**CSE 848 HW4**

**Tim Taviano**

I Chose to implement Task 1 a genetic algorithm for a variant of the SUDOKU puzzle(nxn) in Python 3

1. My representation of a puzzle was broken down into two parts the first part was the mini blocks inside of the puzzle which were sqrt(n) by sqrt(n) size. I chose to represent these blocks as a one-dimensional array. This Block objects were randomly generated with one copy each of the numbers 1 to N so that each block would not have any duplicates. Thus, only the rows and columns were the only places with duplicates. Then to create a puzzle n blocks were put together to form a puzzle with block 0 being in the top left and block n-1 being in the bottom right corner. Now that the blocks are put together in one object, the rows and columns of the puzzle can be obtained by combining the rows and columns from the blocks. I created functions to retrieve the columns and rows because these will be pivotal for the fitness function. My fitness function calculated the number of duplicates in each row and column. To do this I retrieved every row and column as separate lists and turned them into sets to see how many unique numbers were in the row. If the list and set did not have the same number of elements this meant that there were duplicates, and the number of duplicates can be collected by subtracting the length of set from the length of the list. This means the fitness function of a puzzle is as follows:

With this as my fitness function that means that the best possible fitness for a sudoku puzzle in this representation would be 0. This would mean that there are no duplicates in any row or column, a solved puzzle. For larger puzzles this means that the fitness values can be much higher because there is much more room for duplicates. Thus, the GA is trying to find the global minimum. Now to solve the problem from random configurations, mutation and crossover operators had to be chosen for the genetic algorithm. For this implementation multiple mutation operators were initially tested and the one which performed better will be shown in plots in part B. The first mutation operator which I tried was to just shuffle the one of the internal blocks of a puzzle so that it would just have a different permutation of the numbers inside the block. This mutation operator while it does excel at exploration it severely hindered the algorithms ability to exploit the current population. The changes to the population were too drastic to find a good solution. The second crossover operator which performed better. This operator randomly chose one block in a puzzle and then just swapped the positions of two numbers within the block. This allowed for more subtle changes and better exploitation. For crossover operators, I implemented a combination of three operators which all had a probability of being chosen. The first crossover operator was just creating copies of the two parents selected. This first operator was given a probability of zero in the final runs of the GA because use of this operator led to quick convergence of the population into local minimum. The second operator chose a random block from parent one and a random block from parent two and placed the block from parent 2 into the slot chosen for parent one. Since two children were created from each set of parents the same process could also happen in reverse where a block from parent one is replacing a chosen block in parent 2. The third operator is similar to the mutation operator, where a random block is chosen in one of the parents and then two of the numbers in the chosen block are swapped. This third operator helps with exploitation and exploration while the second operator is much more used for exploration because there is no guarantee that block 0 from puzzle A will be a benefit in a different puzzle. Thus, to understand the second and third operators, I ran the GA with different probabilities for the crossover operators and the mutation operators. These runs will be shown in part B. The overall implementation of the GA is as follows: A population is randomly generated with size 50. The population is evaluated for their fitnesses and sorted. The best fitness and average fitness are recorded. The algorithm checks if the best fitness is 0 (a.k.a. the puzzle was solved), if it has then it stops otherwise it continues. The pool of parents is then chosen using tournament selection with duplicates. Then 2 children are created (using the crossover operators) for each set of parents chosen until the children pool size is twice the size of the population. Then (µ+λ) selection is used to choose the survivors. The population is cut back down to the original population size. The population then has a chance to mutate using the mutation operator discussed earlier. Then the population is evaluated and sorted, and the parent selection process begins again. This continues until a solution is found or the max number of generations was reached. I also ran into a problem where the GA would converge to local minimum so in some of my runs I implemented a convergence reset where, if the best fitness had been the same in a row for a tenth of the max allowable generations. I would kill the population and generate a completely new population. Runs with and without the convergence reset will be shown in part B.

1. I have implemented my GA using Python 3 and plotted using the python library matplotlib. The code can be found on github at <https://github.com/tavianot/CSE848HW4.git> .The first two plots show a comparison between 9x9 and 16x16 runs of the GA with the same number of generation and same probabilities of crossover and mutation.

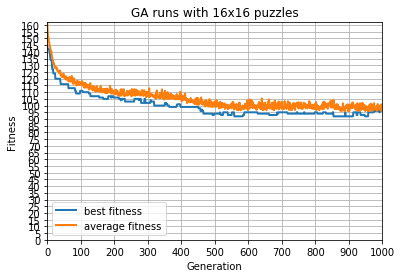
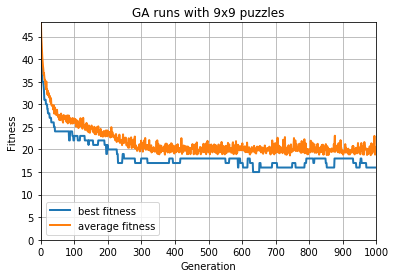
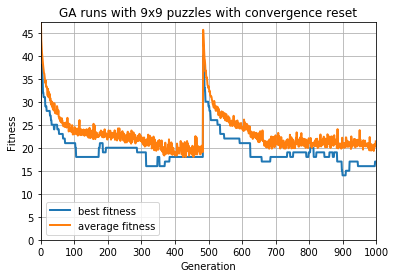


