Josh Audibert
Nick Cebry
Breanne Happell
Morgan Hopeman

Project 2 Genetic Algorithms Write-Up

Question 1

Child Generation: Child generation was handled the same for all three problems. The function would take in two parents and choose a random number that was not the end indices. That would become the splitting point for both parents. The first half of parent one was then added to the second half of parent two to make the resulting child.
For Example Parent 1 = [1, 0 , | 1, 0 , 0] Parent 2 = [0, 1, | 1, 0 ,1] Crossover Point = 2 (represented by |) the resulting child would be [1, 0, 1, 0 ,1] .

Evaluating small population selection example: While the input to the selection function changes for every problem the process is always the same. This is an example using numbers from problem 1 but could easily looked at as if it was from 2 or three. If we have an example of a goal of 15 and a population of 5, 6, 7, 3, and 8 we will first generate 10 parents (for this I am only going to generate 3). If the first three parents are [5, 6, 7] = 18, [6, 7, 3] = 16 and [5, 6] = 11 then these all go through the fitness function to return -3, - 1, and 1 these are then all linearly scaled and manipulated so that if represented on a number line 1 would have the largest space, then -1, then -3. We then generate a random number on the number line and the area where it fall determines the parent. For example the first random number could end up in 1 and the second could end up in -3 thus those become the next set of parents.

Problem 1:

Fitness Function: The fitness function for problem one was modified from the scoring function. We calculated fitness by the difference between the sum total and goal number. The best fitness was the smallest difference with a fitness of 0 being the ideal answer. A negative fitness isn't better than 0, and we take that into account during parent selection.

Selection: Selection linearly scales the fitness so that numbers that went over the goal had a lower fitness than numbers that were just under the goal. Then the selection algorithm chooses two parents with weighted probabilities based on these scaled fitnesses (giving more weight to fitnesses close to 0).

Crossover: Crossover, as described in "Child Generation" was splitting the parents at a random index with the exception of the end indices which were removed from the possible crossover number.

Mutation: Mutation for problem one chose a random index and switched it on or off. (To better understand read question 2 about representation.

Population size: We used a population size of 10 for problem 1 because more variety increased the chance of finding the solution.

Problem 2:

Fitness Function: The fitness function for problem 2 utilized the scoring system for the questions and took the product of bin one and the sum of bin two and added them together (we decided not to do the extra computation of dividing this by 2 because fitnesses are relative anyways). The best fitness was the largest total.

Selection: Selection linearly scaled the fitness so to account for combinations that ended up with negative scores. Then the selection algorithm chooses two parents with weighted probabilities based on these scaled fitnesses (giving more weight to greater fitnesses).

Crossover: Crossover, as described in "Child Generation" was splitting the parents at a random index with the exception of the end indices which were removed from the possible crossover number.

Mutation: Mutation chooses two indices with different bin numbers and swaps them. This keeps the same numbers in each bin while changing the contents slightly.

Population size: In problem 2 we used a population of 8. This is because 2 needed to evaluate things through the fitness function to weed out very negative numbers. However, if we decreased the population size too much it would close around local maximums where negative numbers were in box 3 because it didn't generate enough with even numbers of negatives.

Problem 3:

Fitness Function: The fitness function for problem three is the most complex in the number of things it has to consider. The fitness function can be written as $(10 + h^2 - c) * .75^r$ (where h is the height of the tower, c is the total cost of the pieces used, and r is the number of rules broken) which is just the scoring function multiplied by .75 for every time the tower breaks a rule. This way towers with fewer broken rules is rewarded slowly teaching the algorithm to choose acceptable towers.

Selection: Selection linearly scales the fitness to account for potentially negative scores from the cost of the tower. Then the selection algorithm chooses two parents with weighted probabilities based on these scaled fitnesses (giving more weight to greater fitnesses).

Crossover: Crossover, as described in "Child Generation" was splitting the parents at a random index with the exception of the end indices which were removed from the possible crossover number.

Mutation: Mutation chooses a random input piece and removes it from the tower if it was already in it, and adds it in a random location if it wasn't already in the tower. We then have to check whether the location of the new piece in the tower overlaps with an existing one, in which case one is removed at random.

