StrangeMind:
A Genetic Algorithm for a Time-Constrained Scalable Mastermind Player

By

Andres Quinones, Nishad Sharker,

Iden Watanabe, and Gordon Zheng

CSCI 350

Professor Susan Epstein

1 Introduction

Many papers have been written regarding Mastermind. The game in its original format of four pegs and six colors is trivial to solve, but variations of Mastermind grow increasingly difficult as the number of pegs and colors increases. Most papers seem to be concerned with two things: solving the game in the shortest amount of steps, and limiting the search space required to reach an answer. Though none of them address the specifics of our problem (the pattern maker and time limit), we felt that they were an appropriate starting point for the framework of our player.

1.1 Literature Review

Rao's algorithm treats Mastermind as a type of constraint satisfaction problem. Starting with a guess of a single color, it then goes through each possible color in order until the code maker gives positive feedback with a combination of black and white pegs. It begins using that color from the leftmost position of the board, and moving it to the right on each successive guess until the correct location is discovered, all the while still trying to find the next correct color among the remaining pool of colors [3].

Another algorithm considered was Knuth's classic Mastermind algorithm. Knuth generates a list of all possible codes and maintains it throughout a game. It starts off with a set guess (1122) in a four peg game. Once feedback is given, Knuth's algorithm prunes from the set of potential answers any codes which would not have resulted in the same feedback as the last guess. From the set of all possible guesses, the algorithm searches for the guess, which when played against each code in the set of possible answers, would in the worst case eliminate the most codes from the set of possible answers [2].

A more complex algorithm was Temporel and Kovacs's heuristic hill climber model. Rather than generate the entire search space at the start like Knuth's, the hill climber started with a simple, random guess. Utilizing a heuristic based off of a score consisting of the black and white pegs, where black pegs were weighted higher, the algorithm would generate a new guess as follows: for the number of black pegs, the algorithm would randomly select that number of

pegs to keep for the next guess, for the number of white pegs, the algorithm would select that number of pegs to swap positions on the board, and any remaining pegs would then be mutated according to a specific algorithm. That guess would then be checked against the previous highest scoring guess to ensure consistency. If the new guess scored at least as well as the best guess, it would be played against the actual code maker and the process would repeat [5].

Finally, we considered a genetic algorithm designed by Berghman et al. As with the previous algorithms mentioned, a fixed guess is played first. Offspring are created from parents utilizing 1 or 2-point crossover and further mutated using mutation, inversion, and permutation. A code is considered eligible if it is consistent with every previous guess and response. Each guess involves accumulating a set of eligible codes over many generations. This is done until either the set of eligible combinations reaches a certain size, or until a certain number of populations have been created. Lastly, the algorithm selects a guess from the pool of eligible combinations that will hopefully restrict the search space the most, then uses that as its next submission. The algorithm repeats until an answer is found [1].

1.2 Algorithm selection

Though we knew Rao's algorithm would not scale well, we were initially reluctant to drop it entirely as it seemed to be one of the few algorithms which did not involve full enumeration of the search space. In contrast, Rao's algorithm is methodical and locks down the position and color of each peg. We felt that Rao's approach would be more likely to find an answer, instead of coming up with guesses at random. However, the iterative process would not scale well for our purposes as it can verify only one position at a time.

The hill climber was another algorithm that was inevitably abandoned. The scoring heuristic, while explained well in the paper, proved difficult to scale as was required in the abstract. Still, it had been considered because, as with Rao's, it did not require a whole search space to be generated at the beginning, but rather implicitly created and reduced it with each successive guess [5].

Knuth's algorithm was our first implementation of a successful Mastermind player. However it suffered scalability problems as well. Since the algorithm enumerates and evaluates

the entire search space in memory, it would require $colors^{pegs}$ space and $(colors^{pegs})^2$ comparisons. Knuth was very strong at the original 4,6 game but saw steep performance drops beyond that size.

We settled on exploring a genetic algorithm approach because it would require significantly less space and computation as the game scales up. A genetic algorithm does not need to fully enumerate the solution space. The amount of computation is mostly determined by its population size. Berghman et al. claim that their GA can solve an (8, 12) game in an average time of 20.571 seconds [1]. Though the time is beyond the bounds of our given problem, we felt confident that, given knowledge of the secret-code selection algorithm, we could win games within the time constraint.

2 Implementation

Our genetic algorithm is heavily influenced by the work of Berghman et al. Our implementation of the algorithm they described was not able to reproduce the results they claimed for more difficult games, so several modifications were made. Building on their algorithm, we introduced new concepts -- notably: response pruning and SCSA consistency.

We took an iterative, experimental approach to building our player. For each major change, we measured performance by benchmarking it against previous iterations and the baseline Knuth's algorithm. This allowed us to validate our assumptions and understand the impact of our changes. The following describes the six iterations of our player leading to its final version.

1. Russell and Norvig's basic genetic algorithm
2. Adding new mutation techniques
3. Adding multiple generations per guess and eligibility restriction (Berghman et al.)
4. Replacing eligibility restriction by most fit
5. Adding SCSA consistency as a fitness factor
6. Pruning responses

2.1 Russell and Norvig's basic genetic algorithm

Initially, we implemented a simple genetic algorithm described by Russell and Norvig [4]. It used 1-point crossover for reproduction and, with some probability, mutated offspring by setting one peg in the individual to a random color. It gave a fixed initial guess which included as many colors as possible. To determine its next guess, a new population was bred from the previous, and the most fit individual was selected from the new population.

The fitness function prefers codes which are *consistent* with previous guesses and responses. A code c is consistent with a previous guess g, and g's corresponding response r, if c played against g returns the same r.

Consistency is defined as:

x is the current turn in a game.

g is a guess in a game.

a is an answer in a game.

r is a response in a game.

$r_i = P(g_i,a)$ is the way a response of black and white pegs is generated from a given guess

$G = \{g_1, g_2, g_3, \ldots, g_x\}$ is the set of guesses we received over the course of a game

$R = \{r_1, r_2, r_3, \ldots, r_x\}$ is the set of responses we received over the course of a game

$E(r_1, r_2) = \{$ 1 if r_1 equal r_2 , otherwise 0 $\}$ is a comparator for responses

We can calculate the consistency score of a candidate guess, g_c as

$$S(g_c) = \sum_{i=1}^{x} E(P(g_c, G_i), R_i)$$

In other words, we determine the fitness of a code by counting the number of guesses and responses it is consistent with. For the original 4,6 variation of Mastermind, this initial iteration performed very well, and was nearly on-par with Knuth's algorithm. However, for more difficult variations such as those with 5 or more pegs, it did not scale well. Increasing the population size from 100 to 1000 helped, but only by a little.

2.2 Adding new mutation techniques

In this iteration we added 2-point crossover as an additional reproduction technique, along with inversion and permutation as additional mutation techniques, as described by

Berghman et al. [1]. We believed that different kinds of mutations might allow us to generate a more diverse population which would be useful in finding the answers for larger games.

2-point crossover creates a child from parents p1 and p2 by replacing a subsequence of either parent (chosen with equal probability) between randomly chosen indices a and b where a < b: p1[a..b] = p2[a..b]. Per Berghman et al.'s paper, we perform 1-point crossover and 2-point crossover with equal probability [1]. Permutation picks two random positions in a child and swaps the colors at those two positions, while inversion picks two random positions, then reverses the sequence between those two points.

Each of the mutation techniques has a small chance of being applied to a new individual. The introduction of these mutations alone did not significantly improve the player's performance.

2.3 Adding multiple generations per guess and eligibility restriction

The algorithm by Berghman et al. is focused on searching for eligible codes. A code is deemed 'eligible' if it is consistent with all previous guesses and responses. To find these codes, the algorithm keeps breeding new generations until enough eligible codes have been found, or until the generation limit has been reached. Then, it uses a minimax approach (similar to Knuth's algorithm) to select an eligible code whose response would eliminate most other such codes, because its response would give more useful information.

