CS557 - Project Three

Andrew Struthers, Austin Laverdure, Bridget Smith

May 2023

Magic Square Problem Description

A magic square of order n is an arrangement of the numbers from 1 to n^2 in an n-by-n matrix, with each number occurring exactly once, so that each row, each column, and each main diagonal has the same sum.

Construct a magic square of order n (where $n \leq 10$ is an input parameter) using a genetic algorithm. First, generate an initial population of magic squares with random values. The fitness of each individual square is calculated based on the "flatness", that is, the degree of deviation in the sums of the rows, columns, and diagonals. The program should find the solution in most of the cases. Write your own code.

Magic Square Python Solution

Genetic Operation Implementation

The genetic operation implementation written using Python began by generating n number of squares ($n \leq 10$) containing values 1 to n^2 in an n x n matrix. In this case, $n = 5$ was selected. Then, generations were iterated through where fitness scores of individuals were calculated based on their flatness. Flatness measured by a square's closeness to a magic square through finding the sums of each column, row, and long diagonal, and finding the difference between the magic sum (target) and each of those sums. The absolute value of these differences were summed up to calculate a fitness score. The population was then reorganized to have the fittest members of the population near the beginning of the array of the population matrix. Next, the top performers were selected for subsequent crossover. The new population was generated by generating new individuals through splicing, or crossing over, the parental square information from two randomly selected top performing individuals to produce new offspring squares. Mutation rates were then applied and values were replaced randomly in accordance with the mutation rate. The top performing individual after 1000 generations, starting from a population of 100000, was returned.

Results

Unfortunately, my computer did not complete a run through of the code using those parameters. Using an initial population of 100 individuals and 1000 generations produced a square that had duplicated values, so the code is in need of some additional work.

Magic Square C++ Solution

Genetic Operation Implementation

The genetic algorithm implemented in C++ starts by generating a number of random squares equal to the population size. The "flatness" of these squares is determined by first determining the

optimal sum, or the "magic sum" with the following equation:

$$\text{target} = n \cdot \frac{(n^2 + 1)}{2}$$

which gives us the target value for the sum of each of the rows, columns, and main diagonals. Then we find the difference between the target and the sum of each of the rows, columns, and main diagonals. Adding up the differences between the target sum gives us the overall fitness of the square, which we want to minimize to 0. A flatness value of 0 means that there is no difference between the target and the true sum, meaning we have indeed found our true magic square. The evolution part of the algorithm does tournament selection using two randomly selected parents, then performs crossover. Because the uniqueness of the genes needs to be preserved during crossover (aka we don't want a magic square with two 1's, for example), we implemented the Cycle Crossover algorithm. The Cycle Crossover algorithm was selected because it had the smallest runtime efficiency for a guaranteed unique crossover algorithm that we could find. It runs in O(n) time, which is good because we will be performing crossover a lot. After crossover occurs we then attempt to mutate the child. A mutation will generate a brand new random square. This process will repeat for the specified number of generations, or if the algorithm finds a perfect magic square beforehand, it will break early.

Results

The results of this problem were not very promising. The genetic algorithm can reliably find a 3×3 magic square in very few iterations, but when tested on anything larger than a 3×3, perfect magic squares were not reliably found. There was one instance where a 5×5 square was found, but I believe that was purely due to lucky random generation just generating the proper square. I have not been able to recreate the random generation succeeding. Below we can see some of the outputs with various n values.

