

## CSCI 350 Project cover page

Team name: Dinosaur

Our team, whose signatures appear below, completed this project as a group effort. By our signatures, we indicate that we agree that each of us has made the following contributions. (Given our virtual environment, I suggest that one of you enters all the text, turns this form into pdf, and circulates it for signature.) **Do it now, before you forget.**

**Member 1** (enter your contributions here and then both print and sign your name)

Worked on report: Search space analysis, SCSA complexity analysis, Figure 1, 2, 3, 4, Discussion

Agent: code for dinosaur agent (except what Nga Yu Lo wrote), OnlyOnce, TwoColorAlternating (finished Kyras work), mystery2, mystery5 strategies

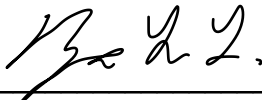
**Adam Samulak**



**Member 2** (enter your contributions here and then both print and sign your name)

Implement Baseline 2. Implement random color guessing, AB Color and TwoColor strategies. Formalize peg checking theoretical analysis. Helped to refine make\_guess function. Worked on report.

**Nga Yu Lo**



**Member 3** (enter your contributions here and then both print and sign your name)

Implemented Baseline 3, background reading (reference #s 2, 3, 4, 5), evaluation analysis (Player vs. Baseline 3) and comparison between all SCSAs, generated Tables 1 and 2, tested tournaments on all SCSAs and mystery codes, generated text results for analysis, worked on TwoColorAlternating strategy but completed by Adam due to family problems. Worked on report.

**Kyra Abbu**



**Member 4** (enter your contributions here and then both print and sign your name)

Implemented Baseline 1, worked on implementation of Rao's algorithm, probabilistic analysis and comparisons for the SCSAs and the mystery codes, suggested improvements for main agent for each mystery codes, and wrote a tournament test generator. Worked on report.

**Eric Li**



# Dinosaur Player

Adam Samulak, Eric Li, Nga Yu Lo, Kyra Abbu

May 10, 2021

## 1 Background

(1) According to Donald E. Knuth, there is an algorithm that always solves a 4 peg problem in 5 moves. This article was helpful such that it helped us understand how to approach and start creating our agent. Especially it gave us an insight into how to analyze the space complexity in our problem.

(2) Kenji Koyama and Tony W. Lai’s paper is similar to the knowledge provided by the 4 peg problem algorithm by Knuth. In this paper, they found an optimal strategy that uses at most 5 guesses in the worst case. This served as additional understanding for the Mastermind problem.

(3) Geoffroy Ville presents an optimal strategy for solving a 4-7 Mastermind problem using depth-first branch. This paper was useful during some of the testing and evaluation process of the agent, such that certain baselines worked better in different peg-color combinations. The knowledge it provided allowed us to think and approach the problem better.

(4) Wolfram MathWorld explains how Mastermind works and its background. It also provided strategies published by authors like Knuth as a starting point in creating our agent and dealing with the baseline players and their scalability problems.

(5) There is a YouTube video by pressmantoy that demonstrates how to play Mastermind. This served as the first step in understanding what the game is and how it is played.

(6) Aside from creating baseline players 1, 2, and 3, we attempted to create a fourth baseline using Rao’s paper. Although unsuccessful, it was helpful moving forward in creating our agent.

## 2 Baseline Implementation

### 2.1 Baseline 1

B1 is the simplest of the baselines. It generates the entire search space and traverses lexicographically. B1 does not account for bulls and cows and is thus the most naive and inefficient solution. B1 is not scalable and will take exponential time.

### 2.2 Baseline 2

B2 begins with a search space of all possible combinations and guesses by lexical order. It reduces the search space based on the response from the last guess. It follows three main rules to eliminate choices.

First, if one color in the guess appears more than the total number of bulls and cows, we eliminate any choices that have the same property. For example, if AAAB gets one bull and one cow, then there must not be three A’s in the solution. Given the number of bulls and cows, B2 also creates a set of all possible correct colors and eliminate choices that does not fit these possibilities. Using the example of AAAB, with one bull and one cow, the solution must have either two A’s or an A and a B. So any choices without these colors must be eliminated.

The last rule is the most strict and is therefore done last. Using only the number of bulls in the response, B2 creates a set of all possible colors in the guess that are in the right position. It

will then eliminate any choices that does not fit any of the possibilities. For example, if the guess was ABAB and gets a response of 2 bull and 0 cow, then it could be the first and last positions are correct, or the first and second, and so on. Then consider BABA, which does not match any of the possibilities. Therefore, B2 will eliminate this choice.

Following these rules, our player can win any 4-6 game within 100 guesses. This is, however, not scalable as the board length increase since the size of the sets of possible correct colors and correct pegs is of  $O(n^2)$ . Therefore, the player is hindered by the time it takes to check against each possibility.

### 2.3 Baseline 3

B3 makes the first  $c-1$  guesses monochromatic, where  $c$  is the number of colors. For example, if there are 4 colors: A, B, C, and D, the first 3 guesses are 'AAAA', 'BBBB', and 'CCCC' and the  $c$ th guess and more are based on the color distribution of A, B, and C from the first 3 guesses.

For the first  $c-1$  guesses, B3 eliminates the choices that do not correspond to the number of colors that every guess reports. For example, if the first guess is 'AAAA' and the guess has 3 correct colors, B3 eliminates any choice that does not have 3 As in any order. It continues to do this until the  $c-1$  guess.

After this process, B3 follows the remainder of the process applied in B2. The only difference between B3 and B2 is the initial step, in which B3 immediately bases its choice selection on the color distribution.

## 3 Design and Implementation

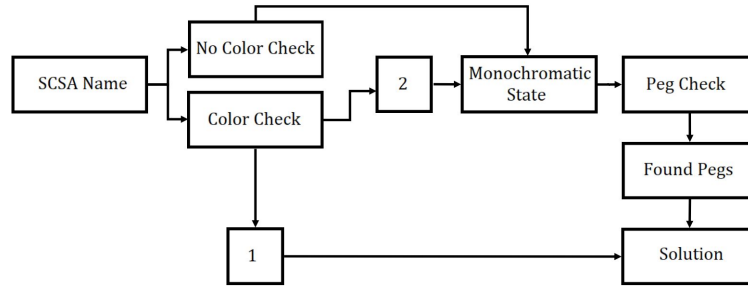


Figure 1: Pipeline for codes with 1 or 2 colors.

