# A Bulls and Cows Solver Using GA and WoC on Python

Adam Eisert, Lauren Mikula and Kieran Waigel, *Member, IEEE*

*Abstract*— **In this paper we test a combination of previous approaches with a new approach to solving an expanded version of Bulls and Cows. Bulls and Cows is NP-complete, with two main variants, digit or character based. In either case, methods for solving Bulls and Cows, or the similar variant Mastermind have involved beam search, backtracking, Genetic Algorithms, and Wisdom of the Crowds. The application of a hybrid Genetic Algorithm and Wisdom of Crowds approach provide the most promising results for larger data sets in the same format as Bulls and Cows.**

*Index Terms*—**Bulls and Cows, Genetic Algorithms, Wisdom of the Crowds.,**

## I. Introduction

THE Bulls and Cows (BaC) game is a two-player game with the objective of guessing an opponent's secret number or word. The essential rules are a player selects a secret number or word and the second player is tasked with guessing that number or word. When the second player's guess contains the correct letter or number, but in the wrong position, the first player informs them they have one "cow". When the second player's guess contains a letter or number in the correct position, they are informed they have a "bull." For example, if the secret word was "CAR" and the second player guessed "RAT", they would have one bull and one cow. "CAT" would have two bulls and zero cows. BaC is a non-polynomial (NP)-complete problem.

## II. Prior Work

The literature for BaC is minimal, with an originating paper from Tetsuro Tanaka [18] which was unfortunately not translated from Japanese. The game was later formalized by Aditya Garg [1]. A set of strategies were proposed by John Francis [4], and an exhaustive analysis of strategies to solve BaC was written by Andy Pepperdine [3]. More attention was given to the variant Mastermind when Donald Knuth published an algorithm to solve the game in an average of 5 steps [5]. The approaches to both are bound by the games size, and keep track of previous states, with the main difference being them focusing on either 1) Pruning a single set of answers, as described by Chen [2] and Guervos[7] with the k-way branching algorithm (KWB). KWB is a modified depth first search (DFS) where only k nodes are considered. To determine which k nodes are picked, solutions are clustered into groups via a hashing function based off the size of the search space. In traditional BaC, there are 14, 4 bulls, 3 bulls, 2 bulls and 2 cows, down to no bulls or cows. This approach has been generalized for deductive problems and works for both BaC and Mastermind. The mastermind solution has been improved over Rhodes [10] and Berghman[9]. The results of this algorithm are consistent with Tanaka's solution, taking 216 minutes to complete on a 2.8 GHz Pentium IV computer. The other type of approach 2) Dividing the set as described by John Francis [11], which takes several techniques, by first avoiding duplication by rejecting guesses which will not split the list of solutions. It then estimates the number of guesses needed to solve a subset and rejects a section if the number of guesses is greater than the limit of guesses. Paths which will require surpassing the maximum search depth are also rejected, and if no further options exist from a certain candidate, the next candidate is considered. This algorithm took just under 45 minutes to completely analyze BaC. This was allegedly consistent with Tanaka's performance in 1996, taking 90 hours to run with 1/60th the processing power. Francis found that the optimization of BaC was only possible due to increases in computing power as well as the combination of several optimization techniques.

## III. Proposed Approach

In order to solve the Bulls and Cows problem, a hybrid approach using a Genetic Algorithm (GA) in combination with a Wisdom of the Crowds (WoC) was implemented. Similar to the solution of a BaC variant from Yampolskiy [6], Bento [12], and Rao[13]. The traditional game of Bulls and Cows involves a secret number or word of 4 non-repeating characters. The game, however, can be played with more or less than 4 characters. For GA and WoC to be beneficial in solving an NP-complete problem, a certain level of complexity must be reached. Therefore, a secret key made of a large amount of characters that could be repeated was used. To simulate BaC, the user acts as player 1 by selecting a secret key. The program acts as player two, generates a random guess, and uses GA to optimize their guess based on the number of bulls and cows, and the results are fed into WoC.

We will describe in detail the two algorithms, GA and WoC. First, GA will be explained, including the specific fitness function, selection method, crossover method, and mutation method and how it was fed into the WoC algorithm.

