combination, taking as few guesses as possible. After each guess is made, the code maker scores it with small score pegs. Black pegs are given for each coloured peg which has the correct colour and correct position. White pegs are given for each coloured peg which has a correct colour but is out of position. If none of the coloured pegs in a guess matches any of the coloured pegs in the combination, then no score pegs are given. The game stops when the code breaker finds the combination, the goal being to find the secret combination in as few guesses as possible.

There are two implicit rules in the original game of MM. Each response peg refers to one and only one coloured peg [2]. Evaluation of exact matches has precedence over evaluation of near matches. Let n_i be the number of times colour i is in the code, and m_i be the number of times colour i is in the guess. Colour i is matched min (n_i m_i) times. The total number of colour matches is the sum of matches for all individual colours. Since exact matches are evaluated first, the number of near matches can be calculated by subtracting the number of exact matches from the total number of colour matches.

We can define games with arbitrary numbers of colours and pegs. A famous variant of the original game with 8 colours (N=8) and five positions (P=5) is known as Super Mastermind. The number of possible codes in the search space is equal to the number of colours to the exponent of the number of pegs. We can encode the colours of any combinations in a generalised game of Mastermind using strings of integers where each digit represents one colour. The length of the string of integers is equal to the number of pegs the game contains (i.e. for standard Mastermind: four pegs and six colours e.g. 6311, 4463, etc). Assuming there are N colours then each peg represents one of the "digits" 1,2,...N. If there are P pegs then a set of colour pegs represents a unique integer with exactly P digits in base N. We will indicate the score given to a submitted combination with a notation representing the number of black and white pegs between squared brackets. For example in standard MM, a combination which is not far from the secret combination would score one black peg and two white pegs, denoted [2,1].

1.1 Strategies to solve the game

With every judiciously chosen guess and a consistent strategy, the set of possible secret combinations shrinks. A human-like strategy to find the secret code would be to first discover the colours contained in the code and reorder them to determine the final solution. This strategy can be implemented in a computer using some heuristics. In strategically optimal methods, combinations which are known to be incorrect are played, with the intention of extracting information about the shape of the hidden combination, or reducing further the remaining search space. For a stepwise optimal strategy, each guess is optimal in the sense that it is possibly the right answer, that is, it is consistent with all guesses made so far. The number of guesses needed will be finite since each guess and its answer rule out a non-zero set of combinations, and each new combination must be extracted from that set. The number of possible combinations is

reduced each step, and then, finally, only one is possible. An important observation that Darby makes in [3] is that in Mastermind, scores are commutative. If we reverse the roles of the secret combination and any guess, the score remains the same. The important conclusion from this observation is that at each step we can score potential guesses against the guesses we have already made. This results in the set of possible guesses being partitioned into 14 groups according to the score they receive. The secret pattern lies in the group that has the same score as the previous score. So a very efficient strategy is to play at each step a combination which is consistent with the constraints given by the previous guesses. To do so we select a combination as a potential new guess, we consider it to be the secret combination and we score it against the previous guesses. If it scores the same as the score of the previous guesses against the secret target combination then we play it. This new guess might be the secret combination or if it is not, it will in the worst case restrict the search space.

Section two of this paper details briefly the use of evolutionary computing techniques which are known to be perform well on Mastermind and its generalised versions. Section three describes our hybrid algorithm. Section four is about optimisation of our algorithm and section five presents some results. We compare our experimental results with previous algorithms and discuss aspects and extensions to the present work in section six. Finally, we give conclusions in section seven. Further details of our work can be found in [10].

2 Previous work

MM has already been studied by many researchers using different approaches (see [10]) including Combinatorics, Game Theory, Information Theory and Evolutionary Computing methods. Genetic Algorithms (GA)s are widely applicable in game-playing and well-suited to the generalised version of MM due to its large search space and imprecise feedback. The key papers using genetic algorithms for MM are by Merelo [4], [5] and Bento [6].