Population size: We used a slightly smaller population size of 5 for problem 3 because it was more important that the population went through the fitness function quickly so it could find legal towers faster. When we had a larger population we would get more variety but less legal towers.

<u>Question 2</u>

Problem 1:

To handle illegal crossovers in number 1 we encoded our genome so that each index in the list corresponded to an input number. This number was then either turned on (represented by a 1) or turned off (represented by a 0). The numbers that were turned on were then added to create the sum. This representation meant that we never had to worry about illegal crossovers because the way

that we generated children never changed the number of indices in the list and therefore never created a potential duplicate problem.

Problem 2:

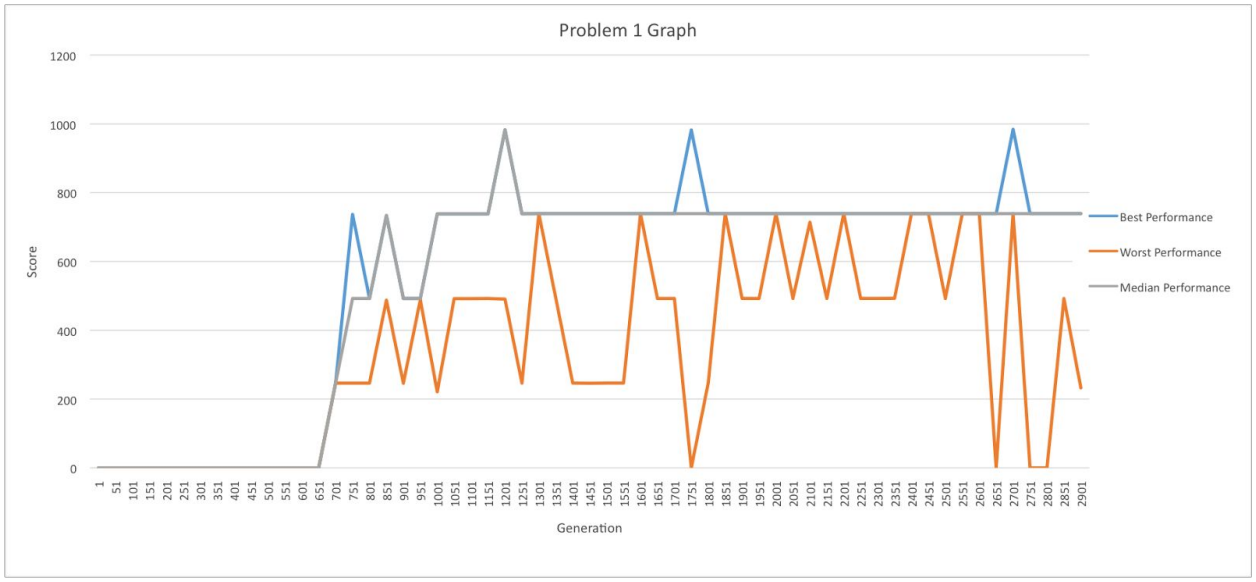
In problem 2, like in problem 1, the index in the list represented the numbers in the original input. Instead of turning them on or off like in problem one, we gave each a random value of 1, 2, or 3 to represent the bin each was in. We would then perform a check (binCheck) to make sure that each bin was equally represented. To deal with illegal crossovers after we create children we run binCheck to make sure that any illegal children are corrected.

Problem 3:

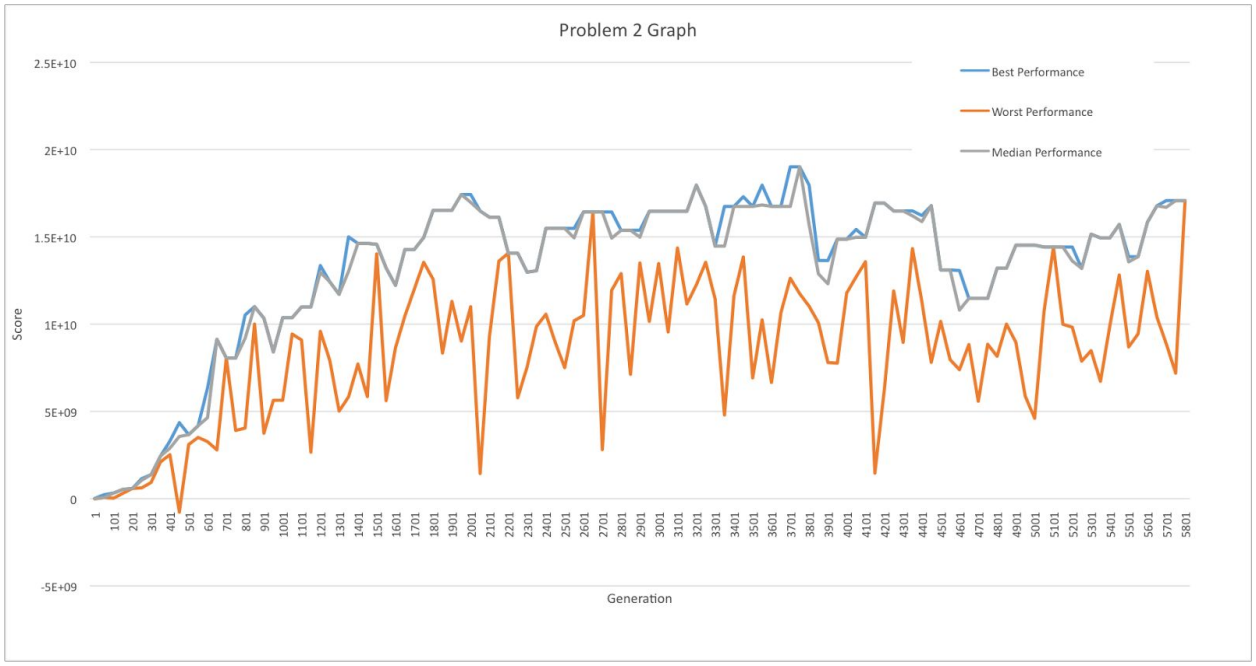
In problem 3, as in the other two problems, the index in the list represents a potential tower piece. The numbers in the list tell you what place in the tower they hold (1 is the bottom). 0s represent pieces that are not included in the tower. By structuring the solution this way we did not have to worry about duplicating pieces only duplicating positions in the tower. To account for this we developed checkTower to make sure there are no pieces in the same spot in the tower. We run check tower on the initial population and on each child created to make sure that they are legal.

Question 3

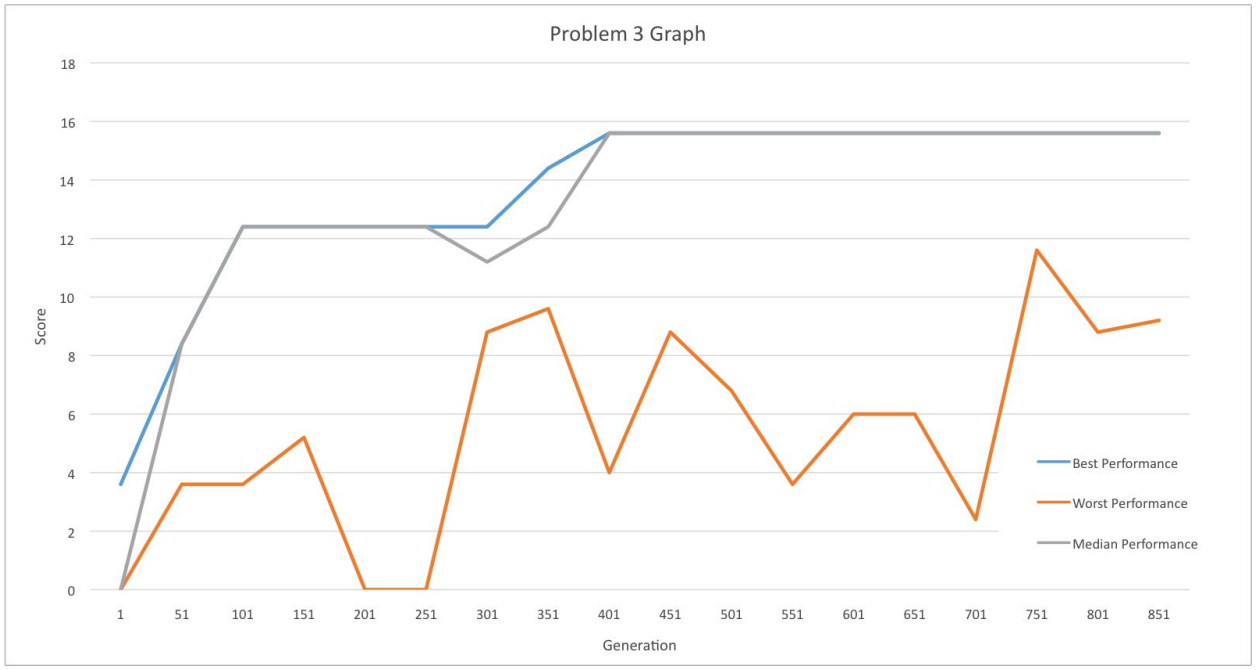
Problem 1 Fitness over generations (average of 5 runs) with no culling and no elitism:



Problem 2 Fitness over generations (average of 5 runs) with no culling and no elitism:



Problem 3 Fitness over generations (average of 5 runs) with no culling and no elitism:



Question 4

For problems 1, 2, and 3 culling was implemented using the cull function in the genetic algorithm class. The cull function takes the population list and the number of individuals to cull as

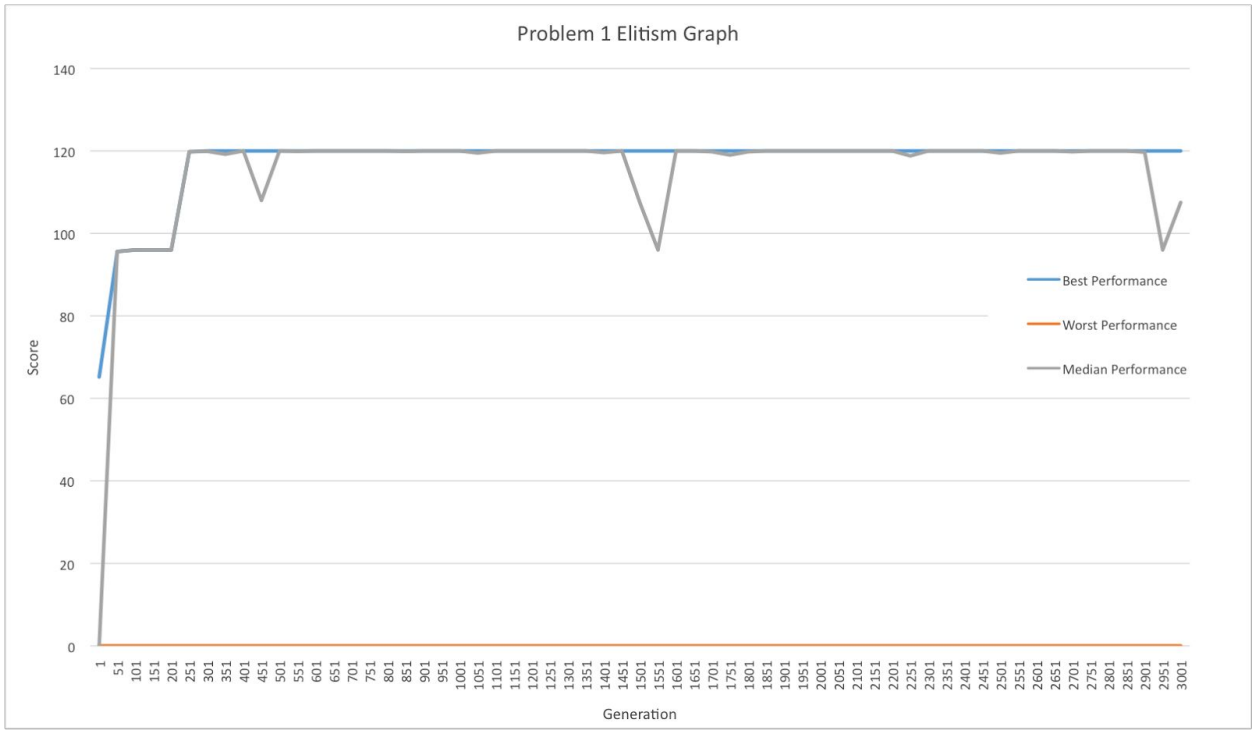
input. It then sorts all of the individuals in the population by the value of their fitness function, and the given number of individuals with the lowest fitness function values are removed from the list.