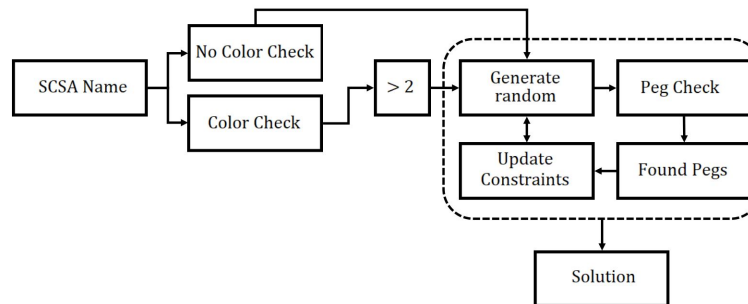


Figure 2: Pipeline for codes with 3 or more colors.

We decide to explore the strategy of B3. Our agent begins by guessing monochromatic guesses until it finds all the colors in the secret code and their occurrence. For certain SCSA, color checking is not required. And for certain SCSA, specific strategies are implemented to determine the correct pegs. Otherwise, after it has found all the colors in the secret code, our agent generates a random guess. It then performs a peg checking stage to determine the correct pegs using the responses it gets. A new random guess is generated based on the new information gained and the cycle repeats until the code is revealed. Figure 1, 2, and Algorithm 1 illustrate the pipeline our agent follows.

---

**Algorithm 1** *make guess()*

---

```
1: if first guess then
2:   Initialize all variables
3:   if OnlyOnce or ABColor then
4:     Initialize appropriate colors
5:   else
6:     return guess  $\leftarrow$  guessColor()
7:   end if
8: end if
9: if not first guess and not all colors found then
10:   All codes attempted  $\leftarrow$  last response(bulls), last response(cows)
11:   AllColors  $\leftarrow$  color(guess), last response(bulls)+last response(cows)
12:   if all colors found then
13:     if TwoColorAlternating or mystery5 or TwoColor or mystery2 or count number of
        colors() = 2 then
14:       Initialize SCSA specific strategy
15:     end if
16:   else
17:     return guess  $\leftarrow$  generate random code()
18:   end if
19: else
20:   return guess  $\leftarrow$  guessColor()
21: end if
22: if all colors found then
23:   if TwoColorAlternating mystery5 or mystery2 or ABColor or TwoColor or count number
        of colors() = 2 then
24:     Complete SCSA specific strategy
25:   else
26:     constraint elimination()
27:     colors found()
28:     if last response(bulls) > desired number of bulls then
29:       All codes attempted  $\leftarrow$  last response(bulls), last response(cows)
30:       Start peg checking
31:     else
32:       return guess  $\leftarrow$  generate random code()
33:     end if
34:     if All pegs weren't checked then
35:       Saved responses  $\leftarrow$  last response(bulls), last response(cows)
36:       for Peg that wasn't checked do
37:         Use the new color to replace the old color
38:       end for
39:       return the changed code
40:     else
41:       for each saved state  $\in$  Saved responses do
42:         Compare the number of bulls and cows and determine the correct pegs
43:       end for
44:       if FirstLast then
45:         If first or last peg found replace the other
46:       end if
47:       constraint elimination()
48:       update All codes attempted()
49:       Reset the variables used for the peg checking cycle
50:       return guess  $\leftarrow$  generate random code()
51:     end if
52:   end if
53: end if
```

---

### 3.1 Color Checking

Lines 1-21 is the color checking stage. With the exception of ABColor where the colors are already known, our agent first makes monochromatic guesses until it knows which color is in the secret code and how often. In other words, our agents makes a monochromatic guess and adds up the total number of bulls and cows as the total occurrence of the color in the secret code. The color and its occurrence is stored in the dictionary, *AllColors*. This is similar to the strategy of B3, however, instead of checking colors in order, color checking is done randomly and stops once the occurrence of each color in *AllColors* adds up to the board length, at which point, we have *all colors found*. This helps to avoid making useless guesses. The function *make guess()* (Algorithm 1) determines when to make a monochromatic guess, upon which it calls *guessColors()* (Algorithm 2).

---

**Algorithm 2** *guessColors(colors, scsa, board length)*

---

```
1: random color  $\leftarrow$  select(colors)
2: Generate guess consisting of random color of length board length
3: if random color  $\notin$  AllColors then
4:   return guess
5: else
6:   return guessColor()
7: end if
```

---

### 3.2 Generate Random Code

If *all colors found* and there are no SCSA specific strategy called, our agent will generate a random guess. *generate random code()* makes use of found colors and constraints to randomly choose a color for each peg. This technique removes the need for computing all the permutations which lead to large time savings and improved scalability. It also allows for dynamic information updates which shrink the search space as we discover more information.

---

**Algorithm 3** *generate random code(colors)*

---

```
1: constraint elimination()
2: for Each found peg do
3:   Remove it from all colors
4: end for
5: for Each peg do
6:   if Peg not found then
7:     Choose a random color from the constraints and add it to the new guess
8:     if Constraints =  $\emptyset$  then
9:       Choose a random color from the all colors and add it to the new guess
10:    end if
11:   else
12:     Add to new guess
13:   end if
14: end for
15: if The new guess never occurred before then
16:   Return new guess
17: else
18:   Return generate random code(colors)
19: end if
```

---

### 3.3 Constraint Elimination

Throughout the game, the agent will store information it knows about the secret code. *constraint elimination()* uses found colors and saved attempts with zero bulls to decrease the size of possible colors for each peg. It aids *generate random code()* by decreasing the possible search space and converges onto a correct code.

---

**Algorithm 4** *constraint elimination()*

---

```
1: for Each peg do
2:   if Color found then
3:     Remove it from all colors
4:   end if
5: end for
6: for Each color do
7:   if All occurrences of that color are found then
8:     Remove it from all constraints
9:   end if
10: end for
11: for Each attempt do
12:   if Bulls = 0 then
13:     for Each constraint do
14:       Remove each color from the constraint
15:     end for
16:   end if
17: end for
18: for Each peg do
19:   if Peg found then
20:     Initialize constraint to that color
21:   end if
22: end for
```

---

### 3.4 Update All Codes Attempted

*update All codes attempted()* removes bulls from prior attempts by the use of the newly discovered pegs. It helps *constraint elimination()* by providing new information whenever it becomes available.