It is possible to see the evolutionary algorithm of Merelo working on the web [7]. It keeps a population of solutions, which are ranked according to their consistency with the scores (black and white pegs) of the guesses already made. When the algorithm finds a solution that is consistent with all guesses, it is played. In [4] and [5] Merelo uses an interesting fitness function to rate a chromosome against the score obtained by the previously submitted guesses. We can calculate the fitness of a potential guess by comparing the number of Black and White pegs between each submitted Trial Score with the score of our potential guess rated against each of the Trial combinations. In other words, this fitness function gives us a measure of how consistent a new potential combination is with the previous combinations' scores. Position_Weight and Colour_Weight are two coefficients which tune the weighting of black and white pegs.

Bento's fitness function [6] is similar but more complicated and consists of two sub-functions. One is used when all colours have been identified, and the other is used until that point.

Finding a good set of coefficients for Merelo's or Bento's fitness functions is difficult as the values of the coefficients need to be tuned to obtain the best performance. There are other free parameters such as population size and selection scheme which affect performance. We will introduce an algorithm which draws on this previous work but is simpler.

3 Proposed algorithm

Our algorithm is derived from Random Mutation Hill Climbing ([8], page 9), adapted to MM. We refer to it as SHC (Stochastic Hill Climber).

1. We submit to the Code maker a random guess constructed with 4 genes that we call the "Current Favourite Guess" (CFG).

2. From the CFG, we induce a new potential code with the method described in section 3.1.

3. If potential code is not consistent with all previous guesses, go to step 2 otherwise submit it as new guess.

4. If submitted guess scores [0,0] then suppress from the pool of colours all the colours present in the last submitted guess. Then find a new random combination (with the new pool of valid colours) consistent with all previous guesses' scores and set this new combination as new CFG and submit it to the code setter.

5. If submitted guess score is **as good as or better** than CFG score according to the heuristic described in section 3.2, then set this guess to be our new CFG and also set the new score as best score.

6. If submitted guess scores [4,0], stop otherwise go to step 2.

Throughout a game we may sometimes unfortunately submit a trial with no peg matching any colours in the secret combination. In this case we always obtain a MM score with 0 black pegs and 0 white pegs. Step 4 of our algorithm is inspired by Merelo [5] and

allows us to speed up the search considerably by setting some colours to be taboo for the remainder of the game if we get a score of [0, 0].

3.1 Induction of new potential guesses

The idea behind our stochastic combination generator is to produce new combinations directly from the guesses (and their respective scores) which have been submitted throughout a game of MM. This method, unlike Bento's, does not use any complex formulas and consequently produces new potential combinations with less computation (less processing power). This new method has other advantages described at the end of this section. Let's take an example and imagine a game where:

- The code to discover is "2413".
- The first guess we submit is "1223" and we get [1,2] as score returned by the code maker. "1233" becomes our CFG. Figure 1 represents this game situation.

Secret Code: 2413

#1: 1233 [1,2]
…

Figure 1: example of a game of MM

The generation of a new potential code with our method consists of three steps:

Step 1: The number of pegs to keep from our first guess/CFG depends on the first figure between square brackets which is equal to one in this example. So in this case we apply step 1 only one time and if the 2nd digit is randomly selected then the peg to keep is "2" in the second position. Our partial potential combination becomes "x2xx".

Step 2: Pegs to shift depends on the second figure between square brackets, which is 2. So we repeat step 2 two times. In this example, let's suppose the 1st and 4th digits are randomly selected, so we shift "1" and "3" to new and unoccupied locations in the partial potential combination and obtain "321x".

Step 3: The 4th digit "3" from the initial combination needs to be mutated by a new digit with a value different from "1", "2" or "3". This peg can be selected randomly or by using a Fitness Proportionate Selection scheme (explained in section 3.4). If the mutated peg becomes, for example "4", we finally obtain a potential combination equal to "3214". If we now submit the combination "3214" to the code maker to be compared with his secret combination, we obtain the score [1,3] which is an improvement compared to the previous score of [1,2]. Using this process, we can generate many different potential combinations including (possibly) the secret code. Advantages of our method are that it does not require:

- any calculation of fitness for the stochastic choice of two individuals/combinations to combine.
- any calculations to work out how many pegs may be in the right place / of the right colour (as in Bento's paper) before doing further operations.
- the need of having a pool of combinations as the new proposed operator is asexual and creates a new individual from one parent. The function can just be called when a new potential combination is needed and can be seen as an online generator.