In addition to BaC, the variant, similar to Mastermind, will also be tested with this same algorithm in order to compare effectiveness. These approaches will be differentiated with the labels non-duplicated and duplicated data sets respectively. Since the English alphabet does not contain enough characters for nontrivial problems, the characters used extended into other symbols used in UTF-8 character encoding, as long as no duplicate characters were used for non-duplicate sets.

### A. *Genetic Algorithm*

The genetic algorithm mimics the natural process of evolution to find more optimal solutions to problems[15]. The algorithm takes a population, selects individuals based on a fitness function, mates them together, and randomly mutates the resulting individuals in order to return a solution. These steps, fitness function, mating, crossover, and mutation, loop for as many generations as are desired, or until an individual with fitness equal to n is found. In this case, the GA was run for 2000 generations, with a population size of 100 to produce an individual to then be used in the WoC. Detailed explanations of these steps can be seen below:

1. Create a population. In this example, a population of 100 individuals are created. Each individual is the length n of secretKey. The first 10 individuals consist of pure strings of digits, where individual 1 is all 0's of length n, individual 2 is all 1's of length n, and so on. After 10 individuals, the next 26 are strings of characters a-z in a similar fashion. This was done to ensure that some individuals would have bulls, to work more effectively with the crossover and mutation function. After the first 36 individuals, the individuals are generated randomly with the randomize function, which creates random strings of length n, containing a combination of numbers, lowercase characters, and spaces.
2. Select individuals to reproduce. An elitist method was used based off individuals' fitness, where unfit individuals were unable to reproduce.
3. Cross over selected individuals. The previously selected individuals are combined to make two new children by selecting a random point and swapping every value from the right half of that point between the individuals.
4. Mutate individuals. A small percentage of the time an individual is mutated. This provides more diversity to the population.
5. The resulting children become the new population and is fed back into the GA.
6. After 2000 generations have been completed, or if the secretKey has been found, the GA terminates.

Further explanation into the fitness function, selection method, crossover method, and mutation method can be seen below.

### B. *Fitness Function*

After the population has been generated, these starting strings are sent to a fitness function that scores each string based on their relative fitness. To score the fitness of each string, the function weights each bull as 1, and each cow as 1/n. This was to ensure that cows retained some value, and has an added benefit that if a string is correct, it would be all bulls, with a fitness value equal to n. Each fitness is stored in a list, with each position relating to the list of individuals in the population.

### C. *Selection Method*

The algorithm then uses an elitist method to find n/2 pairs of mates. To ensure the most fit individual continues, the first 3 pairs are guaranteed to use the most fit individual, but after that, pairs are only selected from the 50% most fit individuals. Asexual reproduction was not included, so pairs of individuals with themselves are not accepted.

### D. *Crossover Method*

After pairs have been selected, the crossover must be performed. Since values can be repeated, this is the simplest operation in the algorithm. A random value is selected in the middle 80% of a pair, and the values to the right of that value are swapped between each individual, for every pair.

### E. *Mutation Method*

After each pair has been crossed over, the algorithm has the chance to mutate individuals. Two mutation functions were created. The base mutation occurs every two generations. It takes the population and rate of mutation, and randomly changes a number of values within the individual equal to that rate. We found it was more effective to mutate 3 values every 2 generations. The goal of this mutation function is to potentially introduce either a new bull or a new cow. This also helps the fitness of the individual not to plateau or be stuck in a local maximum. The second mutation function takes the population and mutation rate and randomly swaps two values within each individual a number of times equal to the mutation rate. It was also found to be effective to use a mutation rate of 3 every 5 generations. The goal of this mutation function is to potentially turn a cow into a bull. The implementation of both these mutation functions allows the algorithm to make the two types of improvements accepted in BaC: adding more cows/bulls, and turning cows into bulls.

