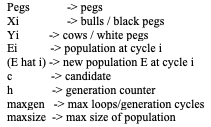
**Using an Updated Genetic Algorithm to Solve the “Mastermind” Puzzle**

            After learning the rules of Mastermind via an online application (Sarcone, 2005), we explored possibilities of implementing our agent by reading several articles, beginning with a study titled “An experimental study of exhaustive solutions for the Mastermind puzzle” (Merelo, Mora, Cotta, & Runarsson, 2012). The paper discussed four algorithms, tested on a search space of (pegs-4, colors-6) and (pegs-4, colors-8), called Entropy, Most Parts, and two new suggested algorithms combining Entropy and Most Parts in different ways. While this study was helpful in broadening our knowledge of methodologies pertaining to solving the puzzle, the relevant information enclosed was surpassed significantly by an article titled “Efficient solutions for Mastermind using genetic algorithms” (Berghman, Goosens, & Leus, 2009). It introduced a strategy using a genetic algorithm. The paper included pseudocode for its implementation, as well as the algorithm’s results: computation time and average number of guesses for sets of size (pegs-8, colors-12). After discussing the readings, we voted to pursue the genetic algorithm as our main approach to solving the Mastermind puzzle. Shortly into its implementation, we discovered an article titled “Removing the Genetics from the Standard Genetic Algorithm” in which the concept of elitism was introduced (Baluja & Caruna, 1995). The idea was that each new generation keeps a population of its most elite to carry into the next generation, thus ensuring that the most fit are not lost in the succeeding generations.

The original genetic algorithm, which proved to be insufficient for our purposes, worked on the basis of generating an “eligible set” of guesses. A guess is considered eligible if it is consistent against all previous guesses in the game thus far (Berghman et al., 2009, p. 1). We found that the approach relying on an eligible set was insufficient regarding the scope of our Mastermind competition. The overview and global scope as suggested in the paper are outlined below:

**A new genetic algorithm** - Lotte Berghman, Dries Goossens, and Roel Leus (Berghman et al., 2009, p. 5)



Set *i* = 1

Play fixed initial guess g1:

Get response *X1* and *Y1*;

**while** *Xi* ≠ Pegs **do**

i = i + 1;

Set (*Êi*) = {} and *h* = 1;

Initialize population:

**while** (*h* ≤ *maxgen* AND |(*Êi*)| ≤ *maxsize*) **do**

Generate new population using crossover, mutation, inversion, and permutation;

Calculate fitness;

Add eligible combinations to (*Êi*) (if not yet contained in (*Êi*));

*h* = *h* + 1;

**end while**

Play guess gi element of (*Êi*);

Get response *Xi* (bulls) *Yi* (cows)

**end while**

The algorithm uses a fixed initial guess and then proceeds into the main algorithm. For the second move, a new random heterogeneous population is generated. The study uses an initial population size of 150. Each successive generation is populated by the parent generation. Two parents are selected at random and mated using one-point or two-point crossover. One-point crossover splices the parents once at a random index before joining them. Two-point crossover does the same, but splices at two random indexes. After mating, the following occurs:

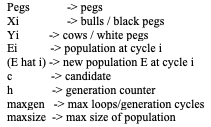
* a three percent chance of mutation, in which a random index is given a random color
* a three percent chance of permutation, where the colors of two random indexes are switched
* a two percent chance of inversion, in which two indexes are randomly picked and the subset from the first index to the second is reversed.

    Once the population is generated, the algorithm uses the concept of fitness to determine the quality of the guess. Berghman et al. accomplish this by comparing each child to all previous guesses as if it were the secret code; the same number of black and white pegs for every previous guess indicates that the code is eligible as a possible answer (Berghman et al., 2009, p. 7-8). Once an eligible set is produced, each code is assigned a score corresponding to its similarity to all other codes in the eligible set. The theory behind this methodology is to avoid a purely random selection process, which may result in losing good quality children as populations are generated. This can be implemented by adding a preference to children that are more similar to the rest of the eligible set than those that are not, an indicator that the algorithm has begun to improve the population’s quality and converge on the solution through mating. However, this concept of eligibility is where the algorithm falls short for our purposes. As the number of played guesses increases, eligible guesses become more rare, and larger board and color sizes create a search space so expansive that finding an eligible set within five seconds seems unlikely given our current computing power. This is where the concept of elitism came into play.

