Team Members:Alvin Lam, Constantine Firinidis, Sacit Gonen, and Rubaiyat Rashid

Artificial Intelligence

Professor Susan Epstein

14 May 2018

MASTERMIND PROJECT REPORT

**Survey:**

The very first thing we did as a team was learn what the Mastermind game was. We did this by playing it on <http://www.webgamesonline.com/mastermind/> . Once we understood the goals and rules, we started to think about the best ways to approach a game from a human perspective, hoping that these approaches could eventually be implemented into our program. At first, instinctively, we tried some of the baseline approaches, like trying the same colors to rule them out. We then decided to compare the basic makeshift approaches we found successful to ones that people posted online. According to a video we watched on YouTube, <https://www.youtube.com/watch?v=XX5TlB6xT3M>, a ‘good’ guessing strategy is to divide the guess into two different colors. This would allow us to simultaneously reduce positions for colors and introduce new possible colors. Although this is a bit efficient for 4-6, it does not work well for larger number of pegs and colors. Scalability became an issue as we worked to generate different ideas to solve mastermind efficiently.

Mastermind scalability being the issue, and our team name being CARS, we remembered the video from class using a genetic algorithm to mutate cars to better adapt to an irregular and ever changing track, <http://rednuht.org/genetic_cars_2/>. We researched genetic algorithms that may already exist for playing Mastermind and found a paper online that described a new genetic algorithm that can be used to play Mastermind. Just like in the online cars example, this genetic algorithm builds off of the most successful prior results. We concluded that this could solve our scalability issue because genetic algorithms do not rely on constraints, number of pegs and colors, to generate a solution.

**Discussion:**

We started with baseline three which guessing all colors monochromatically except the last color. The result of this algorithm was adequate for 4-6 player. However, it was not at all efficient nor scalable as in terms of time. The main issue was we had to generate a large list of permutations. When submitting guesses we did not prioritize the potential guesses from the list. We also completely ignored the last-response where the last-response could’ve been used to make better guesses and solve the problem in shorter amount of time. We also implemented baseline one, baseline two, and baseline four. Baseline four was more efficient than the other three baselines as it takes at most p+c where p is number for pegs and p is the colors.All four of these baseline strategies were not efficient nor scalable for us work on it as the tournament player.

Based on the failure of the baseline strategies, we decided go with a strategy that actually used last-response to generate guesses efficiently. We found an algorithm online written in python, <https://github.com/sp4ke/mastermind-genetic-algorithm/blob/master/gamm.py>, which was the inspiration for our program. In the new genetic algorithm introduced by Berghman and Goossens, “a large set of eligible codes are collected throughout different generations.” The quality of these codes is calculated using a fitness score function which compares the codes to every previous guess.

The initial guess is randomly generated based on the SCSA used during the tournament. By doing that, the initial guess will generate better populations because based on the algorithm because we should give better results if we start with a “good” initial guess . The algorithm calls for us to generate a set of random guesses using the SCSA function used in the tournament. The paper called for randomly generated guesses but since SCSA gives us information as to how the secret code is generated. This is called a population in which each guess has a chance to go through different methods of transformations through our functions of crossover, mutation, or permutation.

For each candidate guess we evaluate how well it would do as an actual guess by running it through our fitness function. The fitness function takes previously submitted guesses stored in a global list and treats each of these previous guesses as the secret code. We then compare how well our new candidates would perform against this secret code which determines their fitness score. This will provide us with a varied set of guesses and expand our pool for potential “good” guesses. Each population that we generate will act as a generation. For this program, we decided to create at most 150 generations with each generation being at most 100 guesses. As we are generating guesses, if we find that any of these candidate guesses gives us a fitness score of zero, which means the colors of the candidates are eligible, we append that candidate guess to an eligibles list . After we have finished generating the maximum number of generations, we go through the eligibles list and attempt to find any duplicate candidate guesses or candidate guesses we have already tried and remove them from the list. As a result, we get a list that holds the “best” possible guesses from the all the generations. The resulting list contains all guesses which we will try as our next code. If for some reason we received no eligible guesses, we attempt to generate another set of generations in which we double the maximum number of population and cut maximum generation by half in hopes of getting at least one good guess. We then proceed to make guesses by treating the eligible list as a queue and popping the first item in that list and using it as our next guess.

**Theoretical Analysis:**

We generate **N** populations, where n is the population cap, **G** amount of time, where g is the generation cap. We generate up to O(**NG**) potentialguesses for every guess that we make. Which means at most of we do this computation 100 time per tournament. However, in average cases, the amount of guesses we make correlates to the number of pegs for the tournament. For every candidate guess, there is a chance to of transforming via mutation, crossover and/or permutations which may require bit more extra computation. The search space for potential guesses grows exponentially with the amount of pegs used for the tournament. For example, when we ran 8-10 game, we saw that our program was stalling and took too long to generate a guess for play. We can confirm that the search space grew exponentially because when we ran 7-9, we got results compared 8-10 where we ran out of time. It seems that 6-8 game seems to produce the best results.

Every guess that we make should subsequently produce result that the previous guess. As pegs and colors increase, the amount of possible combinations. So, when we generate populations, the chance that we get an eligible guess drastically decreases because of the increasing set of possible solutions. This is the reason why we do so poorly for 8-10. As a result, we assume that increasing the pegs and/or colors will simply generate worse results in terms of time.