The idea of creating multiple generations per turn was interesting, as it would allow our player to refine its search pool and hopefully yield more fit individuals. We also found promising the idea of searching for individuals which are consistent with all previous guesses and responses, then selecting the eligible code which would eliminate the most other eligible codes. This would lead to guesses which should more effectively constrain the search space.

While the performance for 4,6 was impressive, it did not perform as anticipated for larger games. At 6 pegs, this algorithm was not able to provide a guess after about 4 or 5 turns. We would later realize that finding eligible codes is more computationally intensive than we thought.

We tried to alleviate this problem in many ways, including raising the maximum number of generations, adjusting the parameters we created in our previous iterations, and by making the

eligibility condition more lenient by allowing a guess that was consistent with at least 90% or 50% (we tried both) of the previous guesses.

2.4 Replacing eligibility restriction by most fit

Though our player was as close to what we thought Berghman et al. described, we were unable to replicate their results for 8,10. Given the difficulty of finding eligible codes, we abandoned the concept and instead picked the most fit individual encountered over g generations. This led to a significant improvement.

We also changed our fitness function to the one described by Berghman et al. Rather than counting the number of guess-response pairs the individual is consistent with, we take the difference in bulls and cows for each previous guess and response [1]. This allows us to evaluate individuals on a more granular level. Instead of measuring the consistency of an individual, we measure its similarity to previous guesses and responses.

The similarity score for an individual is defined as:

p pegs

i number of turns so far

b_i number of bulls in the actual response from turn i

c_i number of cows in the actual response from turn i

b'_i number of bulls in expected response from turn i if the individual was the secret code

c'_i number of cows in expected response from turn i if the individual was the secret code

$$similarity = p * i - \sum_{x=1}^{i} \left| b_x - b'_x \right| - \sum_{x=1}^{i} |c_x - c'_x|$$

2.5 Adding SCSA consistency as a fitness factor

It is very useful to know the SCSA used to generate the secret code for a game as it allows us to rule out codes which do not follow the SCSA's pattern. To incorporate knowledge of the SCSA into our player, we modified our fitness function to prefer individuals which are likely to have been generated by a given secret code selection algorithm.

First we calculate how close the potential guess is to the provided SCSA. For most SCSAs either a 1 is returned if the potential guess matches, or a 0 is returned if the potential

guess does not match. However, some patterns rely on selecting colors based on probability (e.g. prefer-fewer, usually-fewer). The checker functions for these SCSAs return the probability of the potential guesses' number of colors. In order for the SCSA consistency score to scale with the similarity score as the board size and number of turns increases, the confidence is multiplied by the board size, the number of guesses, and the SCSA consistency multiplier. The consistency multiplier allows the weight of the SCSA consistency score to be adjusted relative to its similarity score.

The SCSA consistency score is given by:

M SCSA consistency multiplier

h SCSA confidence (0 <= h <= 1)

$$SCSAConsistency = p * i * M * h$$

Finally, the fitness is determined by adding the similarity score and SCSA consistency score

$$fitness = similarity + SCSAConsistency$$

2.5a Checking for SCSA consistency

Out of the eight known SCSAs, insert-colors is the only one that does not give the player an advantage. The other seven known SCSAs provide information. Checker functions were created using patterns observed from the known SCSA functions.

The checker function for the two-color, ab-color, and two-color-alternating SCSA functions were straightforward to define. A checker function for two-color was implemented by counting the number of unique colors in a guess. If there are only two, return true.

The implementation of the SCSA checker function for AB-color was similar. It counts the number of unique colors for a guess and checks if A and B are in the guess. If the number of unique colors is equal to two, and A and B are both in the guess, then it returns true.

The two-color-alternating SCSA checker function was one of the easier SCSAs to match. First the checker function creates a pattern that repeats 0 and 1 for the length of the potential guess. Then the checker function converts the potential guess from unique colors to numbers. The first unique color in the potential guess is mapped to 0, the next unique color will be mapped

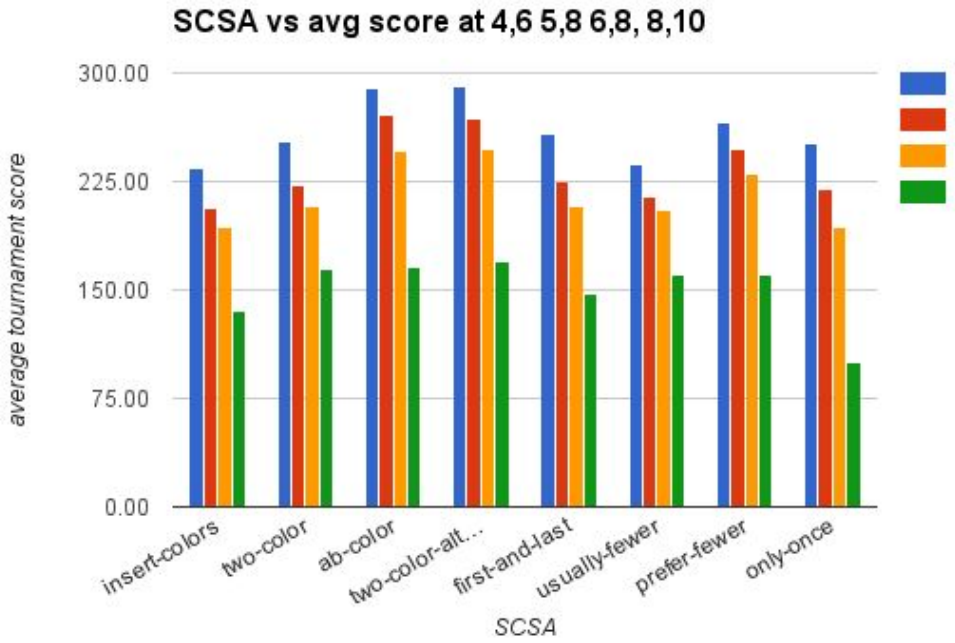
to 1, and so on until all the colors in the potential guess are mapped. For example, (C C D E A G A C B) is mapped to (0 0 1 2 3 4 3 0 5). If our mapped guess matches the original pattern created, then the potential guess satisfies the two-color-alternating SCSA.

The checker function for only-once counts the number of unique colors in the guess. If that equals the length of our guess, then it returns true.

The checker function for first-and-last checks the following condition: if the first peg in the potential guess is equivalent to the last peg. If the condition holds true, then the potential guess matches the pattern of the first-and-last SCSA.

The checker function for usually-fewer is one of the more interesting SCSA cases because it generates secret codes based on a probability distribution of the number of colors. The checker function returns the probability for that color combination to occur, which is used in the fitness function to calculate the SCSA consistency bonus for the potential guess. Usually-fewer has the following probability distribution: 90% chance for using either 2 or 3 colors and 10% chance for any other number of colors. The checker function for usually-fewer calls remove-duplicates on the potential guess, then checks that the length of the truncated guess is equivalent to 2 or 3. If this condition holds true, the checker function returns 0.9, otherwise returns 0.1.

The checker function for prefer-fewer follows a similar implementation to that of usually-fewer where it returns the probability for the number of colors in the potential guess. The prefer-fewer SCSA probability distribution is the following: 50% chance for the code to have one color, 25% chance for two colors, 13% chance for three colors, 8% chance for four colors, 3% chance for five colors and 1% chance for six colors.



Legend: Blue: 4,6. Red: 5,8. Orange: 6,8. Green: 8,10.