---

**Algorithm 5** *update All codes attempted()*

---

```
1: for Each attempted code do
2:   for Individual color in the code do
3:     if Color was found then
4:       Remove it from the bulls for that attempt
5:     end if
6:   end for
7: end for
```

---

### 3.5 Update Deciphered From Constraints

*update deciphered from constraints()* updates found pegs when a constraint for a given peg converges onto a single color.

---

**Algorithm 6** *update deciphered from constraints()*

---

```
1: for Each constraint do
2:   if Constraint contains one color and given peg wasn't found then
3:     Update the peg with that color
4:   end if
5: end for
```

---

## 3.6 Strategies for Specific SCSA

### 3.6.1 ABColor or TwoColor

In the cases where SCSA is ABColor or TwoColor, the function *make guess()* defers peg checking to the function *twoColorProblem* (Algorithm 4). This function starts with a monochromatic guess of *max*, the color with the highest occurrence. This monochromatic has been made initially, so our agent can retrieve its response from *All codes attempted*. Our agent will store this guess as our best guess in a dictionary called *Saved responses*. It tend proceed to change the first peg to *other*, the

other color in the code. We record a variable *counter* to keep track of the last peg we checked. After the first guess, our agent will repeatedly call function *twoColorProblem*. At each call, the agent compares the number of bulls in *last response* to the last guess recorded in *Saved responses*. If the number of bulls increase, we found the right peg by changing the color. Else, we need to change back to the previous color. We then continue the process by changing the next peg.

---

**Algorithm 7** *twoColorProblem(board length, last response, max, other)*

---

```

1: if Saved responses =  $\emptyset$  then
2:   Saved responses  $\leftarrow$  current guess
3: else
4:   if last response(bulls) < Saved responses[lastEntry](bulls) then
5:     current guess[counter]  $\leftarrow$  max
6:     counter  $\leftarrow$  counter + 1
7:   else
8:     Saved responses  $\leftarrow$  current guess, last response
9:     counter  $\leftarrow$  counter + 1
10:  end if
11: end if
12: if counter < board length then
13:   current guess[counter]  $\leftarrow$  other
14: end if
15: return current guess

```

---

### 3.6.2 TwoColorAlternating or mystery5

Whenever the agent encounters TwoColorAlternating or mystery5 it finds the two occurring colors and then generates two possible alternating codes.

---

**Algorithm 8** If statement within *make guess()*

---

```

1: Use the two found colors to generate two possible patterns
2: if Remainder of board length/2 is not 0 then
3:   Add remaining peg
4: end if
5: Store first code
6: Return second code

```

---

### 3.6.3 mystery2

When agent detects the mystery2 SCSA it finds the three colors and generates the six possible codes.

---

**Algorithm 9** If statement within *make guess()*

---

```

1: Use the three found colors to generate six possible codes
2: if Remainder of board length/3 is 1 then
3:   Add the remaining peg
4: else if Remainder of board length/3 is 2 then
5:   Add the two remaining pegs
6: end if
7: Store five generated codes
8: Return first code

```

---

## 4 Evaluation

### 4.1 Player VS Baseline 3

Table 1 shows that dinosaur player can run extremely well on 8-10 and wins 100 out of 100 rounds in less than 1 second. The player runs the fastest for TwoColorAlternating. Table 2 shows that the baseline player B3 can only run at 5-6. B3 attempted to play a 5-7, 6-7, 6-8, 7-9, and 8-10 game but it was extremely slow and was ended after 5 minutes of running. As mentioned in earlier sections,

dinosaur(), 8-10, 100 rounds		
SCSA	Wins	Time
InsertColors	100	0.1145
UsuallyFewer	100	0.0579
TwoColor	100	0.0208
ABColor	100	0.0119
TwoColorAlternating	100	0.0108
OnlyOnce	100	0.1370
FirstLast	100	0.1006
PreferFewer	100	0.0343

B3(), 5-6, 100 rounds		
SCSA	Wins	Time
InsertColors	100	6.9941
UsuallyFewer	100	2.5262
TwoColor	100	1.6088
ABColor	100	1.0237
TwoColorAlternating	100	1.4789
OnlyOnce	100	19.0548
FirstLast	100	4.1653
PreferFewer	100	1.4717

Table 1: Table 1 shows the number of wins out of 100 and the time it takes to run each SCSA on 8-10 for dinosaur player.

Table 2: Table 2 shows the number of wins out of 100 and the time it takes to run each SCSA on 5-6 for baseline 3 player (B3).

the only strategy that B3 depends on is eliminating choices that do not correspond to the color distribution of the last guess. Considering that it has to first generate all the choices, followed by identifying which choices from the list of all choices do not hold the same color distribution, and finally trying out each choice in order with no other strategy, it explains why it is not scalable.