### F. *Wisdom of the Crowds*

Once the GA has completed all runs, the resulting individuals are given to WoC as its population. The WoC algorithm mimics the wisdom of the crowd phenomenon. This phenomenon occurs when decisions are made by a group of people. When solving problems such as continuous point estimates of physical quantities, general knowledge, or multiple-choice questions, groups tend to find optimal solutions even if many individuals are not well-informed on the problem [8]. In other words, groups are often incredibly intelligent, more so than the most intelligent individual [14].

We use this idea to solve BaC. Each final guess of the GA is considered as the crowd. The solutions from the crowd are examined and aggregated in order to produce a more optimal solution. The general steps of WoC used after GA can be seen below:

1. Generate crowd. The crowd consists of the final

solutions of GA as described above.

2. Aggregate solutions. The solutions are aggregated into a matrix in order to visually and iteratively evaluate the solutions.
3. Combine solutions. The best aspects of the agreement matrix are combined to create a new, more optimal solution.

### *G. Agreement Matrix for WoC*

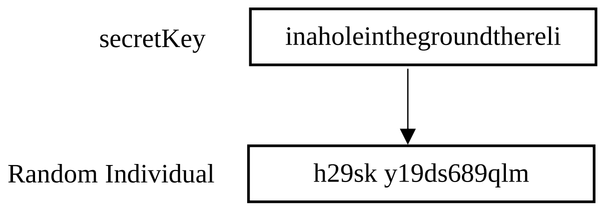
WoC uses the most common results from these individuals to find a new, more optimal solution. To do so, an m x n agreement matrix is created, where m is the number of characters in the secretKey and n is the number of possible characters. For the chosen language of this problem, the possible characters are lowercase letters, numbers 0 through 9, and a space. Each row in the agreement matrix then represents each position in of string with length m. Each column in the matrix represents one of the possible characters. The first 26 rows are used for the lowercase letters, the next 10 rows are reserved for the numbers, and the final row is reserved for the space character. Each cell in the agreement matrix represents how many individuals selected the column-character to be in the row-position of the string.

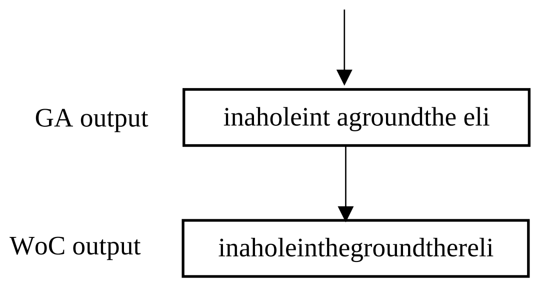
Once the agreement matrix is created, the maximum value for each row is selected. This means the most common character for a given position will be selected as the character in that position for the final string. After each maximum value in each row is found, the algorithm returns an individual.

## IV. EXPERIMENTAL RESULTS

Previous games relied on the structure of a fixed size to be able to use deductive reasoning to prune search trees and structure their guesses. Since that structure has been definitively solved, a slightly larger set is used. Instead of 4 digits, character strings of length 10, 25, 50, 100, 150, 200, 250, and 300 were used. Two separate types of data sets were given to the algorithms: (1) secretKey containing duplicated characters and (2) secretKey containing non-duplicated letters. Every secretKey for the duplicated characters data type were chosen from the initial lines of The Hobbit by J.R.R. Tolkien [17]. Only the baseband was used, without punctuation, spaces, and all uppercase characters were converted to lowercase. Every secretKey or the non-duplicated character data set came from a random combination of lowercase letters and symbols from UTF-8 character encoding. For example, for a string length of ten the duplicated secretKey was “inaholeint” and the non-duplicated secreyKey was “abcdefghij”.

To see the performance of the GA and WoC algorithms, an example of the process for a scan be seen below.





As depicted in this example, a secret key was chosen from the first few lines of The Hobbit. The chosen secretKey for 25-character string length can be seen in the first box of the example. Once the secretKey is determined, a random population with the same length as the secretKey is generated. The individuals in the population are a random arrangement of letters a-z, numbers 0-9, and space(s). An example of such individual is seen in the next box.

This key is given to the GA, it creates 100 random individuals and then executes the algorithm for 2000 generations. This is repeated five times. The best result of the five runs can be seen in the third box. This individual contains in 3 incorrect characters, or 22 bulls and 0 cows.