Insert-colors can be used as a baseline since it provides no helpful information. Comparing the known SCSA's to insert-color shows how the knowledge of each SCSA allows the program to score higher. Only-once is the only SCSA that performs worse than insert-colors, doing so at game size 8, 10. We speculate this discrepancy is because the nature of genetic algorithms do not produce candidates with a unique set of colors often.

2.5b Mystery SCSAs

Three out of the five mystery SCSAs have an easily recognizable pattern that their respective checker functions could exploit. Mystery-1 follows a repeating pattern of three colors in the same order. The checker for mystery-1 applies a similar method used for two-color-alternating. It creates a pattern of 0, 1, and 2 repeated for the length of the board, and transforms the potential guess to use a similar number format. If the transformed guess is equivalent to the mystery-1 pattern, then the potential guess matches the pattern for mystery-1. The mystery-4 sample codes highly resembled the two-color-alternating SCSA patterns, therefore the checker function for mystery-4 used the same approach as the checker function for two-color-alternating.

The Mystery-3 SCSA is also an easily recognizable pattern, however implementing the code required a unique approach. The sample secret codes of mystery-3 seems to have an even distribution of three colors, therefore its checker function first checks if there are only three colors in the potential guess. If true, then it checks three cases that are based on the size of the board. In the first case, if the board size is divisible by three, then all three colors in the potential guess should be equivalent and the potential guess is equivalent to mystery-3. The second case occurs when the board size modulo three returns one, which indicates that two of the colors appear an equal number of times, and the third color appears one extra time. In this case the checker function creates a sorted list of the number of colors in ascending order, then checks the following two conditions: if the first and second element of the list are equal, and if the third element is equal to the first element plus one. If both conditions return true, then the potential guess matches mystery-3. The final case occurs when the board size modulo three returns two, which indicates that two of the colors appear an equal number of times and one color appears one less time. The checker function creates a sorted list of the number of appearances of each color in ascending order, then the checker function checks if the second and third element are equal, and if the third element and first element plus one are equal. If both conditions return true, then the potential guess matches the mystery-3 SCSA pattern.

The mystery-2 and mystery-5 SCSAs had no immediately visible pattern from the sample secret codes provided. A program was written to parse the sample secret codes of each mystery SCSA and returned the frequency of the number of colors used for each SCSA. The results for mystery-2 SCSA indicated that 45% of the sample codes had only one color, 39% of the sample codes had two colors, 4% had three colors, 8% had four colors, and 4% had five colors. Mystery-5 SCSA had the following distribution: 6% of the codes had two colors, 45% had three colors, and 49% had four colors. The respective SCSA checker functions use these percentages when checking for mystery-2 and mystery-5 in the same way that the checker function for prefer-fewer SCSA uses the prefer-fewer probability distribution.

2.5c SCSA Prediction

We originally believed that the SCSA name would not be given and that our player would have to determine which one it was up against. Our original plan was to count the number of answers so far in a tournament that matched the different SCSA patterns. To calculate the confidence score for each SCSA, the program would divide the number of hits for this pattern by the total number of SCSA pattern hits. However, since we found out that the SCSA name would be provided to our player, we abandoned this system.

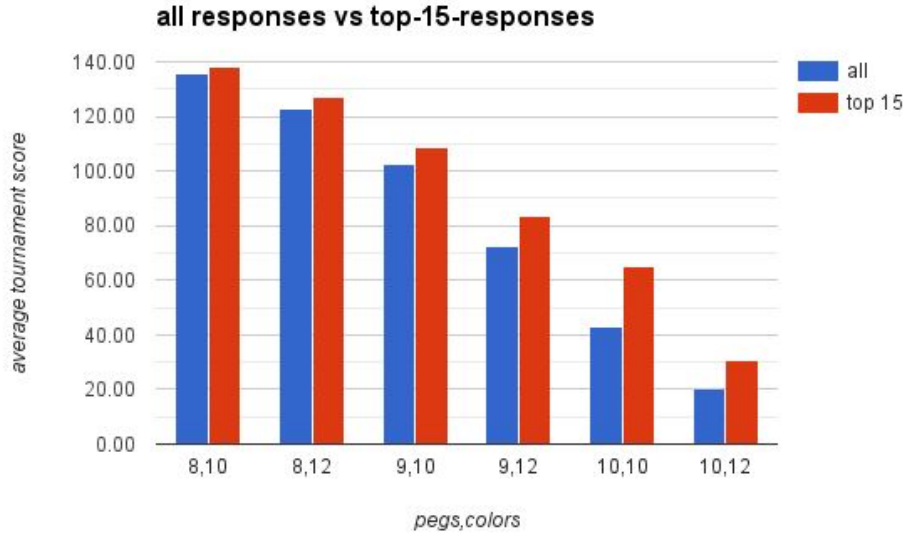
2.6 Pruning responses

In our experiments we observed that the time for a guess increases as we get further into the game. Each successive guess requires more computation than the one before it, since the cost of evaluating the fitness of an individual increases as there are more guesses and responses to compare the individual against.

We set an upper bound on the comparisons we were willing to do by only considering the top n most desirable responses. We define the desirability of a response by the number of bulls and cows, with a preference for bulls.

$$desirability = 1.5b + c$$

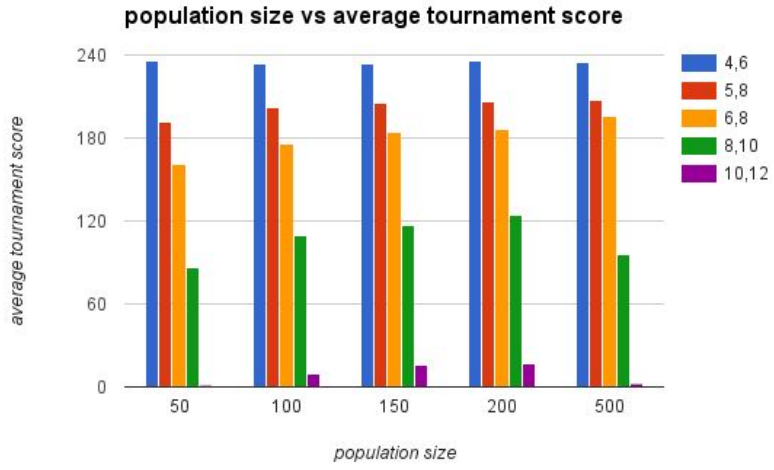
This prevents the time per guess from growing past a given point. Our player has this set at 15. Performance improved for larger games and did not significantly change for smaller games.



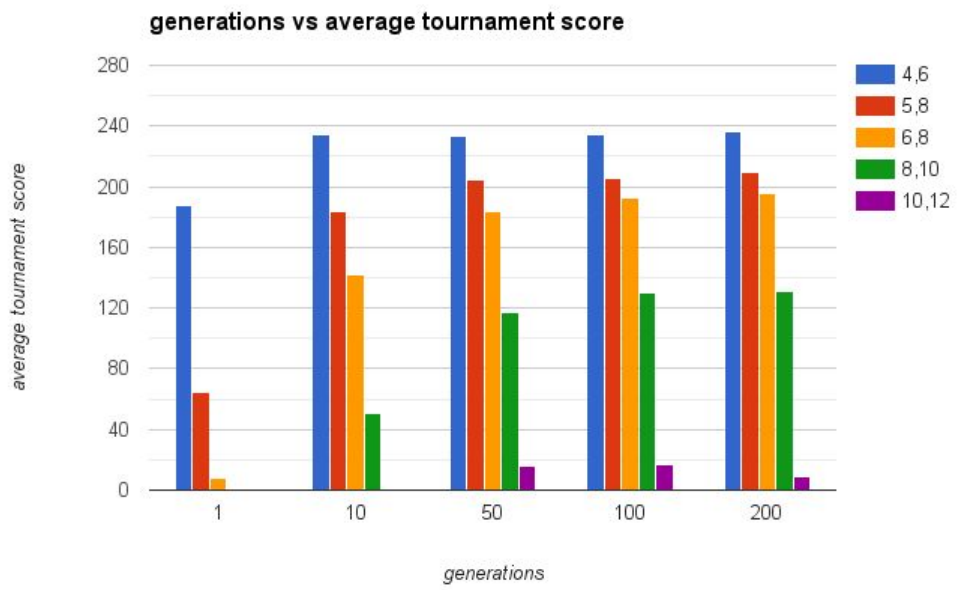
2.7 Optimizing parameters

We wanted to understand the relationship between the genetic algorithm's parameters and the player's performance. To this end, we selected a baseline player and set each variable to a range of values while holding all other variables constant.