More importantly, the method allows us to produce new induced combinations that are at least consistent with the combination they have been induced from. When following a consistent guess strategy, this method should consequently be very reliable.

Our method creates new potential combinations from the best known guess (the CFG), which is thought to be close to the secret combination according to the heuristic explained in the next section.

3.2 Number of possible responses and distance to goal heuristic

The standard version of Mastermind allows 14 possible scores (responses) by the code maker. Our method uses a sub-algorithm which treats the scores as a "distance to goal" heuristic. The range of values depends on the number of pegs involved in a game of MM. For standard MM, we saw that our sequence contains 14 values (from 0 to 13) and for superMM 20 (from 0 to 19).

A good heuristic to estimate how far we are from the goal seems to be the following sequence: [0,0] = 0, [0,1] = 1, [1,0] = 2, [0,2] = 3, [1,1] = 4, [2,0] = 5, [0,3] = 6, [1,2] = 7, [2,1] = 8, [3,0] = 9, [0,4] = 10, [1,3] = 11, [2,2] = 12 and [4,0] = 13. This simple heuristic gives the distance to the goal by penalizing the pegs with the right colours but in wrong position. This heuristic follows the idea that if we submit a guess and get a good score, that means that the search space is more reduced than getting a less good score. Indeed, depending on the first guess, our heuristic is generally a good approximation of how far we are from discovering the secret combination. In any sequence of scores on any generalised game of Mastermind, the Inversed Score ID of 0 means that we discovered the secret combination.

The number of possible responses for any generalised version of the game is given by:

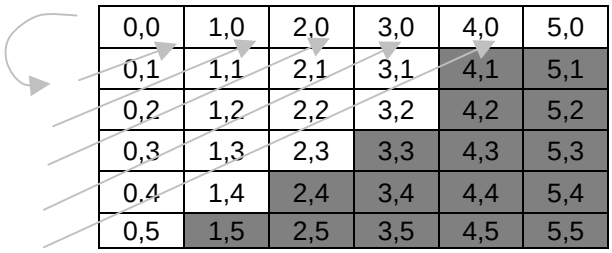
$$((\text{Number\_of\_pegs}) * (\text{Number\_of\_pegs} + 3)) / 2$$

Note that only the number of pegs in a generalised game of Mastermind changes the number of possible responses (it does not depend on the number of

		Guesses		Number of evaluations		Maximum evaluations
	Colour selection type for pegs to mutate	Average number of guesses	s.d.	Average number of guesses	s.d.	
Code tracker "OFF"	Random selection	4.621	0.854	85	441.4	67672
	FPS selection	4.623	0.852	83.4	319	29041
Code tracker "ON"	Random selection	4.64	0.867	66.6	86.3	1021
	FPS selection	4.625	0.854	41.9	50.5	528

Table 2: Performance of our stochastic hill climbing algorithm on 4x6 games

colours). For example in the version of Super Mastermind: 5 pegs / 8 colours, the number of responses using the formula above is 20 (with Nb_of_pegs = 5). The sequence of responses is: [0,0]=0; [0,1]=1; [1,0]=2; [0,2]=3; [1,1]=4; [2,0]=5; [0,3]=6; [1,2]=7; [2,1]=8; [3,0]=9; [0,4]=10; [1,3]=11; [2,2]=12; [3,1]=13; [4,0]=14; [0,5]=15; [1,4]=16; [2,3]=17; [3,2]=18; [5,0]=19 and can be represented in Table 1. Note that the response [4,1] (4 black pegs, 1 white peg) is impossible (the cell with the stripes) as well as all the cases where the addition of the number of black pegs and the number of white pegs is greater than 5 (grey cells).