## 4.2 Scalability

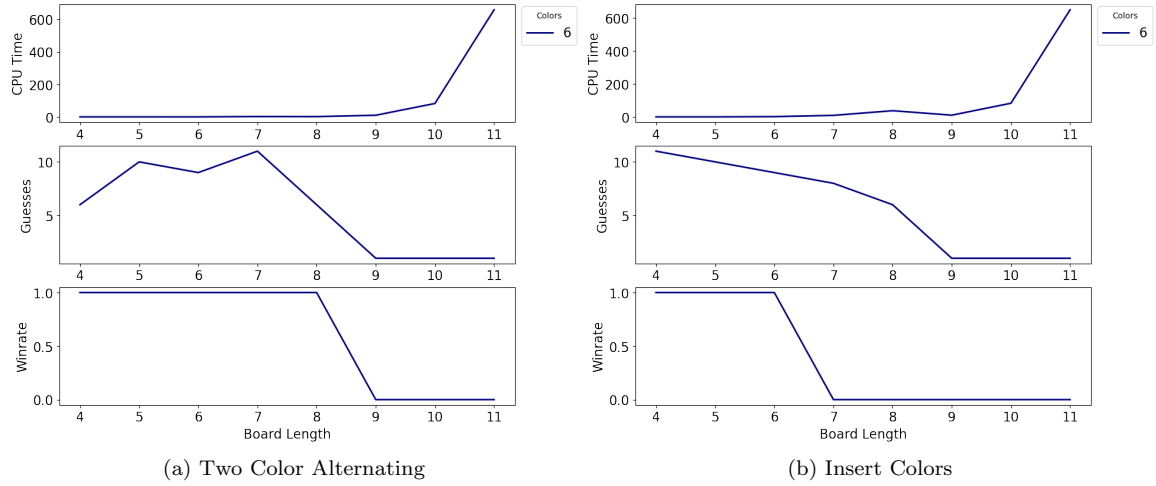
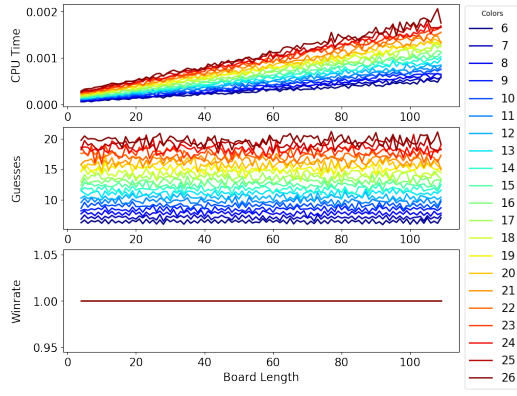


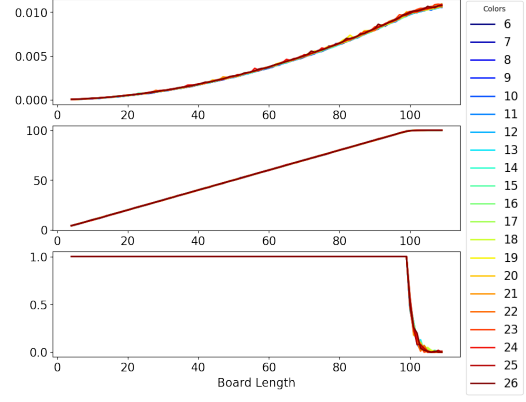
Figure 3: Performance of B3 against TwoColorAlternating and InsertColor SCSAs. Each graph contains three sub-graphs that correspond to an average CPU time per round, average guesses per round, and win-rate per tournament

B3 scalability is extremely poor as can be seen in Figure 3. Playing against the easiest and the hardest SCSAs produced very similar results. CPU Time goes beyond 600 and the agent fails to make more than one guess. Another important thing to mention is that when B3 reached 11-6 RAM usage on the test machine went all the way to 46 GB. We didn't see any need to perform this test on all SCSAs due to the fact that the agent doesn't discriminate between them.