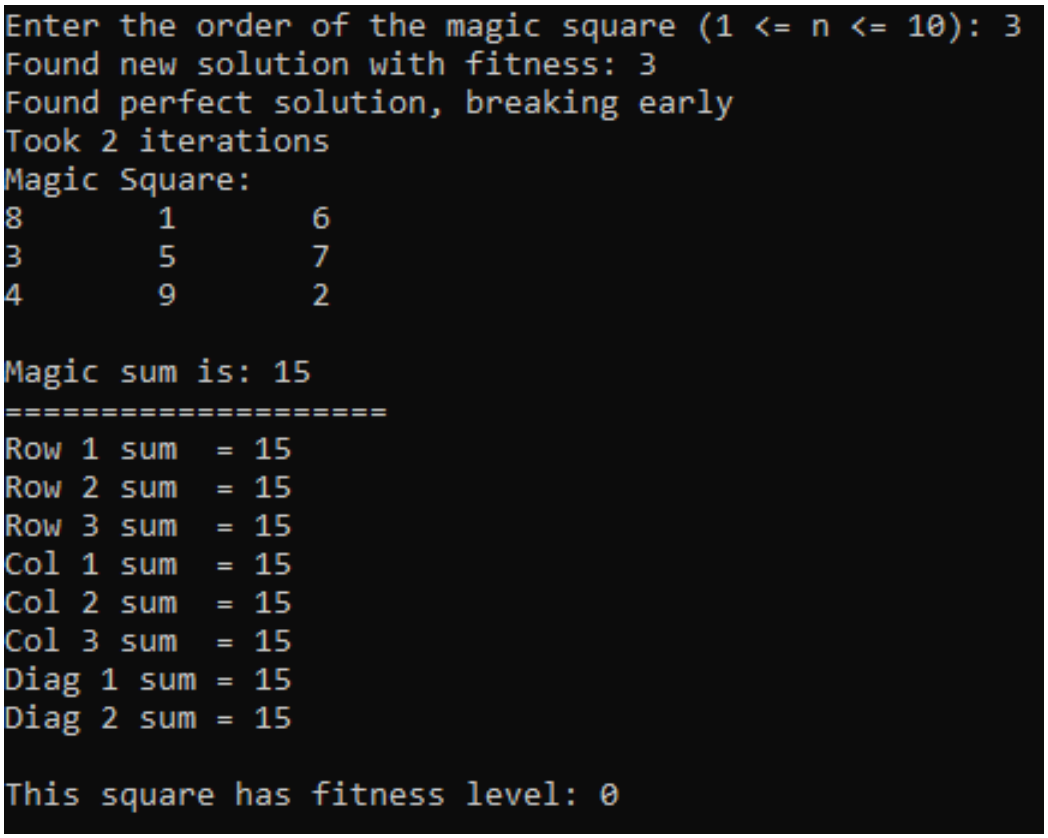

Figure 1: 3×3 magic square generating in two iterations, with population size of 10,000

```
Enter the order of the magic square (1 <= n <= 10): 4
Found new solution with fitness: 20
Found new solution with fitness: 15
Found new solution with fitness: 14
Found new solution with fitness: 13
Found new solution with fitness: 12
Found new solution with fitness: 9
Found new solution with fitness: 4
On iteration 100
On iteration 200
On iteration 300
On iteration 400
On iteration 500
On iteration 600
On iteration 700
On iteration 800
On iteration 900
On iteration 1000
Took 1000 iterations
Magic Square:
16      3       13      2
1       5       12      15
8       11      6       10
9       14      4       7

Magic sum is: 34
====================
Row 1 sum  = 34
Row 2 sum  = 33
Row 3 sum  = 35
Row 4 sum  = 34
Col 1 sum  = 34
Col 2 sum  = 33
Col 3 sum  = 35
Col 4 sum  = 34
Diag 1 sum = 34
Diag 2 sum = 34

This square has fitness level: 4
```

Figure 2: 4×4 magic square failing to converge after 1,000 iterations

```
Enter the order of the magic square (1 <= n <= 10): 5
Found new solution with fitness: 49
Found new solution with fitness: 46
Found new solution with fitness: 42
Found new solution with fitness: 39
Found new solution with fitness: 33
Found new solution with fitness: 32
Found new solution with fitness: 25
On iteration 100
On iteration 200
Found new solution with fitness: 22
On iteration 300
On iteration 400
On iteration 500
On iteration 600
On iteration 700
On iteration 800
On iteration 900
On iteration 1000
Took 1000 iterations
Magic Square:
2       12      13      16      17
6       11      23      21      4
22      15      9       8       10
20      5       19      18      7
14      25      1       3       24

Magic sum is: 65
====================
Row 1 sum  = 60
Row 2 sum  = 65
Row 3 sum  = 64
Row 4 sum  = 69
Row 5 sum  = 67
Col 1 sum  = 64
Col 2 sum  = 68
Col 3 sum  = 65
Col 4 sum  = 66
Col 5 sum  = 62
Diag 1 sum = 64
Diag 2 sum = 66

This square has fitness level: 22
```