0,0	1,0	2,0	3,0	4,0	5,0
0,1	1,1	2,1	3,1	4,1	5,1
0,2	1,2	2,2	3,2	4,2	5,2
0,3	1,3	2,3	3,3	4,3	5,3
0,4	1,4	2,4	3,4	4,4	5,4
0,5	1,5	2,5	3,5	4,5	5,5

Table 1: Possible MM responses-5 peg game

3.3 Code tracker

A 'Code Tracker' is simply an array with a number of elements equal to the size of a particular search space. It stores how many times a particular combination in the search space has been examined and eventually considered as a new guess to submit.
Two things can happen once a new potential combination is generated with the method described in section 3.1:
• Either this potential combination, when scored against the previous guesses, does not give the scores previously obtained in the game and is consequently not consistent and not good to play as next guess.
• Or this potential code is consistent with all previous trials and it can be submitted as a new trial (because it might be the secret combination).
So when a particular combination is evaluated to check its consistency against previous trials, if it is not consistent then it can be discarded because there is no need to bring further attention to this particular combination again. Consequently using a code tracker does speed up the search as it avoids combinations which have already been examined.

3.4 Diversity heuristic and Fitness Proportionate Selection (FPS) of pegs to mutate

When using the method described earlier in section 3.1, we saw that the pegs to mutate (in step 3 of the induction of potential combinations) can be chosen randomly and that this technique was a feasible way to select new colours. Table 2 shows the results obtained when the Stochastic Hill Climbing agent is used on the standard version of Mastermind (4x6). Runs consist of a set of 100 000 games and have been carried out with the Code tracker "ON" or "OFF". Also, the type of selection for the pegs to mutate is either random or using a Fitness Proportionate Selection method described in the following paragraphs. A better method than choosing randomly the new colours for the mutated pegs is to pick colours which are different than the ones already present in the partial created combination. This stochastic replacement assures a maximum colour diversity in the successive guesses at each step of a Mastermind game. (Notice the better performance in terms of number of examined evaluations in Table 2 when selecting pegs for mutation using FPS and not random.) Also by choosing colours which are different from the colours already present in the partial potential guess, we make sure that the complete potential guess will be at least consistent with the combination it has been induced from or, in other words, the CFG. This means that if a new potential guess is consistent with the previous trial scores then this potential guess can be played and might be the secret combination. Inducing potential codes from the best known combination (the one which scored the best score throughout the game according to our "distance to the goal" heuristic), allows us to induce very good (fit) potential guesses. Indeed, the possible number of eligible combinations shrinks as we discover fitter and fitter combinations (with better and better scores) throughout a game of MM.

When inducing new potential guesses, the colours chosen for the pegs to mutate should tend to be different from the colours present in the partial potential guesses to create a complete potential guess which is consistent with its CFG. A very small probability for the colours already present in the partial guess must be given because we can't guarantee that the pegs to keep and the ones to shift have been correctly chosen as they are picked randomly (in step 1 and 2 of the method).

Let's consider the incomplete game of Mastermind shown in Figure 2 where the best known combination is "3155" with a best score equal to [2,1]. We are close to the goal with an Inversed Score ID = 2.

Secret Code: 5165

#1: 1122 [1,0] ~ CFG: 1122
#2: 6143 [1,1] ~ CFG: 6143
#3: 6332 [0,1] ~ CFG: 6143
#4: 3155 [2,1] ~ CFG: 3155

Figure 2: new example of a MM game

Here is an example of the generation of a potential guess induced from the latest CFG (the best combination found so far "3155").

Step 1: We choose 2 pegs to be kept. We suppose that the 2nd and 3rd digits have been selected randomly so we keep "1" and "5" in second and third place and copy these colours in the partial induced guess which becomes "x15x".

Step 2: We select 1 peg to be shifted to a new and unoccupied location. We suppose that the 4th digit has been selected randomly and found that the first digit of the partial guess is suitable. We consequently copy the colour "5" to this new location and obtain "515x".