Each of the five individuals are then given to WoC as its population. WoC creates the agreement matrix and uses the maximum values to determine the new individual. This individual can be seen in the final box shown previously.

As clearly shown in this example, GA was able to return a more optimal solution than the randomly generated one. It gave an answer that resembles the secretKey, but it does not find it. Once all five GA solutions are given to WoC, and it found a solution with 25 bulls and 0 cows. In other words, the secretKey was found. Similar results can be seen for the remaining data sets. The results of which can been seen in the Table 1 where the fitness of each output is determined based on the number of bulls and cows in the string. The maximum fitness is equal to the length of the string. A similar table showing the results of the non-duplicated sets can be seen in Table 2.

*Table 1: GA vs WoC Fitness for Duplicated Set*

| String Length | Best GA Fitness | Worst GA Fitness | Average GA Fitness | WoC Fitness |
|---|---|---|---|---|
| 10 | 10 | 10 | 10 | 10 |
| 25 | 22 | 21 | 21.8 | 25 |
| 50 | 44.01 | 42 | 43.222 | 50 |
| 100 | 92.1 | 81 | 83.68 | 100 |
| 150 | 121.1 | 118.1 | 119.3 | 148 |
| 200 | 157.1 | 146.1 | 152.5 | 199 |
| 250 | 193.1 | 181.1 | 190.3 | 249 |
| 300 | 231.1 | 206.2 | 221.12 | 291.0066 |

*Table 2: GA vs WoC Fitness for Non-Duplicated Set*

| String Length | Best GA Fitness | Worst GA Fitness | Average GA Fitness | WoC Fitness |
|---|---|---|---|---|
| 10 | 10 | 10 | 10 | 10 |
| 25 | 24 | 10.04 | 19.06 | 24 |
| 50 | 43 | 33.16 | 38.51 | 49 |
| 100 | 76.1 | 42.4 | 61.6 | 93 |
| 150 | 103.2 | 90.2 | 98 | 140 |
| 200 | 116.3 | 89.4 | 107.5 | 184 |
| 250 | 133.3 | 116.4 | 125.34 | 200 |
| 300 | 135.4 | 117.4 | 130.3 | 221 |

For the smallest set the GA was able to produce an optimal solution. Further optimal solutions required the use of WoC, which performed optimally for sizes 10-100. The worst result of the WoC was for length of 300 characters with a 3% error rate, as opposed to the 23% error rate of the best GA which was a part of the crowd which produced this result. The comparison of each algorithm's fitness can be seen in Figure 1. Similar results can be seen in Figure 2 for the non-duplicated data set.

*Figure 1: GA vs WoC Fitness for Duplicated Sets*

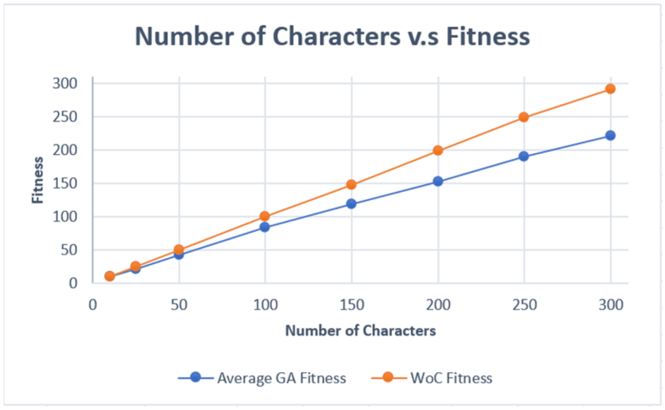
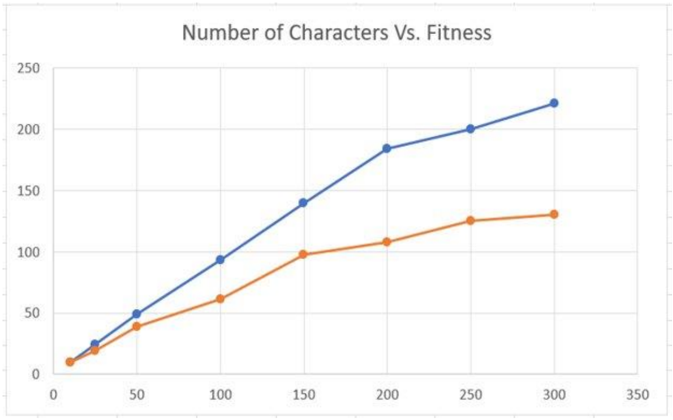