Rather than choosing an eligible set from the population after a maximum number of generations ormaximum eligible set size, we integrated the principle of elitism that was put forward in “Removing the Genetics from the Standard Genetic Algorithm” (Baluja & Caruna, 1995). Eliteness scores the guesses based on their fitness and returns a sorted list of the *most eligible* subsets in the population (lowest fitness value when scored against all previous guesses). It then retains a proportion of those most eligible in the next generation and generates the remaining portion of the population by means of the previously described mating process. According to Baluja and Caruana, elitist selection reduces the probability of losing the best guess due to random chance (Baluja & Caruna, 1995, p. 2). The choice in using eliteness comes with a caveat: since we do not use the comparison feature as described previously for the eligible set, we may not curtail the search space as efficiently. Our imposed time limit restricts the number of possible generations, possibly preventing us from ever finding an eligible guess. Our updated algorithm is as follows:

**Genetic Algorithm Based on Eliteness**

Set *i* = 1



Play fixed initial guess g1:

Get response *X1* and *Y1*;

**while** *Xi* ≠ Pegs **do**

    i = i + 1;

    Set (*Êi*) = {} and *h* = 1;

    Initialize population:

**while** (*h* ≤ *maxgen*) **do**

Calculate and sort population by fitness;

       Push elite percent of past generation into new generation;

**while** (|*Êi* | ≤ *maxsize)*

    Generate child using crossover, mutation, inversion, and permutation of top 50

                   percent of past generation;

            If child is not a duplicate, push child to new generation;

**end while**

*h* = *h* + 1;

**end while**

    Play guess gi first element of (*Êi*);

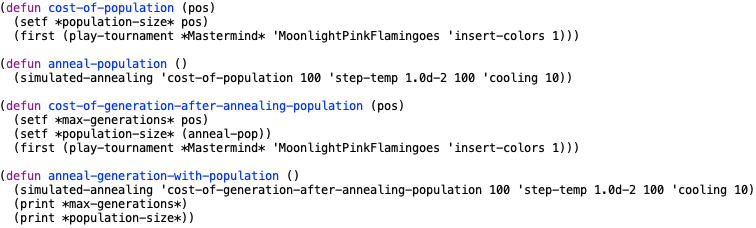
    Get response *Xi* (bulls) *Yi* (cows)

**end while**

The methodologies involved in minimizing the guess count came in a multitude of forms. The majority of our time was spent creating a general-purpose player that was quick and efficient against any case. Due to the nature of the genetic algorithm, all SCSAs performed fairly similarly before adding constraints for biased SCSAs; as such, we decided to use “insert-colors” (random color generator) to simulate the worst possible case for the majority of our learning. With our general-purpose player, we experimented with the following:

**Altering the generation/population count** - Larger generations lead to potentially higher quality candidates and reduce the overall number of guesses. The greater the board size and number of colors, the more time is spent analyzing fitness in order to reduce the number of guesses, therefor, a “sweet spot” exists for each search space; an optimal generation count that can balance the tradeoffs and keep each round under five seconds. As the search space grows, we reduce the generation count in order to favor more guesses at the expense of higher quality guesses, preventing a timeout. We conceived two possible approaches to obtaining appropriate generation counts, the first through analysis of prior outcomes, and the second being an impromptu method of raising and lowering the generation count and observing the results.

*Case 1 - analyzing the data using simulated annealing.*

To find appropriate generation sizes, we hypothesized that simulated annealing could generate the best possible generation sizes over time. As represented in an article written by mathematician Atabey Kaygun, “the idea is that we randomly iterate our points using a stochastic process. The probability of jumping from one state to the randomly chosen next point depends on the ambient temperature. If the cost *ϕ*(*xi*+1) in the next step is smaller than the cost *ϕ*(*xi*) at the current state then the probability of a jump is e*ϕ*(xi + 1) - ϕ(xi)kT. Otherwise, the probability is set to 1” (Atabey Kaygun, 2013, para. 3). Using our current terminology, we would initially choose a random generation count. As time moves forward using a constant k and temperature T, we would converge on the optimal generation count. Due to the nature of our algorithm, that convergence on the global maximum is inevitable (given a slow enough cooling rate) as there should exist no other maximum. Upon theoretical analysis, we realized that for each generation count, there must also exist an ideal population count. The rationale behind this is the assumption that high generation counts may score poorly, as the population size could be too high and force a timeout. Overpopulation could cause unnecessary computation after a highly elite population has already been found. In order find the both the best generation count and best population size, we nested the annealing process for the population size within the annealing of the generation count. Current analysis cannot be provided at this time, as time complexities were too great. A section of our lisp implementation for the annealing process is contained below. 

*Case 2 - on the fly generation count updates*

In our current implementation, a generation count is conservatively chosen in order to reduce the probability of exceeding the five-second timeout. Since the tournaments will be run on a powerful machine, it is likely that our chosen generation counts are too conservative. We speculate that after a certainnumber of guesses, we could calculate a weighted average of the time spent for each guess. Using this calculation, we postulate that it is possible to increase generation count to the minimum number of guesses without inducing a timeout. Doing so would likely lead to higher quality candidates and thus converge on the secret code in fewer guesses.