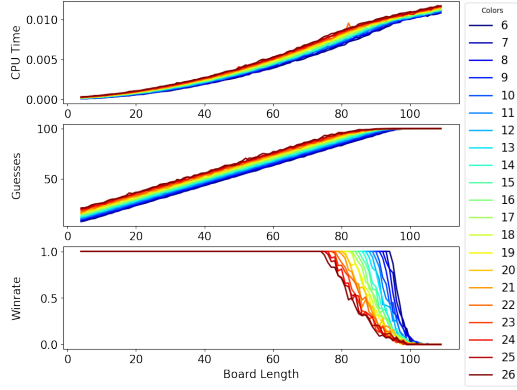




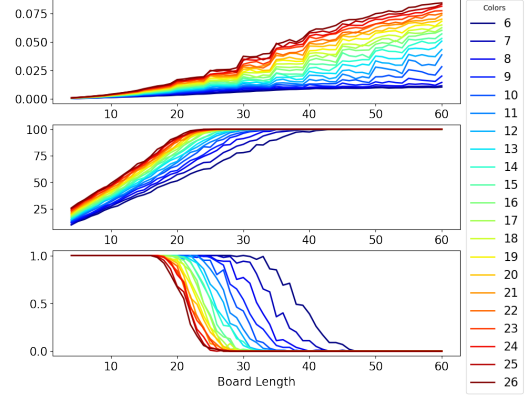
(a) Two Color Alternating



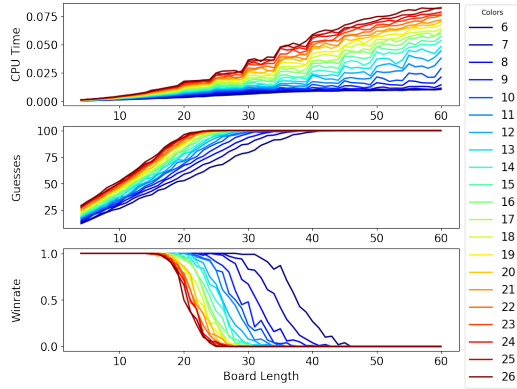
(b) AB Color



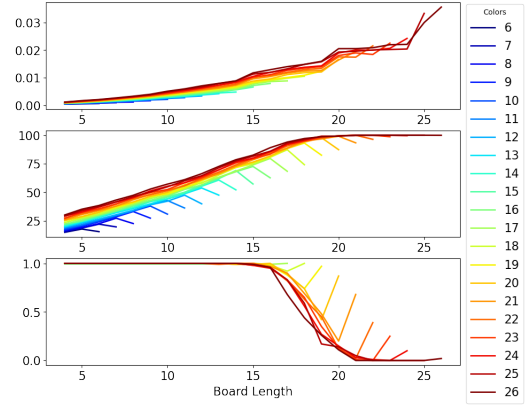
(c) Two Color



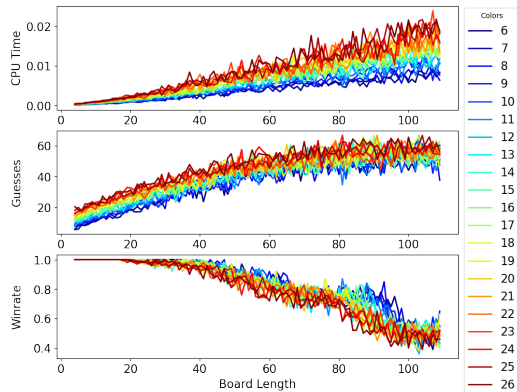
(d) First Last



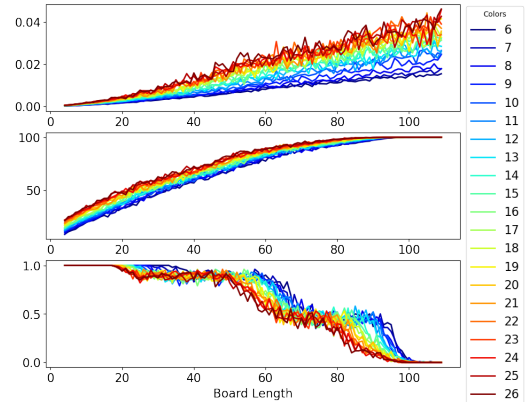
(e) Insert Color



(f) Only Once



(g) Prefer Fewer



(h) Usually Fewer

Figure 4: Performance of individual SCSAs. Each graph contains three sub-graphs that correspond to an average CPU time per round, average guesses per round and win-rate per tournament

## 5 Discussion

### Guess Construction:

Whenever the agent receives a code that cannot be associated with any strategy it finds all the color occurrences within the secret code first. Then, it uses the newly discovered colors to determine whether the code contains two or more colors. If the secret code contains only two colors, the agent uses a monochromatic state that consists of one color and it uses the second color to replace individual pegs. The pattern is revealed once the agent performs peg substitution for all of the pegs in the secret code. (Figure 1) However, if the code consists of more than two colors the agent takes a different path (Figure 2). In the first stage, it needs to generate a state that contains high enough bull count in order to make peg checking worthwhile. Once it finds a suitable state (Algorithm 1, line 28), it initializes a peg checking sequence which consists of two parts. Lines 34-50 in Algorithm 1 describes the peg checking sequence. First, the agent replaces already found pegs with the color that it will be performing a peg check with. This is the color that occurred most often as recorded by the dictionary *AllColors*. This ensures that whenever a peg was found the peg checking cycle will skip it in order to check only the relevant pegs. Then, the agent changes individual pegs and changes the found pegs to the original ones. At the end the agent sends the new state to the game environment to obtain information about it. The *Counter* variable ensures that the agent keeps track of the peg checking position between the different states. Once all of the suitable pegs are checked, the agent compares the number of bulls and cows that occurred during the peg checking and extracts the correct pegs and removes the incorrect colors from the constraints. Once the peg checking cycle is complete, the agent resets the peg checking variables and resumes by looking for state that contains desired number of bulls. It is important to note that the agent remembers how many pegs it already found and uses that information to find a state that contains new undiscovered bulls.

### Performance:

As the theoretical analysis predicted, the agent performs very well when the colors are known or the number of colors is low. However, once the number of colors increase the agent needs more guesses to find the correct code. In the case of TwoColorAlternating the number of guesses depend only on number of colors (Figure 4, (a)). When the agent plays against ABColor (Figure 4, (b)) it knows that the colors are always 'A' and 'B' therefore the only constraint that limits its performance is the size of the board. If the agent faces TwoColor (Figure 4, (c)) it needs to find the colors first and then find the corresponding pattern of the code. That explains why we see a early drop of in win-rate when the color number is high and lower when the color number is significantly smaller. Once the number of colors increase beyond two the performance of the agent starts to decrease significantly. This can be observed in the Figure 4, (d and e) which correspond to FirstLast and InsertColor. These SCSAs use a large amount of colors and have a non predictable pattern except that FirstLast contains the same color in the first and last peg. Agent needs to find all of the colors first and then go through multiple cycles of peg checking in order to arrive at the correct code. As expected the agent starts failing when the color number is high due to the fact that it needs to find colors from a larger color selection as well as it needs to reduce a larger number of constraints for each peg. In the case of OnlyOnce (Figure 4, (f)) it can be observed that the performance decreases due to the fact that it is shifted to the left. However, there are few spikes that can be observed on the graph. These spikes in performance correspond to the fact that once the board length and color number are the same the agent doesn't need to look for the colors and starts the peg checking cycle right away. The last two SCSAs in the Figure 4, (g and h) contain a mixture of all of the previous codes. That explains why the agent gradually starts to loose. In the case of PreferFewer the agent never loses completely because the SCSA produces a monochromatic codes that the agent explores at the beginning of the round.

In the initial performance test the agent scaled well given a 100 guesses. In order to prove that it can go beyond that, we provided our agent with 300 guesses in order to see how well it does. The agent scales perfectly when it is challenged with a TwoColor SCSA which is an easily solvable code (Figure 5, (a)). When faced with a hard SCSA like InsertColor (Figure 5, (b)) the agent also scaled well. This proves that it can perform well on larger scale problems.

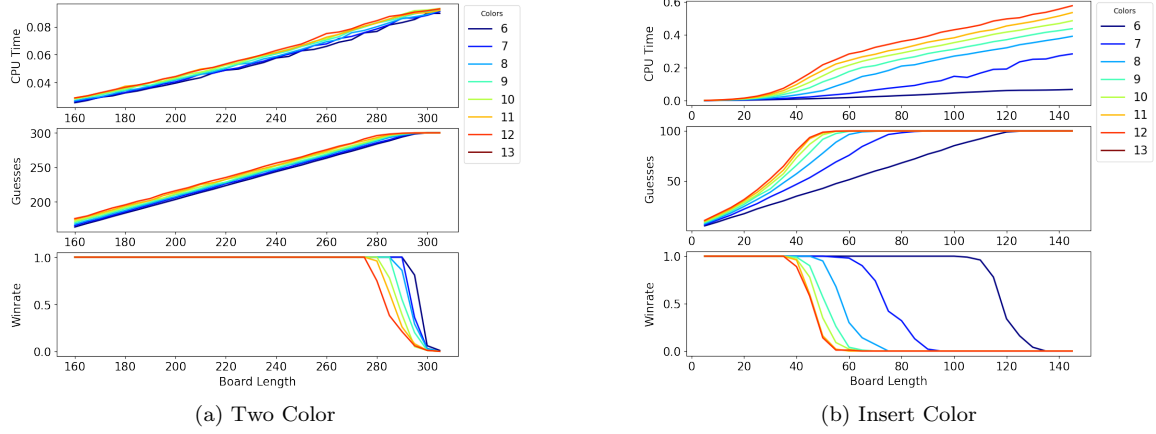


Figure 5: Agents scalability performance at 300 guesses. Test ran every three colors and every five rounds.

