Solutions to Mastermind using a Genetic Algorithm

Ryan Justus
CS 4750

ABSTRACT

I present a genetic algorithm for finding a solution in the game of Mastermind. The algorithm works by generating a populations via selection, crossover, and mutation of a current population until a population that contains the solution is found. The algorithm has a number of adjustable parameters, including the selection method, mutation rate, scoring parameters, and generation sizes. The quality of different configurations is measured by recording the number of generations, and in some tests time, until a solution is found.

1. INTRODUCTION

Mastermind is played by two people, one of which selects a secret code, the encoder, and the other who attempts to guess that code, the guesser. The code is a secret sequence, **s**, of N symbols. The symbols come from a set of characters, C, that is known to both players.

$\mathbf{s} = (s_1,s_2,...,s_n) \in C \equiv \{ c_1, c_2, \ldots, c_m\}^N$

The codebreaker then generates a guessed candidate sequence, $\mathbf{g}_{i..}$

$\mathbf{g}_i = (s_1,s_2,...,s_n) \in C \equiv \{ c_1, c_2, \ldots, c_m\}^N$, i=1,2,..

For each guess, $\mathbf{g}_{i,}$ the encoder tells the code breaker two numbers.

The first number, A_i, is the number of symbols that are in the correct place, $\mathbf{s}_\mathbf{p} = g_{ip}$. The second number, B_i is the number of characters that have the same symbol in both sets. If a component is re-used in the sequence $\mathbf{g}_{i,,}$ then it is only counted once for a unique matching of the symbol in **s**. For example if **s** has one occurrence of the symbol 'a' and g_i has two occurrences of 'a', then the matching score for that symbol is 1.

2. ALGORITHM

2a. Basic Algorithm:

The core of the genetic algorithm involves generating a new candidate population, G_{n+1}, from the current population G_n. For any population before the last we know that there is no solution, so our goal is to generate candidate solutions that have a good chance of matching **s**. New candidate solutions are generated by two-point crossover from the original population along with a chance for a single point mutation.[1]

```
Generate random code c;
Generate initial random population, G, of size N;
while c ∉ of G do
  1. Select top m candidates, S, based via tournament selection;
  2. Generate a set of new candidates, C where |C|=N-m, via two point crossover from randomly selected candidates, c1 and c2 in G, with a random single point mutation chance, p;
  3. Set G = S ∪ C;
end while;
```

2b. Tournament Selection Algorithm:

The purpose of the tournament selection is to create a new population that is better than the previous population. A series of 'tournaments' is held between pairs of candidates, with the winner being added to the next generation.

```
let n be the number of candidates we wish to select from a population $G_i$;
 set j=0;
 set $G_{(i+1)} = \emptyset$
 while j<n
   select two random candidates, $c_1$ and $c_2$ from $G_i$;
   set $G_{(i+1)} = G_{(i+1)} \cup \{score(c_1>c_2)?c_1:c_2\}$;
   set j=j+1;
 end while;
```

2c. Two point crossover algorithm:

We use crossover to generate new unique candidates for the next generation. Using a two point crossover prevents the middle of a candidate string from from being preferred over the endpoints.

```
Given two candidates $p_1$, $p_2$
let a,b be a random indexes in $p_1$;
if  a <= b
   c = $[p_{20}..p_{2(a-1)}]+[p_{1a}..p_{1b}]+[p_{2(b+1)}..p_{2n}]$;
```

```
else
    c = [p10..p1b]+[p2(b+1)..p2(a-1)]+[p1a..p1n];
end if;
```

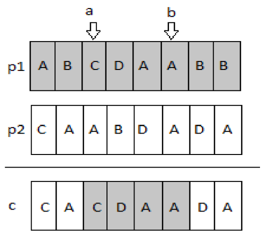


Illustration 1: Two point crossover, a<=b

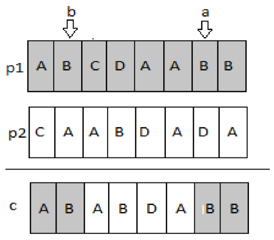


Illustration 2: Two point crossover, b<a

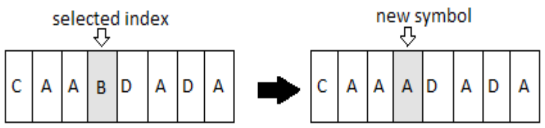
2d. Single point mutation algorithm:

If the initial population doesn't have enough diversity there is a chance that a correct solution will never be generated via crossover alone. Adding a chance for a single point mutation ensures that eventually a solution will be found.

```
 let c be a candidate string produced from
 a crossover, and $r_m$ be a constant in
 (0..1)
 Generate a random number r in (0..1)
 if r < $r_m$
   let a be a random index in c
   set $c_a$ = random symbol in C
end if
```

3. EXPERIMENTS

In our algorithm we have several unknowns, the score function, the chance of a single point mutation, r_m, the number of candidates to keep from the previous generation, m, and the number of candidates in each population, N. The code generation parameters can also be altered by adjusting the number of symbols and the length, but for these experiments we use a character set of size 4 with a sequence length of 10.



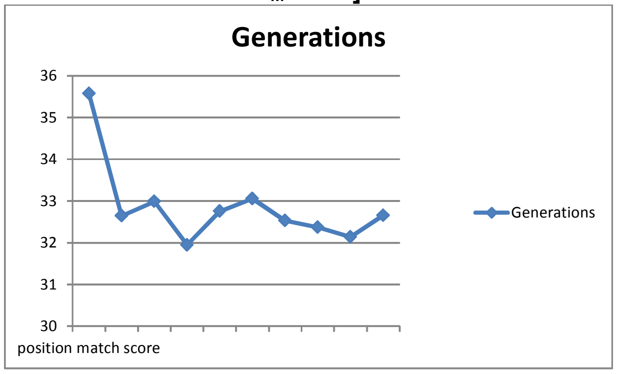
The following chart is a summary of the parameters used in the genetic algorithm.

N	Size of a generation
r_m	Mutation rate (%)
r	Survival rate (%)
r_0	Score weight for a position match
r_1	Score weight for a character match

3a. Score function experiment

The guesser is given feedback with two numbers, the symbol match count, r_0, and the position match count, r_1. A position match should be worth at least as much as a symbol match. To determine a good score function I set the population size, p, to 50 and the number to keep from each previous generation, n, to 25. The mutation rate is set at 15%. Our character set consists of 4 symbols, {'a','b','c','d'}, and the length of our puzzle is 10. For this experiment we will hold the symbol match value constant at 1 and alter the position match value from 1 to 10. The test is run 1000 times and the average number of generations until the solution is evaluated.

Plot 1: r_1 vs generations [r_0=1, N=50, m=50%, r_m=15%]



The minimum value was found at r_0 is found at 4, but everything 2 or above performed comparably. As an additional test I set r_0 to zero to see how much the position score mattered overall in the score. At $r_0 = 0$ it took ~6400 generations to find a solution, so a valid position count is essential to the success of the genetic algorithm. Alternatively setting the symbol match score constant, r_1=0 results in an average of 32 generations until a solution.

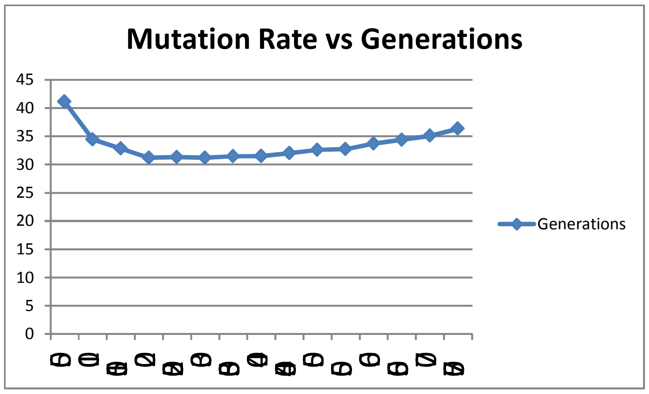
These result tells me that position correct positions is what matters most, and at least for puzzles with a small number of symbols and length using the symbol match doesn't affect the outcome.

3b. Mutation rate experiment

In this experiment I test different mutation rates to see how it affects the number of generations until a solution. After the crossover phase we have a chance to apply a single point mutation on the child. If there is no chance for mutation then it is possible that there is no combination of crossover that will result in a solution. As a trivial example if out character set was {'a','b','c'} with out code as ['c','a','b'] and all of our candidates only had the 'a' and 'b' characters, we would never find a solution. Since I am only applying the mutation to the result of a crossover I expect a fairly high mutation rate will work well.