Now for **Step 3**, we need to replace the remaining peg (1st digit) from the initial combination "3" by a different colour which is distinctive from any colours already within the partial generated guess. To do so, we construct a pool of colours attributing to each colour (from 1 to 6) a probability of selection when using a Fitness Proportionate Selection scheme.

Five new steps allow us to create the probability figures and the colour pool which will help to choose new fit colours to improve the induction of new potential guesses. This new sub-algorithm to select new fit colours for our mutated pegs is simple. The new colour of each mutated peg must be:

- different from the colour(s) which need(s) to be mutated
- as distinctive as possible from the colours already present in the partially generated guess

1) All the probabilities for each of the six colours are reset to 0 (CP: 0.0.0.0.0.0). The probability of colour "6" to be chosen is represented by the value on the right hand side of all values in the colour pool.

2) Each colour which is present in the CFG is attributed a positive weight of 100, which means that having as CFG the combination "3155", "3" and "1" are attributed a weight of 100 each and "5" is attributes a weight of 200 (CP: 0.200.0.100.0.100).

3) Each colour in the partial generated guess is attributed a negative weight of 100 which means that we subtract 200 from the weight of colour "5" and 100 from the weight of colour "1". We obtain a new colour pool which finally sets a weight of +100 to colour "3" (CP: 0.0.0.100.0.0).

4) We will see in section 4 that 45 has been found to be close to an optimal value to discriminate the colours already present in the partial generated guess. So now we add a positive weight of 45 to each colour present in the CFG. We then obtain 90 for colour "5" because we have two pegs of this colour and we also add 45 to the weight of colour "1" which is present just once. We obtain the following colour pool (CP: 0.90.0.100.0.45).

5) At this last stage, we subtract the weight of each colour from 100 to reverse the fitness of each colour. We finally obtain the pool of colour (CP: 100.10.100.1.100.55) represented graphically in figure 3 where each vertical bar indicates the percentage probability of picking a colour. During the last process of subtraction, if we had a weight with a value bigger than 100 then we end up with a negative number! In this case, an additional rule is that if the subtraction is equal to or less than 0 then we set the final weight for this colour to 1. This is to allow a minimum probability for all colours to be finally chosen just in case the random digits we picked in step 1 and 2 were not so good. Indeed after generating many new potential guesses, we must be able to find a potential guess which is consistent with all previous guesses! That is why each colour is given a minimum selection probability of 1.

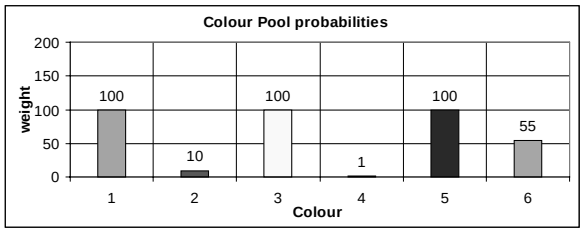


Figure 3: colour pool probabilities

The sum of all fitnesses for each colour is 366 in our example. We now use a roulette wheel to select a colour. We choose a random number between 1 and 366, say, 324, which corresponds to colour “6”. So colour “6” will replace the original colour to mutate (“3”). We can note that colours “1”, “2” and “4” had very strong chances of being chosen but although colour “6” had only 366/55 = 15% chance, it is the colour which we will use to complete our partial potential guess. We obtain the complete potential guess “5156”. If we compare this to all previous combinations, it is consistent with all scores given to our previous guesses and our potential code might be the secret combination.

Submitting this guess gives us a score of [2,2] which means that we have not found the secret combination, although it represents an improvement compared to the previous score of [2,1]. Generating a new potential guess by repeating the process discussed in this section should hopefully uncover the secret combination at our next guess.

4 Optimisation of our algorithm

In our algorithm we have the choice to replace the CFG if the score is “**the same as**” or “**better than**” the best combination’s score so far. The next test aims at discovering which option is best and what value (from 1 and 100) is the best to optimise our Fitness Proportionate Selection parameter.