Figure 3: 5×5 magic square failing to converge after 1,000 iterations

```
Enter the order of the magic square (1 <= n <= 10): 6
Found new solution with fitness: 89
Found new solution with fitness: 77
Found new solution with fitness: 75
Found new solution with fitness: 73
Found new solution with fitness: 57
On iteration 100
On iteration 200
On iteration 300
On iteration 400
Found new solution with fitness: 49
On iteration 500
On iteration 600
On iteration 700
On iteration 800
On iteration 900
Found new solution with fitness: 48
On iteration 1000
Took 1000 iterations
Magic Square:
22      31      25      8       18      7
24      4       23      16      13      32
1       6       36      29      33      5
19      15      3       30      17      21
11      35      12      26      9       28
34      20      10      2       27      14

Magic sum is: 111
====================
Row 1 sum  = 111
Row 2 sum  = 112
Row 3 sum  = 110
Row 4 sum  = 105
Row 5 sum  = 121
Row 6 sum  = 107
Col 1 sum  = 111
Col 2 sum  = 111
Col 3 sum  = 109
Col 4 sum  = 111
Col 5 sum  = 117
Col 6 sum  = 107
Diag 1 sum = 115
Diag 2 sum = 121

This square has fitness level: 48
```

Figure 4: 6×6 magic square possibly trying to converge over 1,000 iterations

Conclusion

In conclusion, we feel as though the genetic algorithm implementation wasn't done effectively. Playing with these parameters and increasing the population size by orders of magnitude didn't help the algorithm converge, nor did playing with the mutation probability. The only success the algorithm had after the 3×3 case was when the random generation algorithm randomly found a magic square during mutation. This is very unlikely to occur, and is not the intention of the algorithm anyways, so that result should be discarded. In reflection over where the algorithm could

have gone wrong and how it could be improved, we could have implemented some form of elitism where, during the selection phase, the possible parent candidates could only be selected from the top 50% of squares in that generation. We tried multiple different unique crossover algorithms, like Order Crossover or Partially Mapped Crossover, but the algorithm implementation didn't seem to make a huge difference. We also tried implementing the genetic algorithm with no crossover, where we just do tournament selection to select a single "parent" that we then try to mutate for the next generation, as opposed to using crossover to make a child that then gets potentially mutated. This didn't work any better than any of the crossover implementations in cases where $n \geq 4$. We also tried introducing more mutation, where instead of grabbing two elements of the current square and swapping them, we could set a variable for the number of swaps that each child undergoes. We had hoped that, by adding more mutation into the algorithm, we could force the generations to jump out of the local minima that they seemed to be falling into. This didn't help too much, as we were still getting the same behavior of almost converging but never quite actually reaching the destination.

Maximize Function Problem Description

Maximize the following function:

$$f(x,y) = \sin\left(10\pi x + \frac{10}{(1+y^2)}\right) + \ln\left(x^2+y^2\right)$$

using the following operations: selection, crossover, and mutation.

Your code should:

- find the maximum value of $f(x,y)$ and the values of x and y for which this maximum is obtained
- the evolution of the average $f(x,y)$ value for all generations (as a plot)
- the values of the following parameters: number of generations, population size, mutation probability
- your conclusions regarding how the above parameters influence the convergence process

Maximize Function Python Solution

Implementation

We used Python to implement the maximum of the function. The genetic algorithm used fulfilled the requirements of utilizing selection, crossover, and mutation methods. The parents are selected randomly. The mutations are applied over the children randomly.

Results

The results generated from our genetic algorithm did a pretty good job of maximizing the function. If the code is ran a few times, sufficient results will be yielded. We found that raising the population size is more likely to give an increase in size than raising the number of generations if the number of generations is already significant.

Conclusion

In conclusion, genetic algorithms do a great job in maximizing a function if the peroper requirements are fulfilled. Using code that made the best parents have offspring with each other made our genetic algorithm generate poorer results. In previous versions of the program, mutations would be generated on every child in the population. This would lead to unpredictable results. Sometimes the results would be good but other times they would be a lot lower than the max from other runs. By applying mutations to only a certain percentage of randomly selected offspring, we were able to come up with more consistent results.

Maximize Function C++ Solution

Implementation

We used C++ to implement the maximization of the provided function. The genetic algorithm used implemented selection, crossover, and mutation. During the evolution process, parents were randomly selected from the population by choosing two random entries in the population and returning the entry with the highest fitness. Each group of two parents underwent crossover to form a child, which consisted of taking the average x and y values of the two parents as the child's

new (x, y) pair. The child then undergoes a mutation process. If the child has a mutation, the x and y values of the child are completely randomized. This represents a full mutation, effectively randomizing the children with mutations. The new population is then formed by each of the children. This process will continue for the specified number of generations. Each generation we also store the average fitness of the population for graphing at the end.