*Figure 2: GA vs WoC Fitness for Non-Duplicated Sets*



This shows the fitness of the algorithms. The GA is hinting at a logarithmic relationship between number of characters and the average fitness that can produce with the set variables of 2000 generations and population size of 100. Similar to the fitness of each solution, the miss rate, or number of incorrect characters in the individual compared to the secretKey can be seen in the Figure 3. Similar results for the non-duplicated sets can be seen in Figure 4.

*Figure 3 : GA vs WoC Miss Rate for Duplicated Sets*

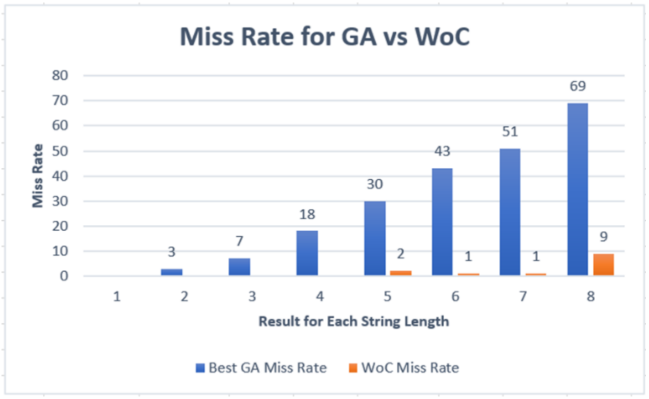
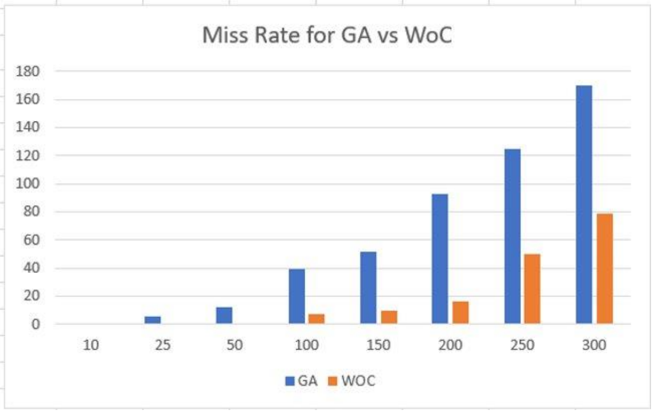
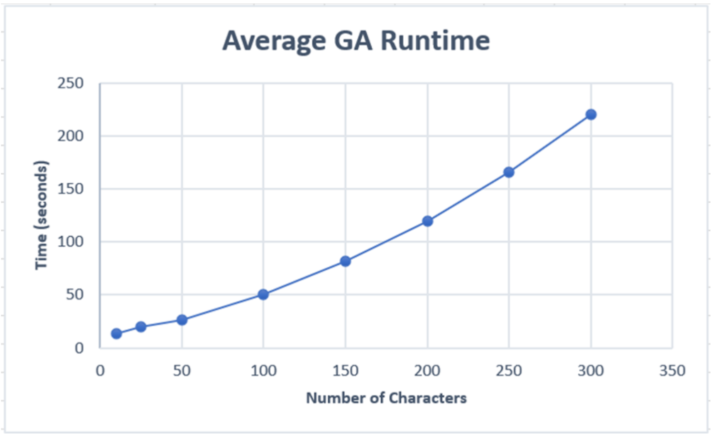