## 6 Theoretical Analysis

### 6.1 Search Space Analysis:

After implementation of our baseline players, it became clear to us that generating a complete game space and subsequent shrinking of that space after each guess will not scale. We performed a basic analysis on when the player would lose due to the round limit. After 6-8 the player would run out of time (Table 3) and the same thing can be observed after 5-21 (Table 4).

Table 3: Permutation time for a game with 6 unique colors.

Time (s)	Board Size
0.008	5
0.022	6
0.164	7
1.426	8
10.551	9

Table 4: Permutation time for a game with 5 unique colors.

Time (s)	Number of colors
1.655	18
2.169	19
3.031	20
3.908	21
5.141	22

In order to extend the capabilities of our agent we decided to explore the strategy that would find all the colors first. Finding colors at the beginning of the round would allow for significant reduction in possible outcomes and that would lead to a possibility of producing a usable search space at higher numbers of colors and pegs. Our test focused on the worst-case scenario in which we assumed that each color occurs only once. Game of size 9-9 was the last size that this strategy was able to use before it exceeded the 5 second mark.

Table 5: Permutation time for the game given the colors are known.

Time (s)	Number of colors	Board Size
0.0	6	6
0.417	9	9
5.396	10	10

### 6.2 SCSAs Complexity Analysis

Codes that all of the SCSAs produce can be divided into three categories:

1. Simple code; ex: 'OOOOOOO', 'NGNGNGN'
  - Codes are made up of one color or two regularly repeating colors

- SCSAs that fall into this category: Two Color Alternating, Prefer Fewer
2. Semi complex codes; ex: 'ABBABBB', 'ZHHZHZZ'
    - Codes are made up of two colors that don't regularly repeat
    - SCSAs that fall into this category: Two Color, AB Color, Usually Fewer, Prefer Fewer
  3. Complex codes; ex: 'QKUCUUE', 'YYCYCYL', 'TGVZRPJ'
    - Codes are made up of more than two colors and don't regularly repeat
    - SCSAs that fall into this category: Insert Color, First Last, Usually Fewer, Prefer Fewer, Only Once

SCSAs contain one of the categories listed above or a mixture of them. That means that our agent needs to have an internal logic that starts from the easiest case and goes all the way to the hardest one.

### 6.2.1 Simple code theoretical performance:

Whenever a code maker produces a monochromatic or a two color that regularly repeats our agent should be able to win at any peg or color size. In the case of the monochromatic code the worst case scenario would involve checking all the colors before reaching the right one. If the code consist of two regularly repeating colors, the agent would have to check both patterns before finding a correct one. In the case of Two Color Alternating, our agent will always win under 100 guesses. This occurs because after finding the right colors, which must be done within 26 guesses, the agent only needs to make at most two guesses of the alternating patterns to determine the correct pattern.

### 6.2.2 Semi complex codes theoretical performance:

In this case when the code maker generates a code that contains two colors. As mentioned in the simple code analysis the worst case scenario occurs when the agent checks all available colors before it finds the correct ones. That means if the agent checks all 26 colors the remaining space for peg guess is equal to 74 attempts. That means that in the worst case scenario our agent should be able to win all games at 74 peg size. However, since the worst case scenario doesn't occur on average we suspect our agent to work well on the range between 75-85 pegs. It is important to point out that in the case of AB Color SCSA the colors are known and thus the agent has a full 100 pegs to decipher the secret code. It means we would suspect that our agent would work well up to 100 pegs.

### 6.2.3 Complex codes theoretical performance:

Once the secret code contains more than two colors solution the problem becomes much more complicated. A single pass through all of the pegs will only be able to remove only one a limited number of colors from the constraints and at the same time only few pegs will be found. That means that in order to find the correct solution for a peg that has 25 possible colors the agent needs to performer multiple peg checking cycle.

## 6.3 Peg checking cycle:

Peg checking is done in lines 34-51 in Algorithm 1. This cycle is initiated once a *desired number of bulls* is found.

### 6.3.1 Space shrinking analysis:

The number of possible codes after the color checking stage can be represented by the equation below, where  $n$  is the number of different colors in the code,  $b$  is the board length, and  $p$  is the number of pegs found.

$$f(n, b) = n^{b-p}, b \geq p$$

This is well-defined since if  $p = b$ , i.e. all pegs have been found, then there's only one choice of guess. In the worst case,  $n = 26$ , but otherwise, since our agent first checks for the colors in the code,  $n < 26$ . Meanwhile,  $b$  can range from 4 to 100. Our agent performs the peg checking cycle as its strategy for complex SCSAs since finding just one peg can reduced the search space by  $n$  fold.

### 6.3.2 Determining the Desired Number of Bulls

The problem, however, is having to find the secret code within 100 guesses. In the worst case of color checking, the agent has to complete the task within 100-26 guesses. We argue that it is more optimal to require the agent to perform peg checking only when more than one bulls has been found in the random guess.