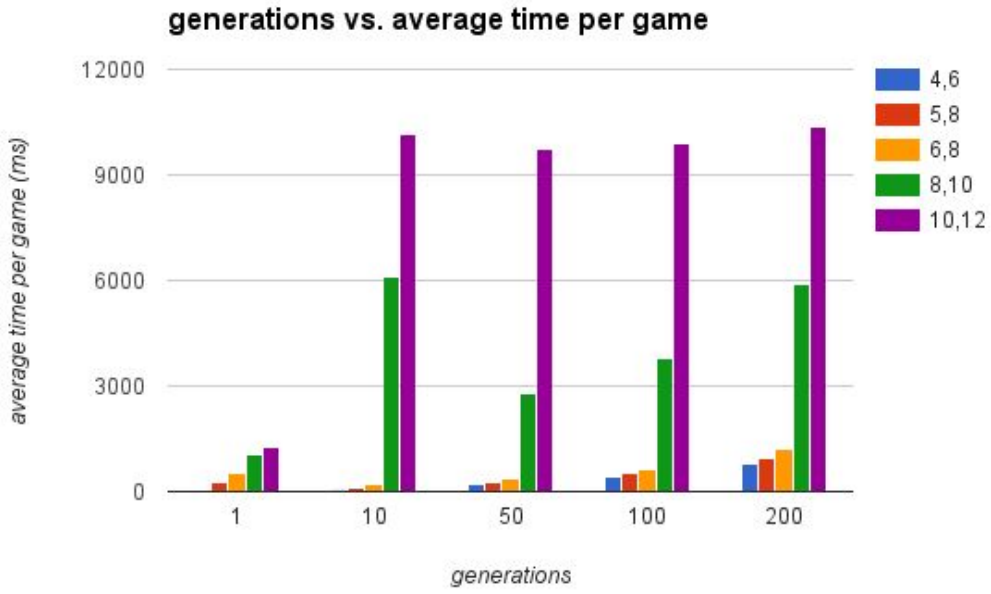
We observed that in general, a higher population size earned a higher score with higher average game times. If the population was too high, we timed out for some games, which negatively impacted the score. We found 200 to be a good balance between wanting to search large swaths of the search space for higher quality guesses, and needing to beat the game in under 10 seconds.



We held population size constant at 150 and observed the performance of the player as the number of generations increased. For 4,6 1 generation fares decently, but for larger games having multiple generations is key to making good guesses. This is easy to understand, as our candidate guesses become more refined with every successive generation. We observed little improvement in score as the generations increased from 100 to 200, despite response times increasing proportionally. Thus, we chose to use 100 generations per guess.

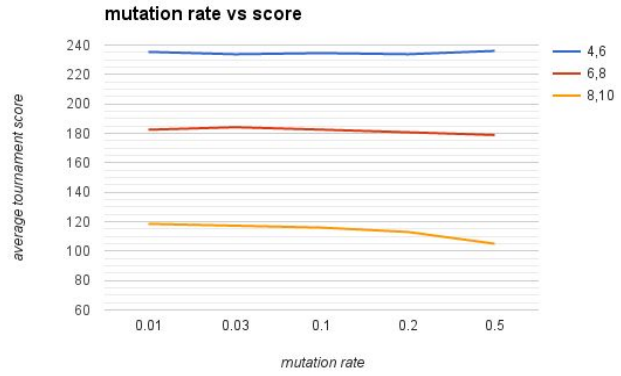


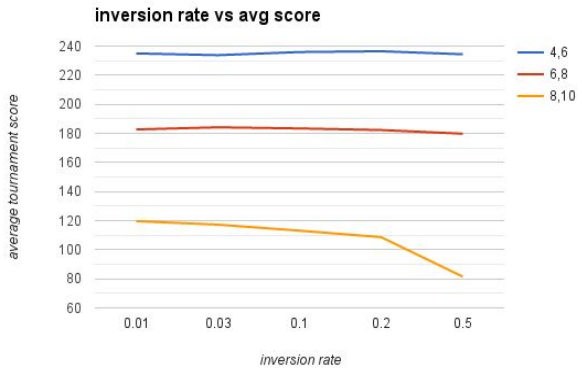
The number of generations per turn influences the average time per game. We observed that for lower numbers of generations guesses were computed faster, but the increase of speed came with lower scores because it took more guesses to arrive at the answer. On the other hand, if the number of generations was too high.

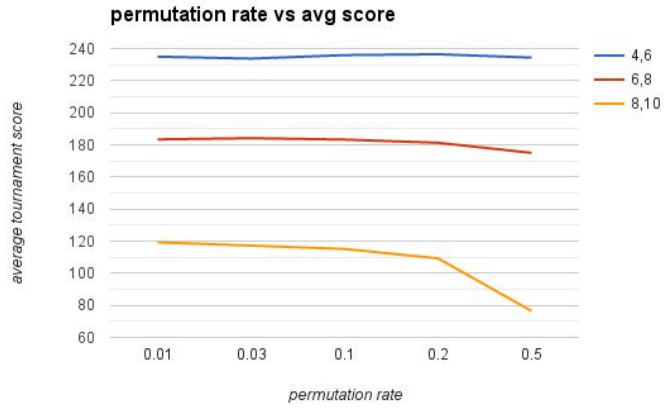


We found that higher mutation rates led to worse performance. The impact was low for smaller games but higher for larger games. We found performance to be best overall when

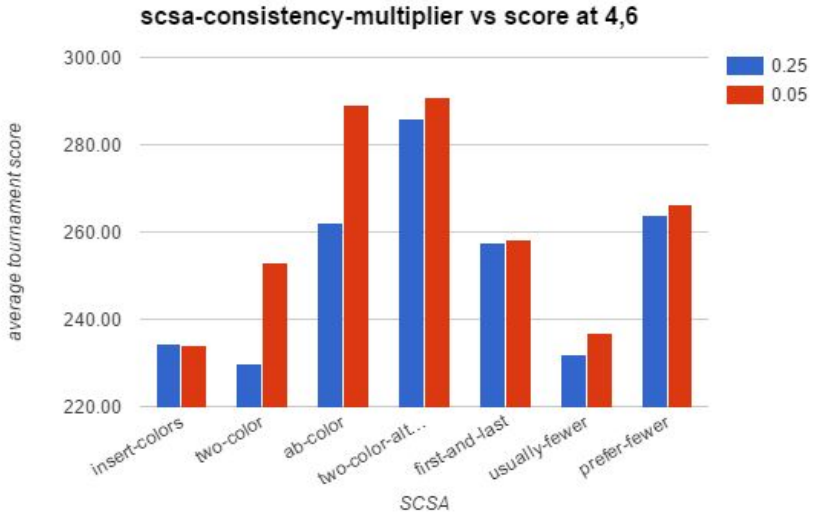
mutation, inversion, and permutation rates were set to lower values, thus we set these rates to 0.01.