Figures 4 and 6 were generated from runs where a new combination score as good as the score of the CFG replaces it (option 1). Figures 5 and 7 were generated from the other condition where the CFG is replaced only if a new trial scores better than the CFG (option 2). Figures 4 and 5 show the number of average guesses and their corresponding standard deviation when the parameter to optimise varies from 1 to 100. Each point represents an average over 100000 games. Figures 6 and 7 show the average number of evaluations and their standard deviation when the parameter varies from 1 to 100. Each point represents an average over 100000 games. Figure 8 shows the maximum number of evaluations which occur over 100000 games for each test where the parameter varies from 1 to 100. We can also observe the time taken by the algorithm to complete the run.

Figures 4 and 5 do not really show which option is best. Performance in both is similar and the only conclusion is that performance improves as the parameter increases. However, we can see in figures 6 and 7 that best performance is obtained (in term of minimum size of explored search space) when the parameter is between 40 and 80. Also, we examine fewer evaluations if we change the CFG only when a real improvement and a better score is found throughout a game. In conclusion, tests with Option 1 seem not to perform as well as tests with Option 2.
Now, one can wonder what should be the value of the parameter to optimise between the range 40–80. It is clear when looking at the graph shown on the left hand side of figure 8 that the algorithm becomes less efficient in terms of time to solve a set of 100000 games when the parameter is greater than 45. (Time is shown in seconds along the y-axis.) We can also notice on the same graph that in that region (the parameter is approximately 45), the maximum number of evaluations done over 100000 games is never greater than 560. This means that in the worst case, we don’t look at more 43% of the search space before cracking the secret combination.