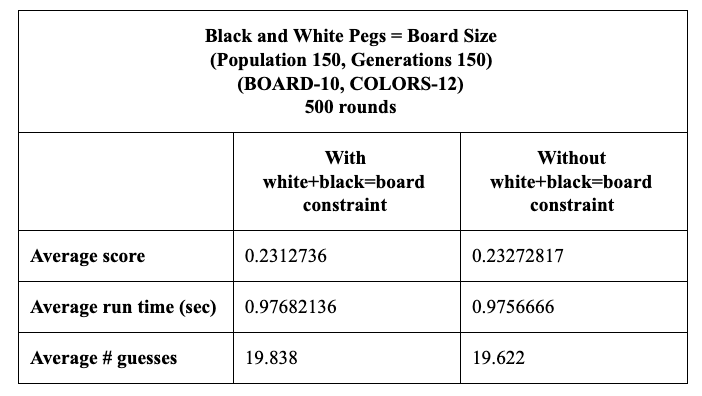
**Combining the elite and eligible algorithms** - If and only if an eligible set exists with a size greater than two, the similarity heuristic could be used to determine an eligible guess (the “best” guess is the guess most like all others in the eligible set).  If the set size is less than two, then the difference between the most elite and the most eligible guesses is either negligible or non-existent. In practice, we found that eligible guesses are highly probable within the first few guesses. Using this knowledge, we hypothesize that using the eligible set until it reaches a size less than three could make limited reductions to the size of the search space. We reason that after the eligible set drops below a certain number, it is increasingly unlikely that we will find an eligible guess in the proceeding generation. These possible reductions are attributed to the similarity heuristic as described above. In our current implementation, many members of the elite population often share the same fitness value, especially in early guesses on large sets. Because of this, there may exist another guess with the same fitness, within in the current population, that would reduce the search space.

**Eliminating known wrong colors -** We implemented three ways of constraining the color set.

*Case 1* - *both the black and white pegs equal zero for the previous guess -* All colors from the previous guess are removed from the color set. Since we know that they do not exist within the secret code, they can be discarded. We did not record statistical data on the change, but postulate that significant improvements have been made by constraining the search space with minimal increase to time complexity.

*Case 2 - only the black pegs are zero for the previous guess* - All colors at their index are removed from the color set. In this scenario, we initially stored the color set as a list of all available colors at each index (i.e. guess (A B C) on pegs-3, colors-3 with last response (0 1 1) returns ((B C) (A C) (A B)) as eligible colors). With an unlimited run-time this would create a marked improvement, however, the implementation that we used was poorly written and cause run-time to double. Since the eligible color set is stored in a list, the lookup time for many of our function calls was drastically reduced. Storing this in an array would have possibly minimized the issue as the time complexity of a lookup would have been reduced to O(1).

*Case 3 - white and black pegs sum to the board size -* When the sum of black and white pegs is equal to the board size, the previous guess contains all available colors. Interestingly, this provided no known statistical improvement, as shown in the following table. It is likely that by the time this function is implemented, our population has already converged on the solution. Thus, it only serves to increase the run time.

**

**Rapid monochromatic guesses -** For larger boards at (pegs-15, colors-20) and higher, the search space started becoming unwieldy, and our genetic agent struggled to converge on the secret code within the time constraint. In order to aid in this, we sequentially guessed all colors to eliminate those that do not exist in the secret code. At the cost of (# of colors) guesses, the search space became limited enough to prevent timeouts on larger search spaces and thus improved the quality of the score.

**Applying probability to each color at each index** - When the last response contains more black pegs than half the board size for the previous guess, it is probable that majority of colors in the last guess are currently at the appropriate index. In this case, the probability of each color in the previous guess could be increased at its index in the eligible color set in order to favor the chance that it will be placed there again. In practice, we used the same implementation of an eligible color set as *Case 2* in the above process: *eliminating known wrong colors*. Again, it would be useful to retest this function using a data structure that has O(n) access time, as computation times were disproportionately impacted.

**Adjusting the initial guess** - As suggested by the Berghman, Goossens, and Leus algorithm, a good first guess should provide a lower average guess count. Their analysis states that a guess of (A A B C) proves slightly better than an initial guess of (A A B B) as predicted by Knuth (Berghman et al., 2009, p. 5). However, in testing on (pegs-4, colors-6) for 1000 rounds, this was not confirmed by our own algorithm. The following results were obtained:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **FIRST GUESS**  **(Population size 150, Generations 150)** | | | | | | | |
|  | **(PEGS-4, COLORS-6)**  **1000 rounds** | | | **(PEGS-10, COLORS-12)**  **100 rounds** | | | |
|  | **AABB** | **AABC** | **RAND** | **AAAAABBBBB** | **AABBCCDDEE** | **AAAAABBBCC** | **RAND** |
| **Average score** | 0.47129 | 0.46828 | 0.473837 | 0.23140092 | 0.22890417 | 0.23064768 | 0.23201 |
| **Average run time (sec)** | 0.07189 | 0.07122 | 0.06896 | 0.9765466 | 1.0183845 | 0.9819072 | 0.97847 |
| **Average # guesses** | 4.652 | 4.711 | 4.637 | 19.71 | 20.24 | 19.89 | 19.79 |