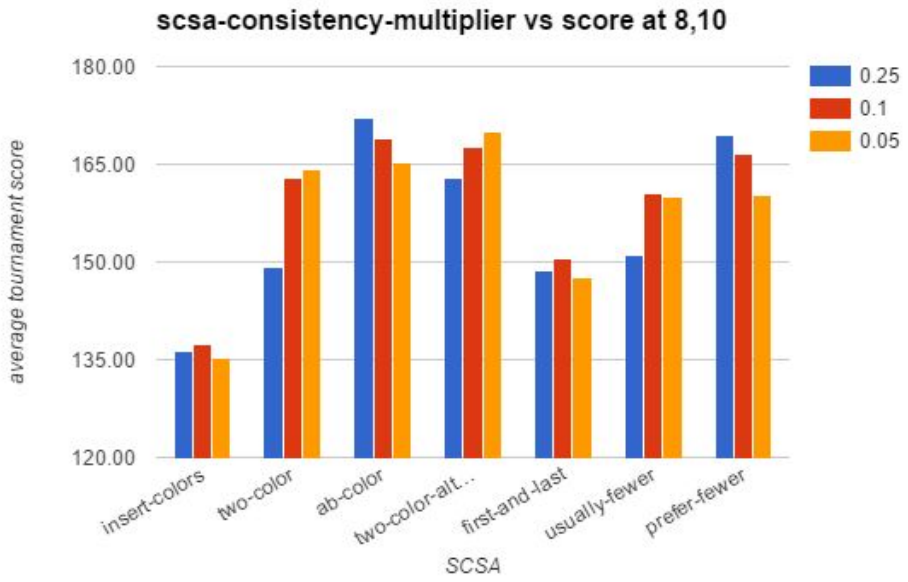




We ran experiments to determine the best scsa-consistency-multiplier to use and discovered that lower values resulted in higher scores at 4,6 for most of the SCSAs.



We also noted that larger values resulted in slightly higher scores for 8,10 but chose not to jeopardize performance at smaller games for performance at larger games and opted for a multiplier value of 0.05. Ideally, we would set this relative to the game size.



2.8 Development practices

Throughout the course of this project, our team adopted techniques which allowed us to be more productive.

We wrote over 100 unit tests covering as much code as possible. In doing so, we naturally decomposed complex functions into many small, simple ones to make them easier to test. This allowed us to make changes with confidence and minimize the risk of hidden side-effects of code changes.

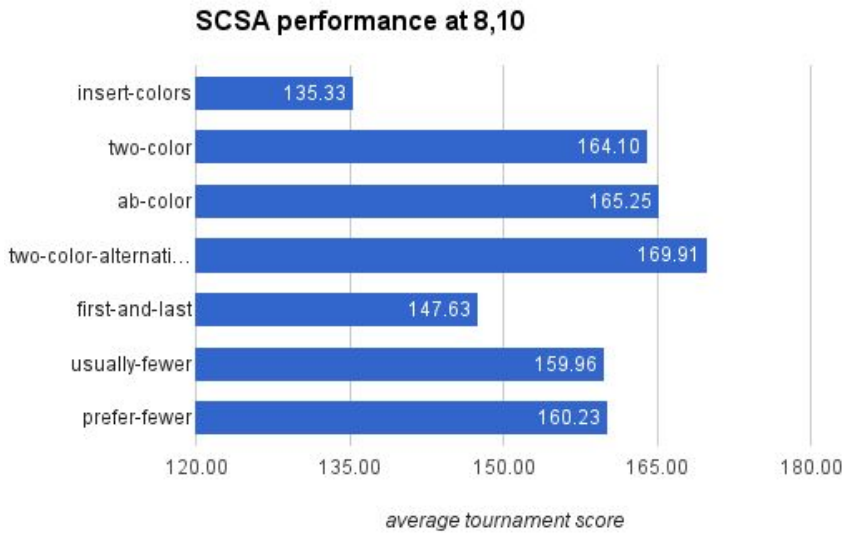
Early on, we developed a benchmarking suite to measure the performance of a team against different configurations of pegs, colors, scsas, and teams. This allowed us to take a data driven, iterative approach to developing our player. We leveraged Travis CI, a continuous integration tool, to run tests and benchmarks whenever a change was committed. During the optimization phase of our final player, we used remote, virtualized servers to distribute the work of running long benchmarks over several days.

We used Git for version control and GitHub for collaboration. The team adopted a style guide to enforce code style consistency.

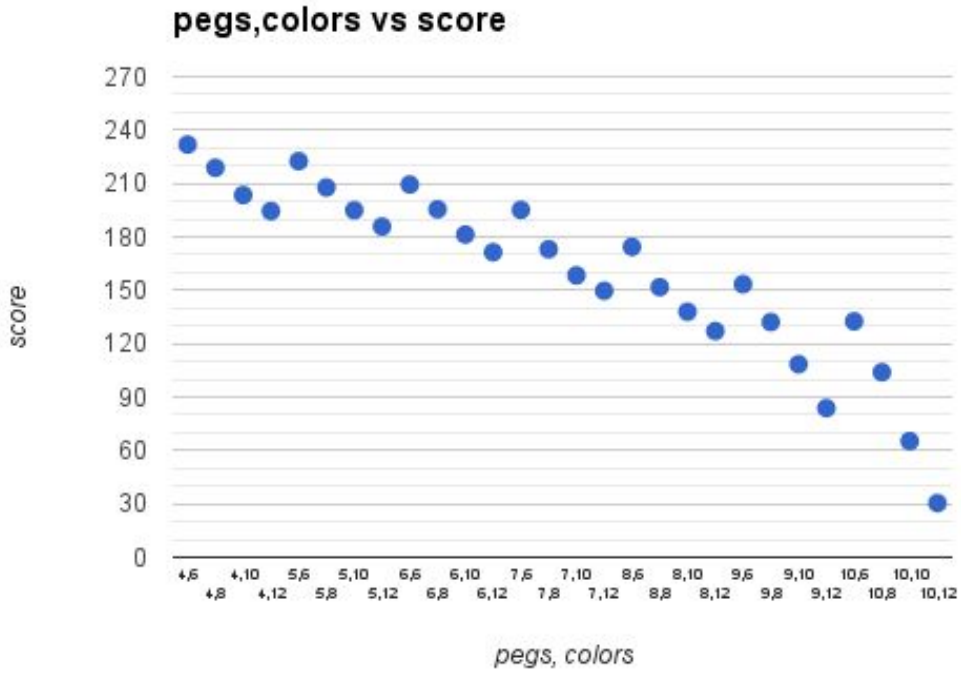
3 Evaluation

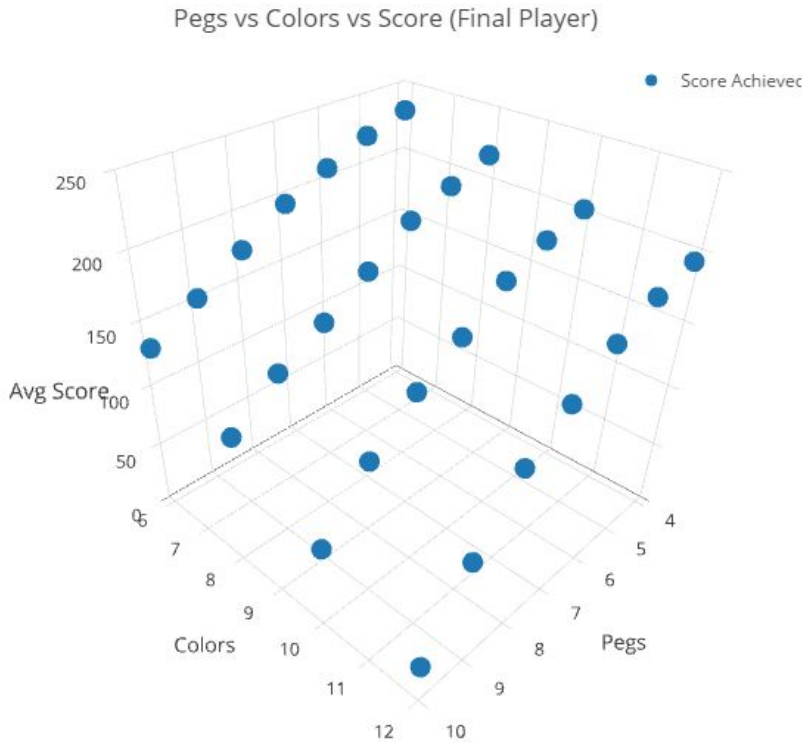
We chose to run benchmarks using trials of 500 games each. We observed the lowest standard deviation with 500 games as opposed to 100 or 200 games. Benchmarks with more than 500 games were avoided because of time restrictions. The worst case time for a 500 game benchmark is 500 games * 10 seconds per game max = 1.38 hours. Some of our benchmarks ran as many as 28 tournaments at a time.