For this test I used a population size of 50, with 25 surviving from the previous round via tournament select. My character set is {'a','b','c','d'} with a code length of 10. I altered my mutation rate 5% at a time and recorded the number of generations until a solution. The average of 1000 tests is recorded

Plot 2: mutation rate vs generations [r_0=1, r_1=3, N=50, m=50%]



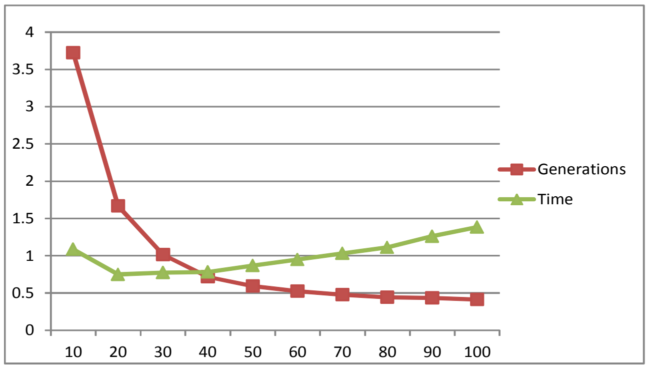
As predicted at low mutation rates the number of generations is higher. The minimum occurs at around 20% and stays fairly level with a slow increase through the rest of the samples.

3c. Population size

In this experiment I will compare population size vs the number of generations until a solution. Since a larger population will obviously result in fewer generations I also record the time it takes until a solution. My populations size varies from 10 to 100, increasing 10 at a time. For the each sample I plot the number of generations divided by the sum of the generations across all the sample points. I do the same for the time.

Tournament select is applied to each population with half the population surviving to the next round. The single mutation rate is set at 15%. The average of 1000 tests is recorded.

Plot 3: populations size vs generations and time [r_0=1, r_1=3, m_r=15%, m=50%]



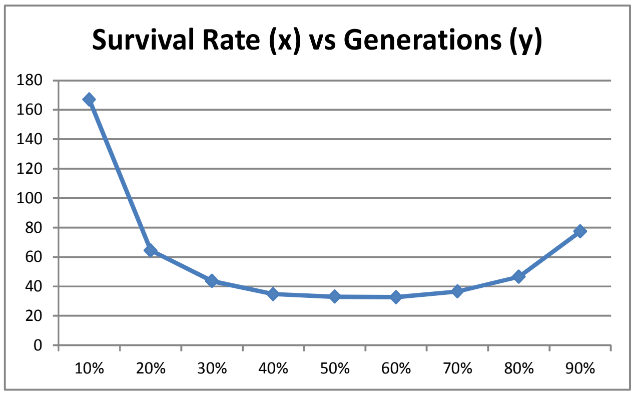
The steepest change in the number of generations verses the population size occurs from the range 10 to 50. After this point the relationship becomes more linear. Excluding the population=10 point the population vs time plot is fairly linear. Given these results a population size from 50-60 provides a good tradeoff between time and number of generations until a solution is found.

3d. Tournament select survival rate

In this experiment I adjust the survival rate from each generation. Each subsequent generation is composed of the a percentage of the current generation with the remainder comprised of new candidates formed by crossover and mutation. Keeping some of the candidates intact from generation to generation provides candidates that are closer to the solution than average, so the algorithm should move toward a solution faster than random candidate generation. On the other hand we know that each candidate we keep via selection is not a solution, so the more we keep the few new possible solutions we generate on any given round.

This experiment includes survival rates from 10 to 90 percent, increasing in increments of 10. My population size is 50 with a mutation rate of 15%.

Plot 4: Survival rate vs generations and [r_0=1, r_1=3, N=50, m_r=15%, m=50%]



As predicted high (>80%) and low (<30%) survival rates don't perform very well. Survival rates in the 50%-60% range seem to give the best results.