Let  $G$  be the number of guesses before finding the solution. Let  $c$  be the total number of color guesses,  $m$  the number of times peg checking is performed,  $b$  the board length,  $k < b$  the number of bulls found to initiate peg checking, and  $r_i$  the number of random guesses before finding at least  $k$  bulls to initiate the  $i$ th peg checking. In the worst case where peg checking only find  $k$  more correct pegs, and assuming that we fix  $k$  throughout the game, we have the following equations.

$$G = c + \sum_{i=1}^m r_i + \sum_{i=1}^m (b - ik) \quad (1)$$

$$G = c + \sum_{i=1}^m r_i + mb - \frac{km(m+1)}{2} \quad (2)$$

We can approximate a value for  $r_i$  using the expected number of guesses until finding one with at least  $k$  bulls. Let  $\beta$  be the number of bulls in the guess. Then

$$P(\beta < k) = \sum_{j=0}^{k-1} \frac{(n-1)^{b-j}}{n^b} \quad (3)$$

$$P(\beta \geq k) = 1 - \sum_{j=0}^{k-1} \frac{(n-1)^{b-j}}{n^b} \quad (4)$$

Now let  $r$  be the number of guesses until we find a guess with at least  $k$  bulls. Then by equation 3 and 4,

$$P(r = x) = \left( \sum_{j=0}^{k-1} \frac{(n-1)^{b-j}}{n^b} \right)^{x-1} \left( 1 - \sum_{j=0}^{k-1} \frac{(n-1)^{b-j}}{n^b} \right) \quad (5)$$

Equation 5 shows that  $r$  has a geometric distribution. Therefore, the expected value of  $r$  is

$$\frac{1}{1 - \sum_{j=0}^{k-1} \frac{(n-1)^{b-j}}{n^b}} \quad (6)$$

Also, consider that if  $k$  is fixed throughout the game, then the number of peg checking cycles we need is  $m = \lfloor \frac{b}{k} \rfloor - 1$ . Therefore, we can approximate equation 2 to

$$G = c + \frac{\lfloor \frac{b}{k} \rfloor - 1}{1 - \sum_{j=0}^{k-1} \frac{(n-1)^{b-j}}{n^b}} + (\lfloor \frac{b}{k} \rfloor - 1)b - \frac{k(\lfloor \frac{b}{k} \rfloor - 1)(\lfloor \frac{b}{k} \rfloor)}{2} \quad (7)$$

Clearly,  $G$  will be at its minimum when  $k = b$ , but that would simply mean we found the solution. Instead, we need to find some  $k$  that would shrink the last three terms. Observe that by equation 4, the probability that  $P(\beta \geq k = 1)$  increases especially as  $n$  and  $b$  increases. In other words, the probability of getting just at least one bull is greater as board length increases. Observe, however, that there is a trade off between terms. If  $k$  is small, then the second term should be small, but then the third and fourth term could be large. Indeed we show that it is best for  $k > 1$ ,

In order to increase  $1 - \sum_{j=0}^{k-1} \frac{(n-1)^{b-j}}{n^b}$ , we need to decrease  $\sum_{j=0}^{k-1} \frac{(n-1)^{b-j}}{n^b}$ .

$$\sum_{j=0}^{k-1} \frac{(n-1)^{b-j}}{n^b} = \frac{(n-1)^b}{n^b} \sum_{j=0}^{k-1} \frac{1}{(n-1)^j} \quad (8)$$

$$= \frac{(n-1)^b}{n^b} \left( 1 - \frac{1 - \frac{1}{(n-1)^{k-1}}}{1 - \frac{1}{n-1}} \right) \quad (9)$$

$$= \frac{(n-1)^b}{n^b} \left( 1 - \frac{(n-1)(1 - (n-1)^{1-k})}{n-2} \right) \quad (10)$$

To decrease equation 10 means to decrease the term  $(n-1)^{1-k}$ . Therefore, we need  $k > 1$ .

### 6.3.3 Heuristic

Having the above, we know the agent should acquire more bulls before initiating peg checking. This number of desired bulls depends on the board length. The formula

$$k = \frac{b - p}{5} - 1 \quad (11)$$

where  $b$  is the board length and  $p$  is the number of pegs with the right color found, stems from a bit of testing different values. Compare to equation 7, our agent varies the value of  $k$  as the remaining number of missing pegs changes. For example, if we have a board length of 25 and no correct pegs are found we use our generator to find at least 5 bulls. That means that in the next cycle we need to find at least 4 bulls, and the next cycle would require even less bulls. But since we shrink our possibilities and remove constraints we are more likely to get a higher number quicker. That means the most important thing is to have a heuristic that optimizes the first random generation of the state since it is the most complex and can take precious space from peg checking.

### 6.3.4 Further Comments

From equation 7 of our analysis, we observe merits from our approach. By first performing color checking, we find the value of  $n$ . If we are able to further control the values of  $c$  and  $r_i$ , we could bound equation 7 to be below 100, which would require further study.

## 6.4 Color Checking

### 6.4.1 Use of probabilities:

The intention of the probability analysis is to see if the SCSA algorithms have a bias towards a certain distribution for colors or order. The verifiable SCSAs all generate randomly with some limitations, and can not be exploited, but will be described for the sake of completeness.

InsertColor is defined by placing random colors in a random order. FirstLast is a subset of InsertColor with the limitation that the first and last are the same colors. TwoColors consists of two steps, the first is generating two different random colors in two random spots. Then it randomly fills up the rest of the board with one of the two colors. ABCColor is a subset of TwoColors but only uses A and B. TwoColorAlternating is another subset of TwoColors, but is limited to alternating the two colors. UsuallyFewer has a 90% chance of using 2 or 3 colors, and a 10% chance of becoming InsertColor. PreferFewer is a variant of UsuallyFewer where 50% of the time it uses 1 color, 25% it uses 2 colors, 13% uses 3 colors, 8% uses 4 colors, 3% to use 5 colors, and 1% to use all. If the total number of colors is less than what the SCSA decides, it uses the total number instead.

As defined by our heuristic, the approach for these SCSAs are solved as described in Figure 1 and 2.

Mystery Code 1 seemed to be a random assortment however, with a larger number than expected of only a single color and double. First plotting out the distribution of colors across, it is roughly uniform meaning there is no hidden color preference. In addition, there does not seem to be any special rule to the color pattern with respect towards alternation or repetition. Next, the density of the distribution of unique letters was plotted and the two was compared to InsertColors, the fully random SCSA. However, extremely evident was the distributions between InsertColors and Mystery Code 1 do not match. This is true for Mystery Code 4 as well.

After sampling 100,000 from InsertColors only 0.04% of the codes consisted of 1 color vs 52% for Mystery Code 1 vs 24% for Mystery Code 4. 2 colors was likewise over represented consisting of 3.8% of InsertColors codes but 42% for both Mystery Code 1 and 4. Conversely, 3, 4, and 5 color were underrepresented. It is clear that the distributions are not mostly random and are significantly biased towards fewer colors.