We believed that this discrepancy from the predicted outcome stemmed from our generation count and population size set at 150. With a higher generation count and subsequently reduced search space, our populations converge very quickly. However, by testing with a reduced generation count of 20 and population size of 30, we were able to corroborate the study’s findings that an initial guess of (A A B C) would outperform (A A B B) slightly. Interestingly, the random initial guess outperformed both of these initial guesses, suggesting that the algorithm prefers homogeneity in its initial guess.

**Learning for biased SCSAs** - The genetic algorithm can perform effectively up to 12 pegs and 14 colors against an unspecified SCSA. When the board size and number of colors surpasses those thresholds, the agent will consider the particular SCSA it is playing against. It will first play solid color guesses in order to reduce the domain of possible colors and subsequent search space. Depending in the result the agent receives from the previous single color guess, it can either remove the color from the list of remaining colors or move it to the back of the list. When the length of the list matches the limit for that particular SCSA, the agent will then proceed to the genetic algorithm and prepare the initial population, starting with a guess generated from the number of pegs and remaining colors.

We determined the maximum possible of colors incorporated by each opponent SCSA by examining the code or the provided output. For instance, we were able to observe that the codes produced by “mystery-3” consist of a repeating triplet of three different colors. Similarly, the codes generated by “mystery-5” consist of two alternating colors. For the SCSAs “two-color” and “two-color-alternating,” the maximum number of colors that can appear in the code is two; the genetic algorithm proceeds with determining the positions of those two colors. For ab-color, the domain of possible colors could be immediately reduced to A and B without wasting guesses. Mystery-4 produces codes containing two to four colors, so the algorithm will limit the domain to four colors.

In order to aid us with the performance measurement of our agent, we wrote a benchmarking program that automated much of the data collection and analysis, providing useful metrics such as average number of guesses, average score, average run time, and win rate. This is done using a loop that continues to play up to a specified number of rounds, allowing us to test the consistency of our agent across hundreds or even thousands of rounds if necessary. During the loop, the program marks the time before each round is started, using the LISP function *get-internal-real-time*, then again when the round is over. By subtracting the first time mark from the second, we are left with internal time units that are then divided by the LISP function *internal-time-units-per-second*, converting the round duration into seconds. The sum of all the run times is divided by the number of rounds played to determine the average run time.

To calculate the average number of guesses made, we used a data structure \*guesses\* to record of the prior guesses put forward by our agent. We read the length of this list as the number of guesses made by agent during that round, and in similar fashion to the average run time, we added up the guesses then divided by the number of rounds played to get the average number of guesses.

The scoring we used for evaluating our agent was not the tournament score provided at the end of each tournament by \*Mastermind\*, but the round score, with 0.5 being the maximum score. We chose this approach since we did not want the data to be influenced by the total number of rounds played in a tournament. Examining round performances on an individual basis provides a clearer metric for evaluating the speed, accuracy, and quality of the guesses made rather than looking for the highest tournament score.

To calculate the win rate, we kept track of how many rounds returned a round score greater than zero, as a zero indicates that the round was lost. We divided rounds won by total rounds played to get the win rate. These were the methods used to retrieve the data we reported on in our paper.

The features included in our evaluation are not the only features we considered. In order to evaluate agent behavior, we recorded the winning sequence whenever a round was won. This data was saved to logs by our benchmarking program so that they could be examined later.

References

Atabey Kaygun. (2013, August 25). Today, I am going to look at a standard optimization algorithm called Simulated Annealing... [Tumblr post]. Retrieved from https://kaygun.tumblr.com/post/59316257966/simulated-annealing-in-lisp

Baluja, S., & Caruana, R. (1995). Removing the Genetics from the Standard Genetic Algorithm. *ICML*.

Berghman, Goossens, & Leus. (2009). Efficient solutions for Mastermind using genetic algorithms. *Computers and Operations Research,* *36*(6), 1880-1885.

Merelo, J., Mora, A., Cotta, C., & Runarsson, T. (2012). An experimental study of exhaustive solutions for the Mastermind puzzle.

Rao, T. (1982). An algorithm to play the game of mastermind. *ACM SIGART Bulletin,* (82), 19-23.

Sarcone, G. A. (2005, January 17). Mastermind History. Retrieved from http://www.archimedes-lab.org/mastermind.html