Results

We ran the code with many different parameter values, but for the purposes of this report, we used $1,000$ generations, a population size of $100,000$, and a mutation probability of 0.05. After the specified generations, the maximum value of $f(x, y)$ was 6.07454, found at $(9.84436, 7.93871)$.

```
Generation 1000:
Best Individual: (9.84436, 7.93871)
Best Fitness: 6.07454
Average Fitness: 5.59061
```

Figure 5: Maximum fitness of 100,000 population over 1,000 generations

Graphing the average fitness of each generation in Excel results in the following graph:

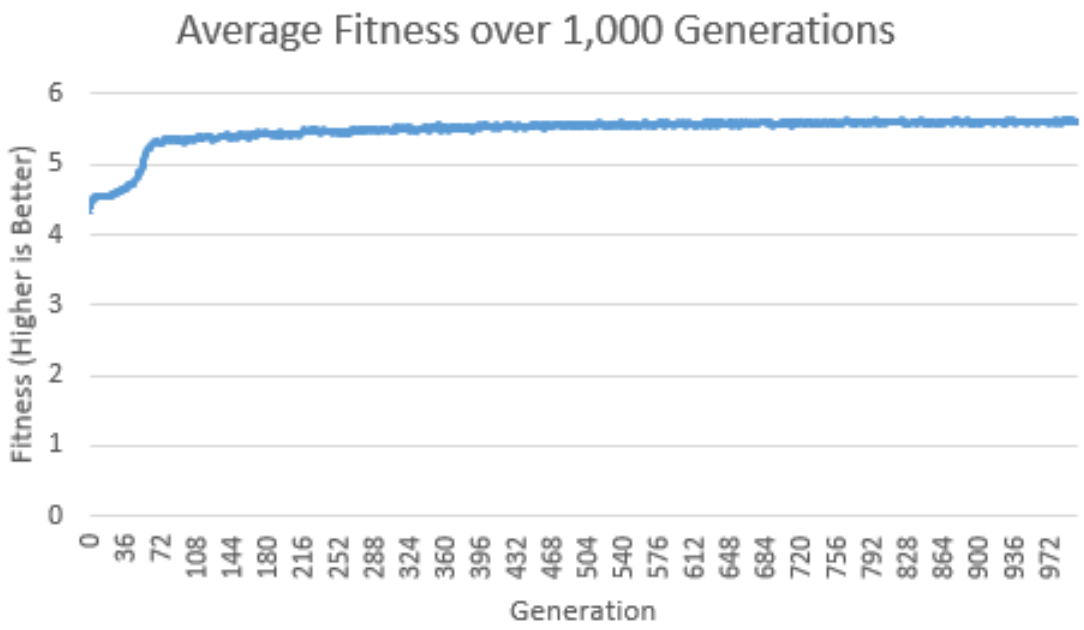

Figure 6: Average fitness of 100,000 population over 1,000 generations

We can see that, initially, the average fitness is at it's lowest (≈ 4.36). This result makes sense, because the first generation is just randomly generated without any optimization. This result seems to indicate that the average value of the function on the given range is actually ≈ 4.36. We can see that in the first 100 generations, the fitness grows exponentially, and for the remaining 900 generations the average slowly crawls to just over 5.5. The slow increase of the average fitness is a clear demonstration that the selection and crossover are creating more fit children over time, and the mutation and lack of elitism in the algorithm is preventing the trend of increasing fitness to plateau or decrease. We can see small jitters in the average fitness, which is due to mutated children being entirely randomized.

Conclusion

We experimented with many different parameters before settling on this final run. We noticed that when the population was relatively small ($\approx 1,000$) and the mutation rate being 0.5, the average fitness would be very noisy and never increase above ≈ 4.5. A mutation rate of 0.5 means that half of all children are completely randomized every generation, which would certainly explain the average fitness never improving. That approach still gave us roughly the same maximum value, but we believe that was only the case because after $1,000$ generations, the likelihood of randomly generating the correct (x, y) pair was relatively large. Having such a high level of stochasticity is not particularly better than purely random generation or exhaustive search of the domain. Likewise, if we had a low mutation rate but a small population, we saw that even after $1,000$ generations, we didn't get nearly close enough to the true maximum value of the function. With a low mutation rate and small population, we needed orders of magnitude more generations to get the true maximum. Through this experimentation with the parameters, we determined that a large population size coupled with a low mutation rate was sufficient enough to have a small number of generations while still getting an acceptable answer.