**Experimental Evaluation:**

**Baseline 3: 8-10**

**Insert-Colors**

|  |  |
| --- | --- |
| **Guesses** | **Time(in seconds)** |
| **100** | **0.118104** |
| **100** | **0.074342** |
| **100** | **0.078464** |
| **100** | **0.082608** |
| **100** | **0.075232** |

**Two-color**

|  |  |
| --- | --- |
| **Guesses** | **Time(in seconds)** |
| **100** | **0.082726** |
| **100** | **0.096009** |
| **100** | **0.091245** |
| **100** | **0.083544** |
| **100** | **0.087968** |

**Ab-color**

|  |  |
| --- | --- |
| **Guesses** | **Time(in seconds)** |
| **100** | **0.082406** |
| **100** | **0.086371** |
| **100** | **0.091846** |
| **100** | **0.106228** |
| **100** | **0.088770** |

**Two-color-alternating**

|  |  |
| --- | --- |
| **Guesses** | **Time(in seconds)** |
| **100** | **0.077010** |
| **100** | **0.079819** |
| **100** | **0.089741** |
| **100** | **0.082186** |
| **100** | **0.083298** |

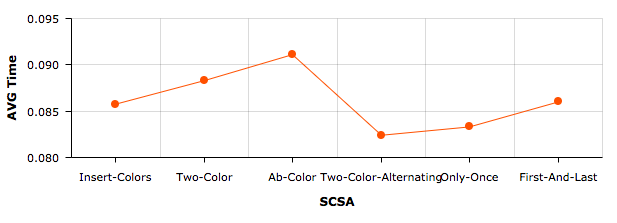
**Only-once**

|  |  |
| --- | --- |
| **Guesses** | **Time(in seconds)** |
| **100** | **0.077453** |
| **100** | **0.072089** |
| **100** | **0.077238** |
| **100** | **0.077020** |
| **100** | **0.112405** |

**First-and-Last**

|  |  |
| --- | --- |
| **Guesses** | **Time(in seconds)** |
| **100** | **0.086075** |
| **100** | **0.079611** |
| **100** | **0.08767** |
| **100** | **0.079290** |
| **100** | **0.085231** |

**Baseline-3 (8-10)**



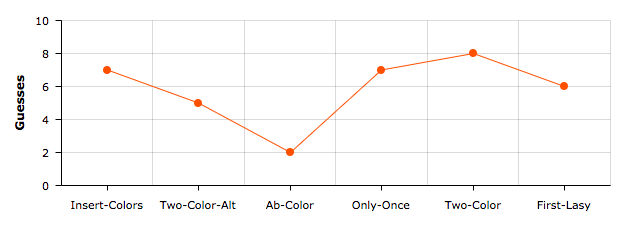
**Tournament Player**

|  |  |  |
| --- | --- | --- |
| **SCSA (Pegs-Color)** | **Guesses** | **Time** |
| **Insert-Colors(7-9)** | **6** | **21.356688 (timed out)) (CPU 0.569938)** |
| **Insert-Colors (6-8)** | **7** | **2.488791 (cpu 0.069359)** |
| **Two-Color-Alternating (6-8)** | **5** | **0.15762, (CPU 0.000376)** |
| **AB-Color (6-8)** | **2** | **0.001288** |
| **Only-Once (6-8)** | **7** | **0.799241 (CPU 0.028454)** |
| **Two-Color (6-8)** | **8** | **0.139528(CPU 0.003607)** |
| **First-and-Last (6-8)** | **6** | **0.051513 (CPU 0.001173)** |

We tried 7-9 which made 6 guesses but time out. Thus, we did not finish the tournament so we scaled down to 6-8 which produced results by playing the game till the end.

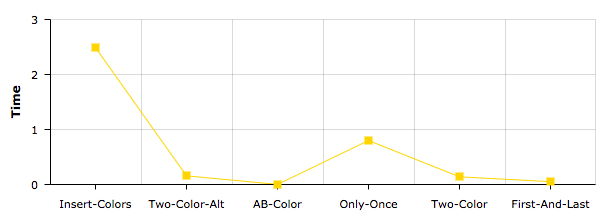
|  |  |  |  |
| --- | --- | --- | --- |
| **Pegs/Colors** | **SCSA** | **Guesses** | **Time** |
| **4-6** | **Insert-Colors** | **11** | **0.000269** |
| **6-8** | **Insert-Colors** | **100** | **0.007575** |
| **8-10** | **Insert-Color** | **Timed out** | **24.003** |
| **4-6** | **AB-Color** | **7** | **0.000363** |
| **6-8** | **AB-Color** | **100** | **0.00300** |
| **8-10** | **AB-Color** | **Timed out** | **20.9848** |

Here we increased the number of pegs and colors.

**Tournament Player Guesses**

This displays the average number of guesses for 6-8

**Tournament Player Time(in seconds)**



This displays the average time for for 6-8.

*Sources*

<https://lirias.kuleuven.be/bitstream/123456789/184247/2/mastermind.pdf>

<http://rednuht.org/genetic_cars_2/>

<http://www.webgamesonline.com/mastermind/>

<https://github.com/sp4ke/mastermind-genetic-algorithm/blob/master/gamm.py>