*Figure 4: GA vs WoC Miss Rate for Non-Duplicated Sets*



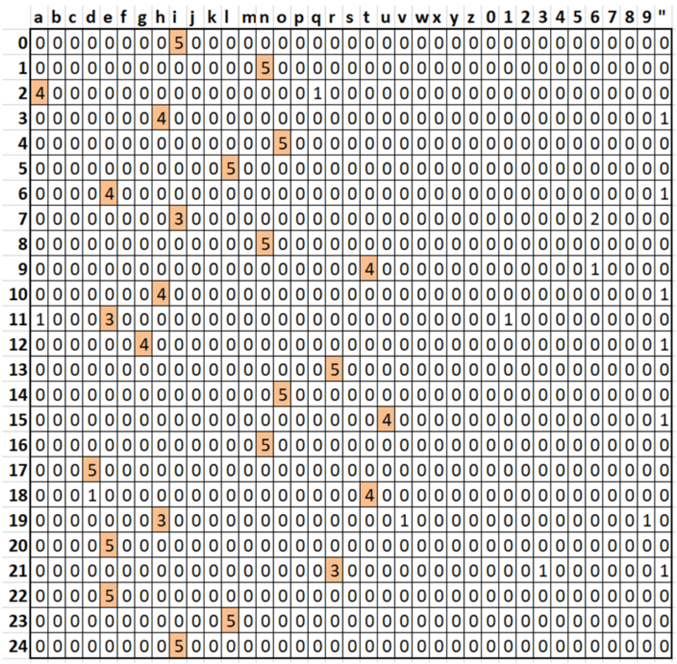
The final description of the GA is thefGraph displaying the length of the key vs. The runtime required to produce a single individual. A graph of the runtime for non-duplicates was not plotted because it follows a similar exponential curve, with approximately 15% more time spent per set. The runtime for WoC is also not shown since WoC only uses aggregated data generated by the GA. Once the data has been collected, WoC takes less than one second to run.

*Figure 5: Average GA Runtime for Duplicated Sets*



An example of the agreement matrix generated by WoC for the preceding example and how it selects each character can be seen in Figure 6. Each column of the matrix represents an accepted character. Each row represents a position in the string. So, each cell represents the number of individuals that selected the column-letter in the row-position of the string. The selected cells for the preceding example are highlighted in orange. The agreement matrix for the non-duplicated data sets is not shown. The only difference in this matrix would be more columns to account for the more characters that are accepted. So, for redundancy's sake, that agreement matrix has been excluded.

*Figure 6: Agreement Matrix for Duplicated Set*



| | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | " |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 11 | 1 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 17 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 18 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 19 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 20 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 21 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 22 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 24 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## V. Conclusions

The WoC algorithm was highly effective at creating accurate solutions from an inaccurate crowd. The runtime of the WoC algorithm was highly dependent on the desired size of the crowd and length of the input string.

Compared to the GA, the WoC algorithm ran significantly slower. This was because each member of the crowd submitted to the WoC was the result of a single GA execution. If the desired crowd size was 10, then the GA had to be run 10 times before the WoC could even begin, making the WoC 10 times longer to execute in this case compared to the GA. Once the crowd had been created the actual aggregation method of the WoC was insignificant in regard to run time.

Despite its slower performance, the WoC produced dramatically better results compared to the GA. The average inaccuracy rate of the WoC was 0.6% for strings up to 300 characters long and an inaccuracy rate of about 3% for a 300-character string. The WoC algorithm consistently guessed the exact string correctly for all runs with 100 characters or less. Since each character in a string could be any alphanumeric character including spaces, there were 37 possibilities for each character in the string. The 300-character string would have $37^{300}$- possible permutations. This is a substantial increase to the search space of traditional BaC, which uses only digits with no repeats for 5040 possible permutations.

A potential increase in the WoC algorithm's performance would be multiprocessing. If each iteration of the GA were to be parallelized then it would greatly reduce the runtime of the WoC algorithm compared to the GA. On an 8-core processor the WoC algorithm of up to size 8 could have approximately the same runtime as a single GA run [19].