However, given this, there is not enough data to form a good heuristic specifically tailored for the Mystery Codes. One possibility might be there is some probabilistic penalty for each additional color above a certain number. Perhaps the probabilities are inverted such that for the probability of a code have  $n$  unique colors, it is swapped with the probability for  $N-n$  unique colors (where  $N$  is

Table 6: Comparison of Unique Letters Per Code Distribution

	1	2	3	4	5
InsertColors	0.04%	3.83%	31.36%	49.85%	14.92%
Mystery Code 1	52%	42%	0%	5%	1%
Mystery Code 4	24%	42%	28%	6%	0%

the maximum available color). Lastly, while in the sample mystery codes 1 and 2 colors were over represented, we do not know if any or all of the following are true: (1) those two particular number of unique colors are special, (2) if it is necessarily similar to prefer fewer SCSAs where fewer colors are special, and (3) if scaling the number of colors and board size will influence it.

Thus a minor improvement is to always try 1 and 2 colors for Mystery Code 1 and 4 first.

Mystery Code 2 has two interesting properties. As usual the color distribution test was uniform so that no specific color is more preferred than another. The first property is that in every single case, there is always 3 unique colors. The second is that there are no repeated consecutive colors (ie "AA..." is invalid).

The first property cannot be successfully exploited in that the number 3 is likely with some relationship with the color length of 7, but whether that is expressed as  $\text{floor}(\frac{N}{2})$  or some other variation like no matter the given N, only 3 unique colors are ever used. However, the second property can be exploited if we assume that no two letters can be repeated. Given the original state space of  $N^M$  (where M = board length), this reduces it to  $N \cdot N^{M-1}$  which is an improvement by a factor of  $\frac{N}{M}^{M-1}$ .

Mystery Code 3 is the only mystery code where there is a bias for certain colors. It is expected the colors to be uniformly distributed across the maximum colors (in the mystery code from A-F). The codes only cover ABC leaving out the other four colors. Additional analysis reveals that it always follows the pattern of 1 X/2 Y/2 Z in a random order. In other words it is not possible for a code to be "AAABC" since that would fall into the pattern of (1,1,3).

If the pattern is just the first piece of having 3 colors, the sample space shrinks to  $5^3 = 125$ . Given the second constraint where the frequency pattern is (1,2,2), we can further express the sample size for the 5-7 case as  $\binom{5}{1}\binom{4}{2}\binom{2}{2} \cdot 3 = 90$ . This is contrasted with the full sample space size of  $5^7 = 78125$  which is nearly 870x larger. However, it is unclear once again how this type of pattern scales. For larger boards and larger number of colors, we are unable to extrapolate. For example for the 6-7 would it be still 3 colors in the pattern of (2,2,2) or might it be 4 colors and a pattern with (1,1,2,2).

Mystery Code 5 seems much like TwoColorAlternating, at first glance, where two colors would alternate. As each sample code only had two colors, it was not feasible to search potential properties such as frequency. However, one property was to see if there was an underlying pattern to the distance between the characters. A list was generated of the distance between the two letters (ie "AC..." would be 0-2 = -2) and the density was plotted. Since there had to be two values, there was nothing at zero. The density on both sides (from zero) were roughly the same with a mean at -0.53 and a median at -1.

The first test was for normality. Given the density graph, we thought there was a possibility it could be normal. The quartile-quartile plot at glance suggested that it might be normal but it was unlikely with the heavier tails. The biggest hurdle was there was no zero. We first assumed the distribution was normal and artificially added a varying number of zeros and conducted Shapiro-Wilk tests. However, for significance levels of  $\alpha = 0.05, 0.10, 0.15$ , across a range of added data, the distribution was not normal. This test was also applied to samples from the known SCSA TwoColorAlternating which also failed the normality tests. Interestingly, this suggest that even with a random distribution, the "distance" was a poor proxy for normality or must be further modified.

However, since we knew that TwoColorAlternating did select colors randomly, we can next try to compare the underlying distributions. By conducting a two-sided Chi Squared test, we discovered

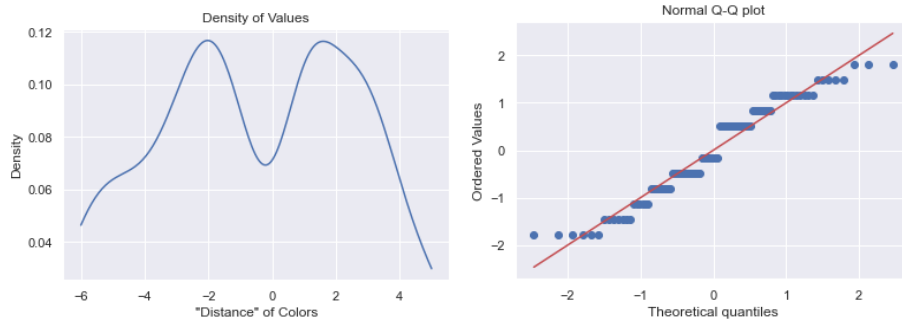


Figure 6: "Distance" as proxy for normality analysis

that with  $\alpha = 0.05$ , we cannot reject the null hypothesis that two are different distributions. We sampled TwoColorAlternating 100 times and completed 99 more tests and found that with  $\alpha = 0.05$ , 53% of the samples failed to reject, and with  $\alpha = 0.10$ , 74% of the samples failed to reject. Thus we were fairly confident that the SCSA for Mystery Code 5 was extremely if not identical to TwoColorAlternating. That meant we can simply use the same heuristic for TwoColorAlternating.

## References

- [1] D. E. Knuth, "The computer as master mind," *Journal of Recreational Mathematics*, Vol. 9(1), pp. 1–6, 1976-77.
- [2] T. W. L. Kenji Koyama, "An optimal mastermind strategy," *Journal of Recreational Mathematics*, 1994.
- [3] G. Ville, "An optimal mastermind (4,7) strategy and more results in the expected case," *arXiv*, pp. 1–20, March 2013.
- [4] "Mastermind," *from Wolfram MathWorld*.
- [5] pressmantoy, "Mastermind 2018 - how to play," *YouTube*, Aug 2018.
- [6] T. M. Rao, "An algorithm to play the game of mastermind," *ACM SIGART Bulletin*, pp. 1–5, October 1982.