Figure 1 16x16 5% mutation 80% second crossover operation(perturbation) 20% third crossover operation(block swap)



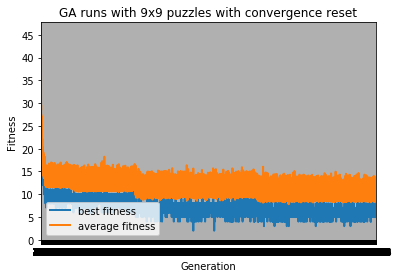
**Figure 2 9x9 5% mutation 80% second crossover operation (perturbation) 20% third crossover operation(block swap)**

The 16x16 puzzles have much higher fitnesses because there are a lot more places for there to be potential duplicates. Both runs seem to have converged to local minimum near the end of the run. To see if this was true, I ran the 9x9 with the convergence reset discussed in part A. The 16x16 puzzles took much longer to run under the same parameters.



**Figure 3 9x9 5% mutation 80% second crossover operation(perturbation) 20% third crossover operation (block swap) with convergence reset**

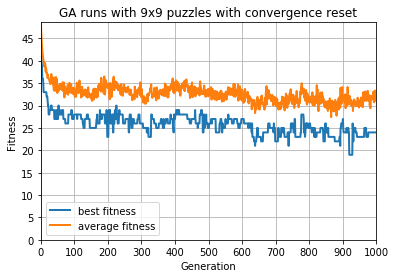
I did not continue to run the 16x16 because the running times on my computer took significantly longer since the running time for the 16x16 puzzles. I believe this is due to the calculation of the fitness function being reliant on calculating the duplicates in each row and column. The Convergence reset seemed to help a little with the minimum fitness found being 14 compared to the 15 found without the reset. This still was not a satisfying result, so I ran the same algorithm with much higher max generation (100,000). This took about 12 hours to run. Also, matplotlib was not able to handle the granularity of the 100,000 generations so the x axis is unreadable.



**Figure 4 9x9 5% mutation 80% second crossover operation(perturbation) 20% third crossover operation (block swap) with convergence reset and 100,000 generations**

This result was much more satisfying seeing that the GA could get to 3 duplicates in the whole puzzle several times. Not finding an exact solution is a little disheartening but makes me think that my mutation and crossover operators need to be slightly more complex to be able to find the global minimum much quicker.

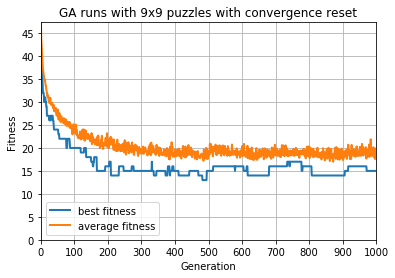
The results so far were hopeful, so I thought maybe if there was more mutation and diversity then the solution would be found so I tried a quick test of raising the chance of mutation in the population to 20% rather than the 5% which has been shown so far.



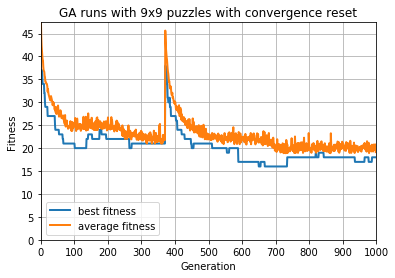
**Figure 5 9x9 20% mutation 80% second crossover operation(perturbation) 20% third crossover operation (block swap) with convergence reset**

This raise in mutation rate did not allow the fittest individual to reach even close to same fitness as previous runs of the GA, so I reverted it back to the 5% mutation rate of the population.

I also tried find the effects of the different crossover operators on the population. So I raised the second operator while lower the third in some runs and vice versa. Some the results are shown below.



**Figure 6 9x9 20% mutation 90% second crossover operation(perturbation) 10% third crossover operation (block swap) with convergence reset**



**Figure 7 9x9 20% mutation 20% second crossover operation(perturbation) 80% third crossover operation (block swap) with convergence reset**

It seems that having a higher second crossover operator (perturbation) was more adept at finding better local minimum than having the third operator (block swap).

With these results, I believe that I would need more complex perturbation to achieve finding the global maximum. Potentially an operator such as two or three blocks swapping some of their numbers rather than just swapping within one block at a time.

1. To solve real sudoku puzzles using my GA, I would need to create some type of correction method that would set the numbers given into their appropriate positions in each block every time. This could include checks to make sure that these numbers were never swapped and shuffled, or maybe even easier would be a repair operators to go in and swap the numbers back into their respective positions within their blocks for each puzzle in the population. With the repair operator running before the fitnesses are calculated, this would ensure that the GA would be directed in the direction of solving the puzzle which was given because the numbers given would be hard coded into each individual in the population. For this to be implemented my GA a list of tuples with (value,block index) could be passed to a function which would find the value in the certain block and swap the number in index with the current location of value in the indicated block and would do so for each tuple in the list. This would allow a specific puzzle to be solved assuming that the GA does not get stuck on a local minimum.