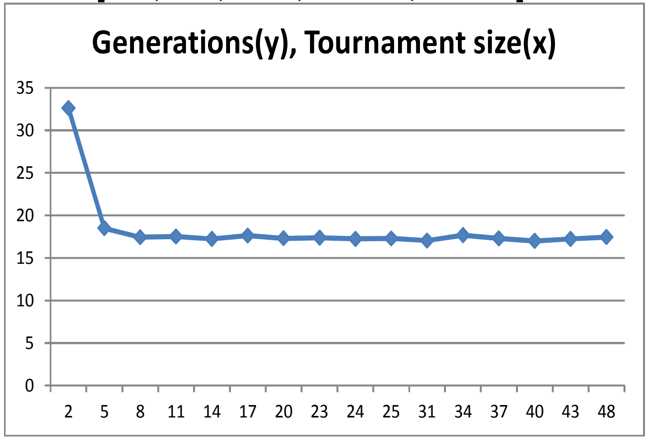
3e. Tournament selection algorithm

Thus far in the tournament selection algorithm we have selected two random candidates and kept the best one, repeating until we were at the desired survival rate. In this experiment we alter the number of candidates in each round of the tournament. Instead of picking two candidates to compete will pick k candidates from the best population and keep the best.

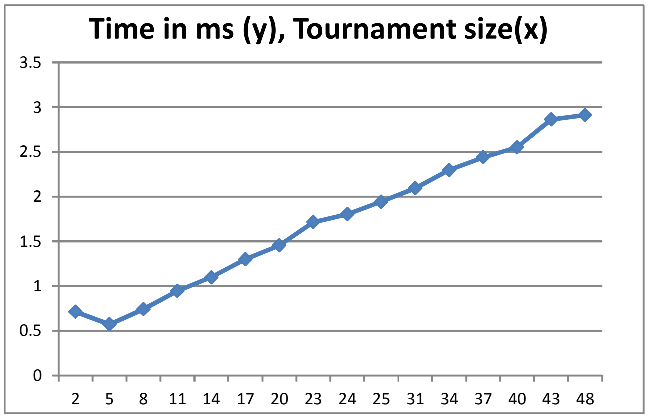
```
let n be the number of candidates we  wish
to select from a population $G_i$;
 set j=0;
 set $G_{(i+1)} = \emptyset$
 while j<n
   select k random candidates,$c_1..c_k$ from $G_i$;
   $G_{(i+1)}=G_{(i+1)} \cup \{max(score(c_1)..score(c_k))\}$;
   set j=j+1;
 end while;
```

The hypothesis for this is that only using two random candidates in the tournament doesn't give enough weight to best candidates for survival. If too many are included in each tournament, however, not as much diversity will be passed on to the next generation. For each tournament size I took the measured the average number of generations and also the average time in milliseconds over 1000 tests.

Plot 5: Tournament size vs generations and [r_0=1, r_1=3, N=50, m_r=15%, m=50%]



Plot 6: Tournament size vs time and [r_0=1, r_1=3, N=50, m_r=15%, m=50%]



The time increase is fairly linear with the increase in tournament size, with a tournament size of only two causing an increase corresponding to the extra number of generations for a solution. For tournament sizes larger than 5 there wasn't a decreased generation count, but the jump from a size two tournament to a size five tournament is significant. For the test parameters a tournament size of six gives the best performance in both time and generation count.

4. CONCLUSION AND FUTURE WORK

4a. Conclusion

Adjusting the parameters had a significant effect on the performance of the genetic algorithm. In general, however, not picking bad values is seems much more important than picking the best value. Most of the parameters had a large range with fairly similar performance. Values in the middle ranges of what I expected would work had similar performance. The biggest improvement I found from adjusting parameters from my initial guesses was increasing the tournament size from two to six.

4b. Future work

Other experiments that would be interesting to carry out are applying a chance of mutation to all candidates of a future generation as opposed to those only generated from crossover. I suspect that the optimal mutation rate would lowered by doing this but don't have any guesses as to whether the performance would be increased.

Additional choosing alternate selection algorithms, such as a proportionate selection based on the Boltzmann distribution or a rank based selection would be interesting. I suspect that a proportionate selection would result in fewer generations but a significantly longer time for a solution. Rank I suspect would perform similarly to the tournament selection when a tournament size of six or more is used, but would take more time because the candidates would all need to be sorted by their score.

4c. Other considerations

Overall the genetic algorithms performed well compared to randomly selecting candidates. Consider that for a puzzle of length 10 and a character set of size four there are 10^4 permutations. A solution would be found in an average of 5000 unique guesses. For the genetic algorithm I described using a generation size of 50 with 25 new candidates generated each generation it took ~425 unique guesses to find the solution with 17 generations, which is the best average over 1000 tests that I recorded (Using a tournament selection size of 6).

5. REFERENCES

[1] Russell, S., Norvig, P. "Artificial Intelligence – A Modern Approach (3rd ed.)", pp 126-128 , Pearson Education Inc, 2010.