In addition, an improvement to the fitness function could be made. The current fitness function makes a copy of the current individual and iterates through the individual. If the value at the individual's location matches, it increments bulls. If the value doesn't match, but exists within the individual, it is counted as a cow. In either case, the first value in the copied individual is deleted. This leads to inaccurate fitness values. It would be more accurate, but more expensive, to first go through the string marking each bull and removing them, then going through the string matching each cow and removing those values.

It was found that the largest factor impacting speed was the size of the population for the GA. WoC took less than a second to run on average, the largest bottleneck was just operating on individuals. However, smaller population sizes yielded less accurate individuals. There is a balance between the population size of the GA, and number of individuals needed to create an optimal solution for WoC, which can be determined to optimize speed.

Mutation rates, as well as frequency of mutation could be tweaked further. Initial trials found that the current setup of mutating every other generation with a small number of values was effective, but it might be better to increase the mutation rate as the population size increases. What works for a character string of 25 is not necessarily going to work for a character string of 800. Preliminary trials testing data sets 800 characters long showed an average of 20 minutes to generate a single individual. This could be due to worse mutation rates, or it could be due to the amount of time needed to operate randomly on strings of length 800. In addition mutation and crossover rates could be modified dynamically as the GA progresses. This could potentially maximize the benefit of a mutation early on but reduce the risk of a mutation removing an ideal solution later on in execution [20]. In either case mutation rates, frequency, population sizes, number of individuals, and fitness functions all have room to be experimented with, improved upon, and optimized.

## ACKNOWLEDGMENT

## REFERENCES

[1] Garg Aditya, Goel Namanyay. *A Mathematical Approach to Simple Bulls and Cows.* Delhi Public Schools. 2015.

[2] Chen S., Lin S., Huang L., Hsu S. *Strategy Optimization for Deductive Games.* European Journal of Operational Research 183. 2006.

[3] Pepperdine A. *The Game of MOO.* 2010.

[4] Francis J. *Strategies for Playing MOO, or "Bulls and Cows".* 2010.

[5] Knuth, D.E. *The Computer As Master Mind.* J. Recreational Mathematics, 1976. Vol. 9(1)

[6] Yampolskiy, R. Khalifa, A.B. *GA with Wisdom of Artificial Crowds for Solving Mastermind Satisfiability Problem.* 2011

[7] Guervos, J.J.M., Cotta, C. Mora, A. *Improving and Scaling Evolutionary Approaches to the MasterMind Problem.* 2011

[8] Berghman, L. Goossens, D., Leus, R. *Efficient Solutions for Mastermind Using Genetic Algorithms.* Computers and Operations Research, 2009. V. 36

[9] Ville, G. *An Optimal Mastermind (4,7) Strategy and More Results in the Expected Case.* 2013

[10] Rhodes, A.D. *Search Algorithms for Mastermind.* 2019.

[11] Temporel, A. Kovacs, T. *A Heuristic Hill-Climbing Algorithm for Mastermind.* 2004

[12] Bento, L. Pereira, L. Rosa, A. *MASTERMIND by Evolutionary Algorithms.*

[13] Rao, T. M. *An Algorithm to Play the Game of Mastermind.*

[14] Norvig, Peter. Russell, Stuart. *Artificial Intelligence A Modern Approach. 2010*

[15] Surowiecki, James. *The Wisdom of Crowds. 2004.*

[16] Yi SK, Steyvers M, Lee MD, Dry MJ. *The wisdom of the crowd in combinatorial problems.* Cogn Sci. 2012 Apr;36(3):452-70. doi: 10.1111/j.1551-6709.2011.01223.x. Epub 2012 Jan 23. PMID: 22268680.

[17] Tolkien, J.R.R., *The Lord of The Rings*

[18] Tanaka,T. *An Optimal MOO Strategy.* Game Programming, Kyoritsu Shuppan, Page 150-157. 1997.

[19] Shanthi, M. and A. Irudhayaraj. *Multithreading - An Efficient Technique for Enhancing Application Performance.* 2009.

[20] Hassanat, A., Almohammadi, K., Alkafaween, E., Abunawas, E., Hammouri, A., & Prasath, V. B. (2019). *Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach.* ,*10*(12), 390. 2019