The chart below shows the average tournament score of our player for the 8,10 game across a variety of SCSAs. It is clear that knowing the SCSA allows the genetic algorithm to find the answer using fewer guesses.

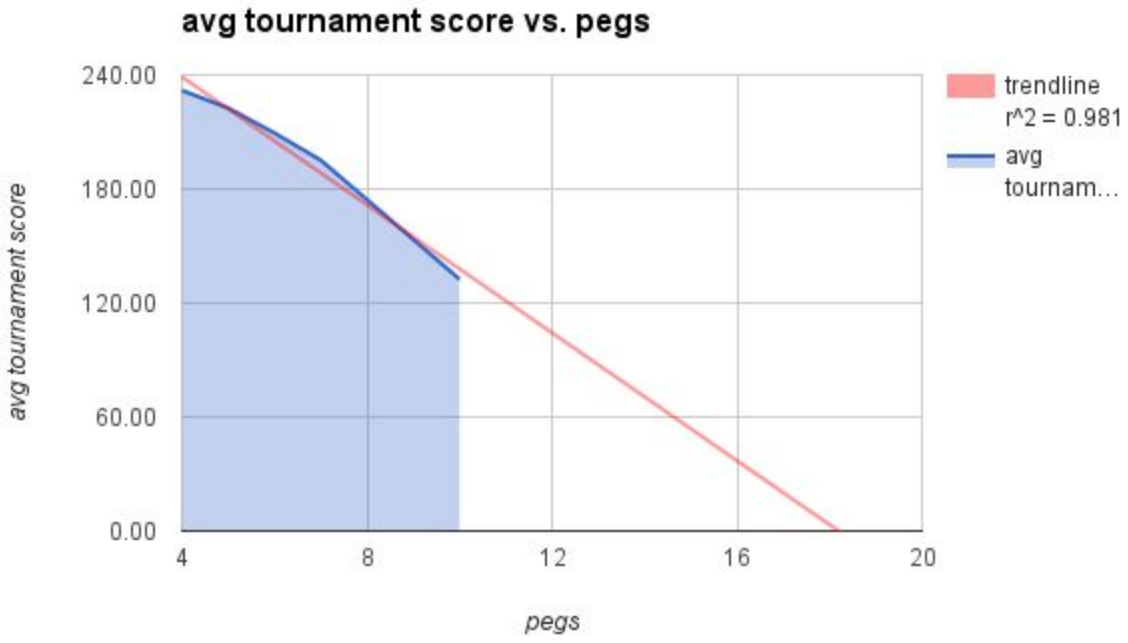


The two scatterplots below show the player's performance as pegs and colors increase.

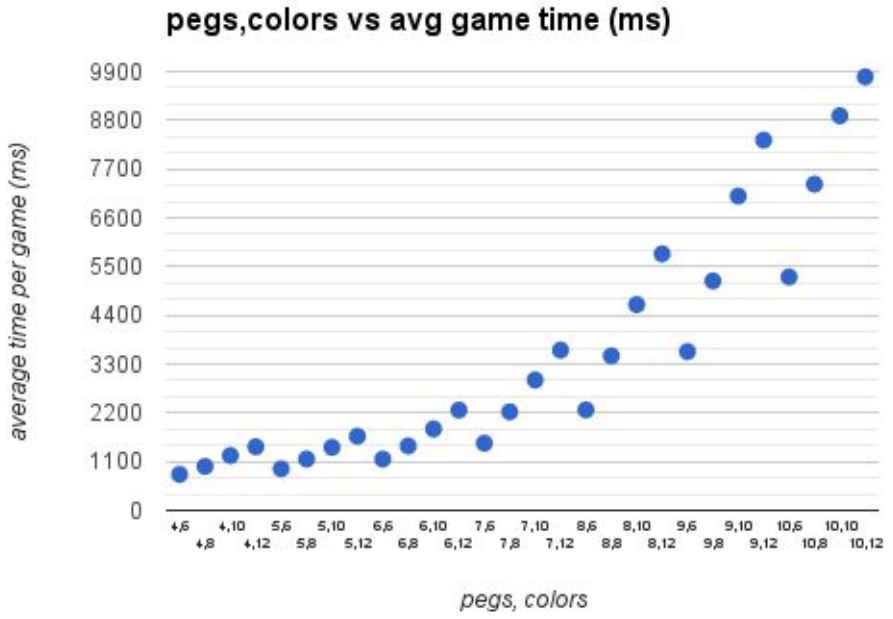

pegs,colors vs score
270
240
210
180
150
120
90
60
30
0
score
4,6 4,10 5,6 5,10 6,6 6,10 7,6 7,10 8,6 8,10 9,6 9,10 10,6 10,10
4,8 4,12 5,8 5,12 6,8 6,12 7,8 7,12 8,8 8,12 9,8 9,12 10,8 10,12
pegs, colors


Pegs vs Colors vs Score (Final Player)
Score Achievec
250
200
150
Avg Score
100
50
0
4
5
6
7
8
9
10
11
12
Colors
Pegs

Holding the color count constant at 6 colors, score decreases linearly as we increase pegs:

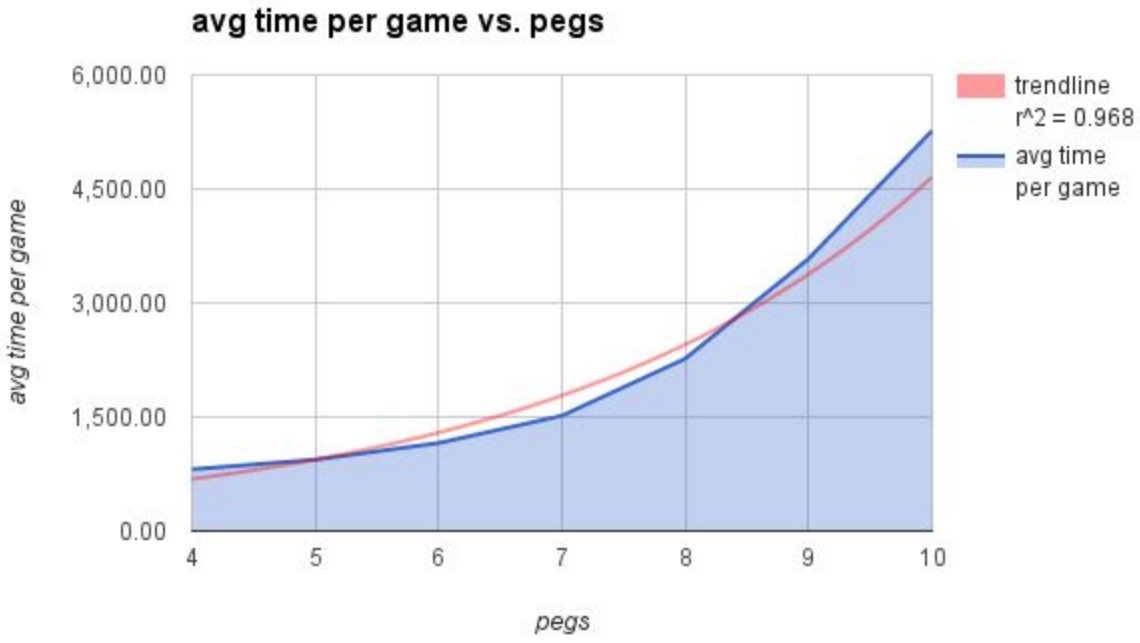


The average time per game increases linearly with the number of colors. This is illustrated by the chart below.