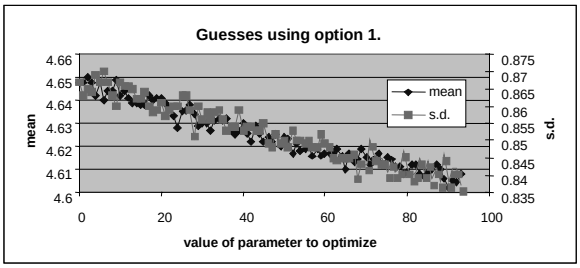


Figure 4

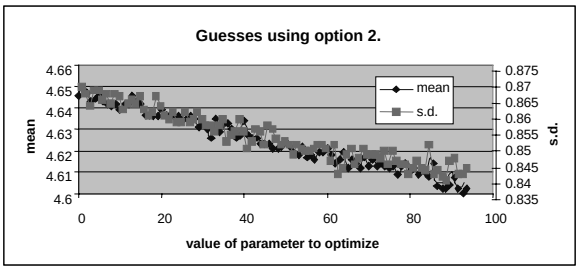


Figure 5

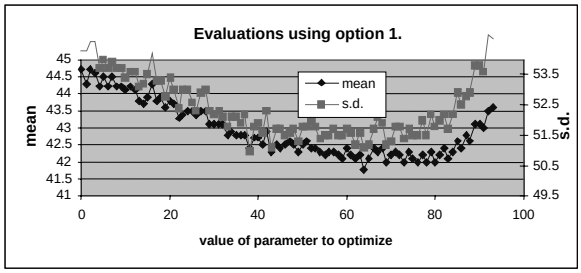


Figure 6

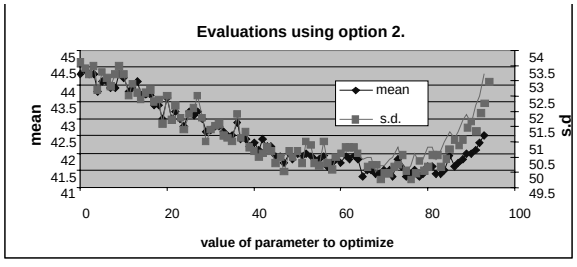


Figure 7

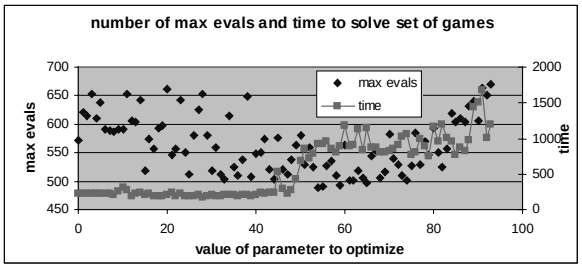


Figure 8

5 Experimental procedures and results

Table 3 shows the average number of guesses and standard deviation of random search (Rosu [9]), Genetic Algorithms (Bento's and Merelo's results) and our Stochastic Hill Climber (SHC) for three sizes of game. Our tests use the same conditions as Merelo's and Rosu's. The first guess is an AABB type of guess and 1000 games are played.

	Number of guesses		
	4x6	**5x8**	**6x8**
Rosu	4.66	5.88	6.36
Bento	-	6.86	-
Merelo	4.132±1.00	5.904±0.975	-
SHC Code tracker OFF	4.661±0.841	5.888±0.949	6.308 ± 0.953
SHC Code tracker ON	4.64±0.857	5.834±0.934	6.289 ± 0.878

Table 3: average guesses and standard deviations

We can observe in Table 3 that our results are consistent with other algorithms. Our average number of guesses is a bit less than the random search but not as good as the performance of Merelo's GA. We can notice that results for all algorithms following a consistent guess strategy (stepwise optimal strategy) are very similar. Table 4 shows the average number of evaluations and standard deviations for the same algorithms as table 3.

	Number of evaluations		
	4x6	**5x8**	**6x8**
Rosu	1 295	32 515	258 000
Bento	-	1029.9	-
Merelo	279 ± 188	2171 ± 1268	-
SHC Code tracker OFF	76.3 ± 136.9	835 ± 1934	2800 ± 5373
SHC code tracker ON	41.2 ± 49.0	480.1 ± 826.6	2759 ± 4728

Table 4: average combinations evaluated and standard deviations

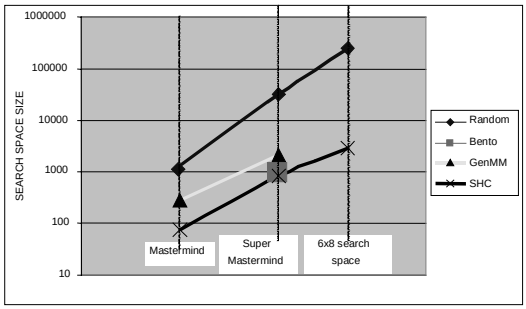


Figure 9

The results obtained by our algorithm are much better than the others in terms of number of examined combinations. Using the code trackers it also has smaller standard deviation than Merelo's algorithm.

We can clearly see in figure 9 (data from table 4) that our Stochastic Hill Climber algorithm outperformed the other algorithms in terms of number of evaluations. Indeed for the standard version of Mastermind, we look at only 4% of the whole search space, 0.014 % of the search space on average for Super Mastermind and 0.01 % of the search space for games with six pegs and eight colours.

6 Discussion / further work

A slight problem in our algorithm is that it has sometimes difficulties to find secret codes when they consist of repeated colours because of our Fitness Proportionate selection which tends to generate new potential combinations with as much diversity as possible in the colour material. It is probably a good idea to keep diversity at an early stage of each game (for the first two trials) but then the parameter setting should adapt itself to a reasonable level of diversity as we progress towards the end of a game. The self adaptation of the FPS parameter could depend on:

• How many guesses have already been submitted.
• How many accumulated black and white pegs we got for the first few guesses.

Another self-adaptive method for the FPS parameter could be to keep track of previous colour pools (for each trial) and then choose new colours depending on the probabilities of the whole set of colour probabilities since the beginning of each game. This method could possibly help us to explore the search space in a more systematic way. Also it might be possible to combine our FPS scheme with a circular mutation scheme (described in [5]).

When looking at the games which take much more evaluations than the others (outliers), one can notice that the first few guesses of each of these games always start with poor scores such as [0,1], [1,0].