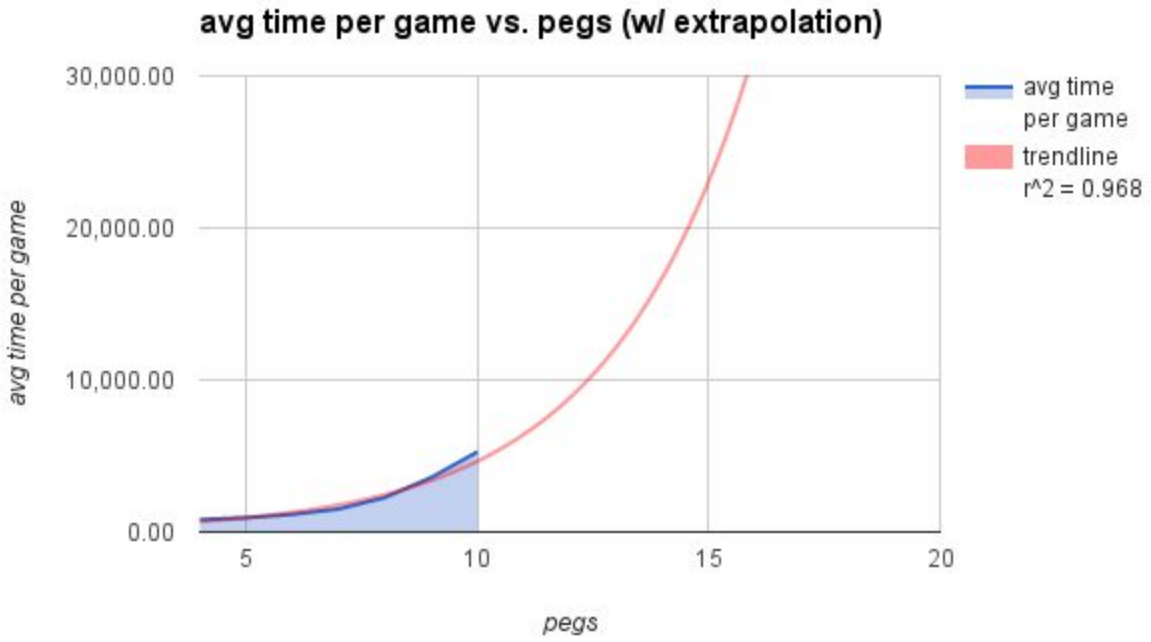


The lower time for 10,12 is not because it was winning games faster, but rather hitting the 10 second time limit and losing the game. Time per game increases for two main reasons: more

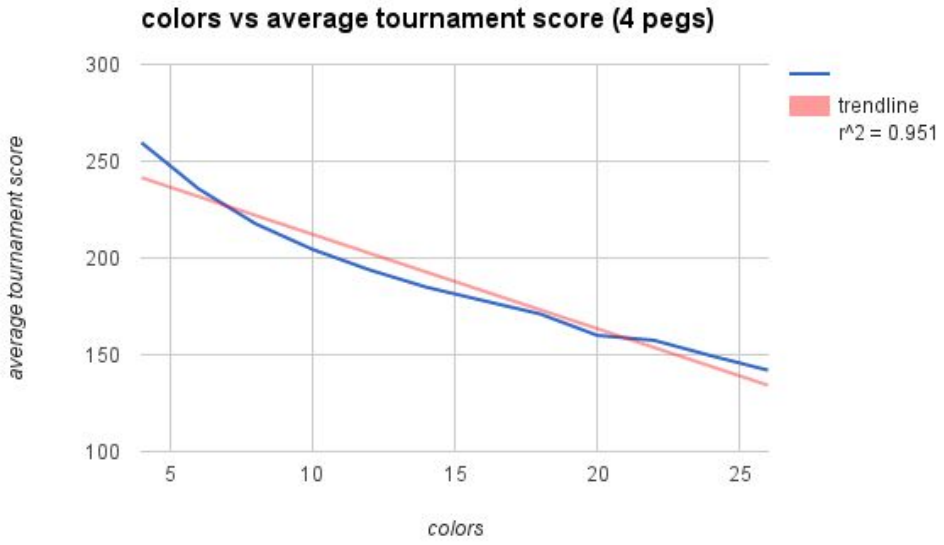
time is spent computing a guess, and more guesses are made until the game is won or lost. Holding colors constant at 6, we can see from the chart below that average time per game increases exponentially as pegs increase.

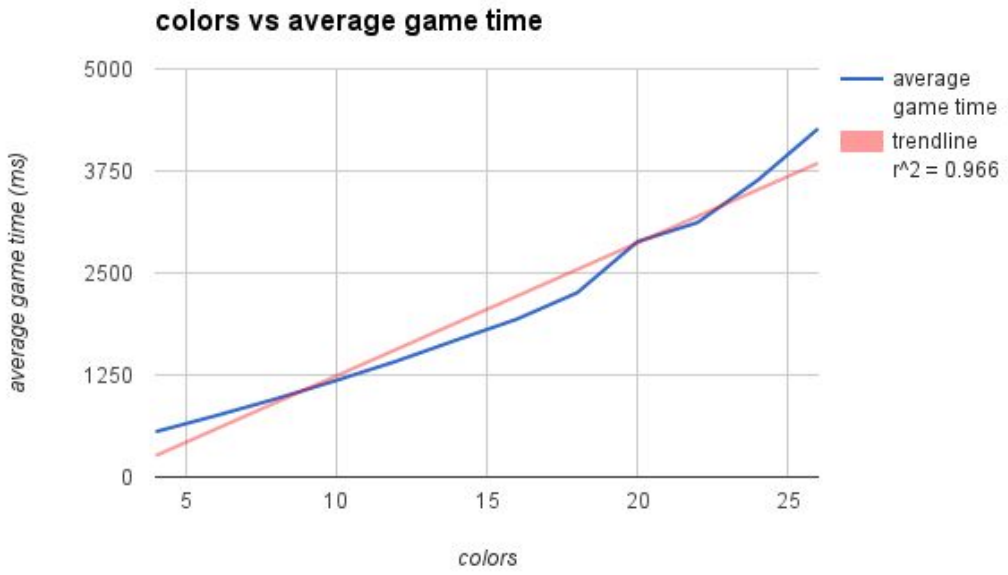


We expect that, for games of 6 colors, under the current configuration and similar computing resources, our player would time out for most games when there are more than 12 pegs.