Consequently, another heuristic could be added to our algorithm to improve the search. For example, if two consecutive submitted combinations get marked with poor scores then it means we are searching in the wrong part of the search space. In this case, we could trigger an extra heuristic to find a random combination as much different as possible from the colours contained in the first two combinations. We could for example add all the black and white pegs from the successive scores together to get a measure of how many pegs have been discovered so far.

Another heuristic could select a new combination which consists of colours as different as possible from the colours of the first guess if the score attributed to this initial guess is poor (score such as [1,0], [0,1], [1,1], [0,2]). It could also be interesting to try our algorithm on variants of Mastermind such as dynamic Mastermind or Ups and Downs [11] to see how it performs on these problems.

Another idea would be to combine our algorithm with a genetic algorithm where instead of having an initial population consisting of random combinations, we could have populations of combinations induced by our potential combination generator described in 3.1. Mixing stochastically combinations between the different populations could possibly lead to very good performance. Different sizes of population could be used with different recombination operators and different mutation and crossover rates.

7 Conclusion

The presented algorithm is a hybrid combination of heuristics based on a hill climbing search concept. A key point is the way new potential guesses are generated from the best known combinations throughout a game. A new Fitness Proportionate Selection scheme allows us to keep maximum diversity in the colour material at each step of the game. Our algorithm is guaranteed to find the hidden combination in finite time, and it takes sub-exponential time to run. It is efficient in term of memory as nothing but the code tracker array needs to be stored, and this is optional. Although simpler, our algorithm considers fewer potential combinations than two genetic algorithm-based methods to which it was compared. Further improvements on our algorithm are certainly possible and it would be interesting to attempt to apply it to related constraint optimisation problems.

Acknowledgements

We would like to thank Seenu Reddi for his help and advice and Todd Ebert for his interest.

References

[1] A Brief History of the Master MindTM Board Game. http://www.tnelson.demon.co.uk/mastermind/history.html

[2] Alex Bogomolny and Don Greenwell. Invitation to Mastermind , MAA Online Column, CUT THE KNOT, December 1999. http://www.maa.org/editorial/knot/Mastermind.html

[3] Gary Darby, Delphi for fun. www.delphiforfun.org/Programs/MasterMind.htm

[4] J.L. Bernier, C. Ilia Herraiz, J.J. Merelo, S. Olmeda, and A. Prieto. Solving mastermind using GAs and simulated annealing: a case of dynamic constraint optimization. In Parallel Problem Solving from Nature IV, lecture notes in Computer Science 1141, pp. 554-563. Springer-Verlag, 1996.

[5] J. J. Merelo, J. Carpio, P. Castillo, V. M. Rivas, and G. Romero. Finding a needle in a haystack using hints and evolutionary computation: the case of Genetic MasterMind. GeNeura Team. Departamento de Arquitectura y Tecnología de los Computadores Universidad de Granada http://kalel.ugr.es/~jmerelo/newGenMM/newGenMM.html

[6] Luís Bento, Luísa Pereira, and Agostinho Rosa. MASTERMIND by Evolutionary Algorithms http://laseeb.ist.utl.pt/publications/papers/acrosa/sac99/sac99.html

[7] Geneura Team and Genetic Mastermind. http://geneura.ugr.es/~jmerelo/GenMM/GenMM.shtm

[8] Eric B. Baum, Dan Boneh, and Charles Garrett. Where Genetic Algorithms Excel. In Proceedings *COLT 1995*, pp. 230--239, Santa Cruz, California. http://crypto.stanford.edu/~dabo/abstracts/ga.html

[9] Radu Rosu. Analysis of the game of Mastermind - the m^n case. Undergraduate thesis, North Carolina State University, 1997. http://www.csc.ncsu.edu/degrees/undergrad/Reports/rtrosu/index.html

[10] Alexandre Temporel. Stochastic Search Methods for Mastermind. MSc Machine Learning thesis. Dept. of Computer Science, University of Bristol, 2002.

[11] Seenu Reddi. Mathmind – variations on mastermind. Unpublished paper, 2002. http://64.224.226.20/pubs/mm.pdf