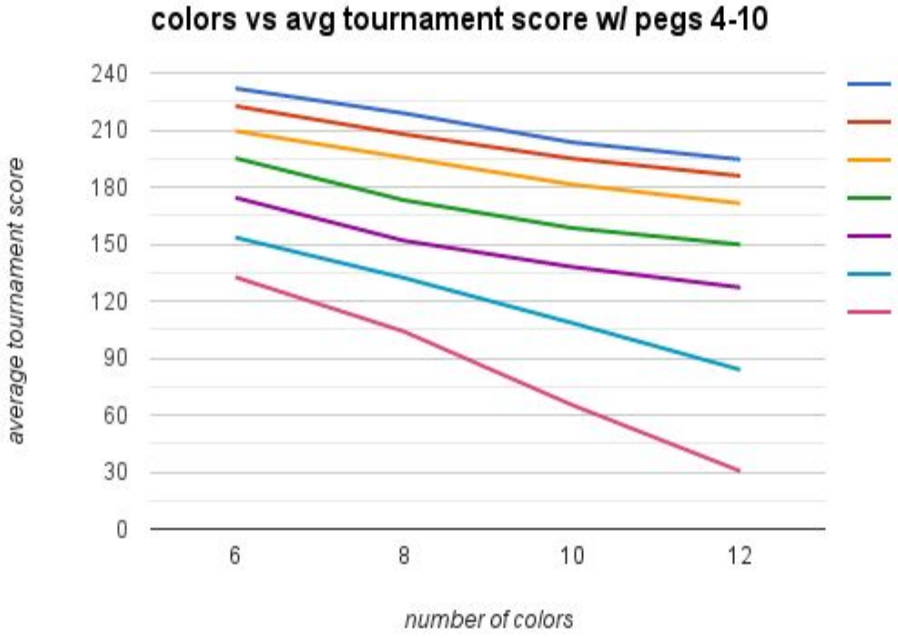


Holding pegs constant at 4, we see score decreases linearly as colors increase from 4 to 26.





This relationship seems to hold for games with more pegs as well. At 10 pegs, the score drops at a higher rate because the player needs more guesses to win, and thus more time, leading to more time outs.



Legend: each line is a number of pegs: 4 (blue), 5, 6, 7, 8, 9, 10 (magenta)

3.1 StrangeMind vs Knuth

Though our benchmarks did not directly measure the number of guesses taken to win each game, we were able to calculate the average guesses per won game using the average tournament score and the number of games won. The tournament scoring function is defined as

$$s = \sum_{i=1}^{w} 1/\sqrt{ri}$$

where s is a score for a tournament, w is the number of wins in a tournament and r_i is round we won for a given win w_i.

We can find the average number of guesses per won game in a benchmark of 5 tournaments of 100 games each by

$$r_{avg} = ((5 * p_{avg})/s_{avg})$$

where s_{avg} is the average score for a tournament and p_{avg} is the average wins per tournament.

We calculated p_{avg} by dividing the total number of wins by the total number of tournaments. The average rounds to win for both Knuth and StrangeMind is presented below:

Pegs	Colors	Solution space ($pegs^{colors}$)	Knuth				StrangeMind			
			Avg Score	Wins	Time per game (ms)	Avg Guesses to Win	Avg Score	Wins	Time per game (ms)	Avg Guesses to Win
4	6	4^6	236.31	500	43.31	4.47	231.87	500	811.82	4.650
4	8	4^8	221.22	500	535.65	5.108	218.81	500	991.93	5.221
4	10	4^{10}	206.68	494	3,793.02	5.713	203.50	500	1,237.34	6.036
4	12	4^{12}	67.3	172	2,1627.5	6.523	194.49	500	1,435.45	6.609
5	6	5^6	226.98	500	1,619.19	4.852	222.57	500	938.16	5.047
5	8	5^8	75.2	172	22,252.51	5.229	207.76	500	1,158.52	5.792
5	10	5^{10}	0	0	Limit	N/A	194.95	500	1,421.51	6.578
5	12	5^{12}	0	0	Limit	N/A	185.81	500	1,672.23	7.241

6	8	6^8	0	0	Limit	N/A	195.51	500	1,452.78	6.540
7	10	6^{10}	0	0	Limit	N/A	158.39	500	2,942.14	9.966
8	12	8^{12}	0	0	Limit	N/A	127.11	487	5,788.13	14.68
9	12	9^{12}	0	0	Limit	N/A	83.79	354	8,354.05	17.848
10	12	10^{12}	0	0	Limit	N/A	30.43	131	9,780.56	18.533

Tournaments where the time is Limit were ended prematurely because Knuth was unable to finish any games in under 10 seconds. StrangeMind is able to perform nearly as well as Knuth in smaller games, and is also able to perform in games that Knuth cannot.

Given the average number of guesses per game, we can approximate the number of codes that the player will consider as every guess requires the evaluation of population * generations codes. Knuth must consider the entire solution space, while StrangeMind considers a small portion of it which grows in proportion to the number of guesses it makes, not the number of pegs or colors.

pegs	colors	avg guesses	codes considered	solution space	% considered
8	6	8.22	164429	1679616	9.79%
8	8	10.86	217131	16777216	1.29%
8	10	13.09	261740	100000000	0.26%
8	12	14.68	293565	429981696	0.07%

4 Conclusion

The StrangeMind player succeeds at playing games of Mastermind at various configurations of pegs and colors while incorporating SCSA information. Without the time constraint of 10 seconds per game, we expect it to perform fairly well at higher combinations of pegs and colors, given some tuning of population size, total generations, mutation rates, response comparison limit, and SCSA consistency multiplier. StrangeMind will also be able to perform well at Mastermind without the knowledge on how a secret code is generated.

Given more time, we would have liked to make further improvements to our player. Our benchmarks only measured the effects of our GA parameters changing in isolation. It would be

interesting to measure the impact of variables changing together. For the sake of time, we chose a small range of values to experiment with (e.g. population = 50, 100, 150, 200, 500), but we could experiment with a wider range with smaller intervals (population=10, 11, 12...1000) to find more optimal values. Additionally, we planned to scale GA parameters with game difficulty (pegs, colors) and SCSA. To reduce the number of games lost by time out, we considered adjusting the population size and generations per guess each turn based on the remaining time in the game. Lastly, to maximize performance for this competition, we could parallelize the fitness calculation for individuals in a population which is where the most time is spent. Though we did not have time to implement these improvements, we are proud of how StrangeMind has progressed so far.

5 Contributions

Each member of this team contributed to initial research, writing and testing code, providing feedback, measuring performance, tweaking parameters, and writing this report. Our group met on almost a weekly basis either in person or through online video chats where we would discuss our progress on tasks assigned, any difficulties, and next steps.

Each member led various initiatives during this project with support from the team:

Andres Quinones- Led implementation of new mutation techniques, resolving problem with eligibility, and response pruning.

Iden Watanabe- Led literature review, outlining the paper, and investigaton of Hill Climbing

Gordon Zheng - Led initial implementation of genetic algorithm, Knuth's algorithm and benchmarking

Nishad Sharker - Led SCSA learning, SCSA matching, investigation of Rao's Algorithm

6 References

[1] Berghman, L., Goossens, D., Leus, R. (2009). Efficient solutions for Mastermind using genetic algorithms. *Computers & Operations Research, Vol. 36* (6): 1880-1885 *https://lirias.kuleuven.be/bitstream/123456789/184247/2/Mastermind*

[2] Knuth, E. (1977). The computer as Master Mind. *Journal of Recreational Mathematics 9*: 1-6.

[3] Rao, T. M. (1982). An algorithm to play the game of Mastermind. *ACM SIGART Bulletin* (82): 19-23

[4] Russell, Stuart J., & Norvig, Peter. (2010). *Artificial intelligence: a modern approach, third edition.* Upper Saddle River, New Jersey: Prentice Hall.

[5] Temporel, A., Kovacs, T. (2003). A heuristic hill climbing algorithm for Mastermind. *Proceedings of the 2003 UK Workshop on Computational Intelligence*: 189-196 *https://pdfs.semanticscholar.org/2e26/663970ac3e11ba9cd15c2d9671cf6216b5ad.pdf*