For all three problems elitism was implemented through the getElites function in the genetic algorithm class. The get elites function takes the list containing the population from which individuals are to be selected to continue to the next generation, and the number of individuals to select. The population list is sorted according to the score the individual would receive if selected as the final solution. A new list is created with the given number of individuals with the highest score and returned. The runGA function is responsible for adding these individuals to the next generation of the population.

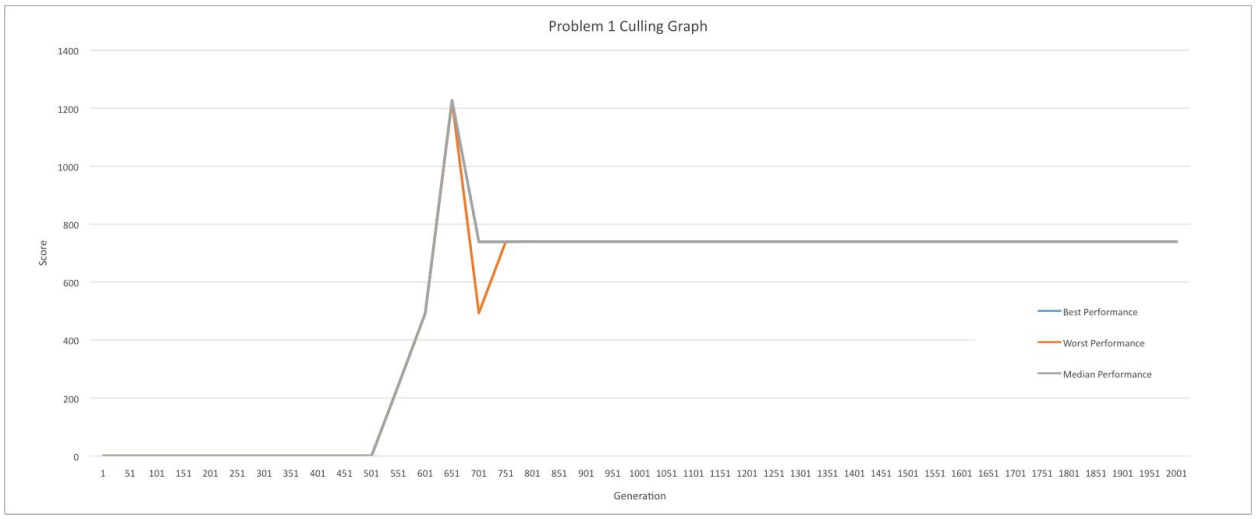
As you can see from the graphs below overall culling and elitism had a positive overall effect but there are some exceptions. When we initially implemented elitism we would have problems when the best score was 0 initially. Because elitism works by saving old score this meant it would save things with scores of zero even when that was obviously not ideal. The other initially unexpected outcome was that elitism negatively affected the worst performance on all three function. This is because of a couple of reasons. The first is that elitism takes effect based on the fitness function and not on the score this means that elitism may often save functions with a 0 or negative score because they have a higher fitness function (for problems 1 and 3). Second, while elitism may retain the best scores it does not affect how the next generation changes. For instance in problem 2 it may save a parent with 2 negatives and when this parent breeds with another it could create a very negative score by having an odd number of negatives. These effects caused a lower worst case than the normal function and thus could also cause the median to drift lower than in the more generic GAs.

Culling always had very positive effects on our graphs. All three categories increased. This is most likely because unlike elitism culling only worked on the current population not retaining anything from previous generations. The only time that culling was ineffective was in number one after the correct answer had been found this is similar to number 1's original graph and may have to do with there only being 1 correct answer and thus the combination needed to produce that child was less ideal than other combinations that got close but not to the goal. You can also see thought that even in the case of problem 1 culling caused it to converge on the goal faster than in the original GA. Because culling tended to drastically improve the performance of worst cast and best case the median case also drastically improved.

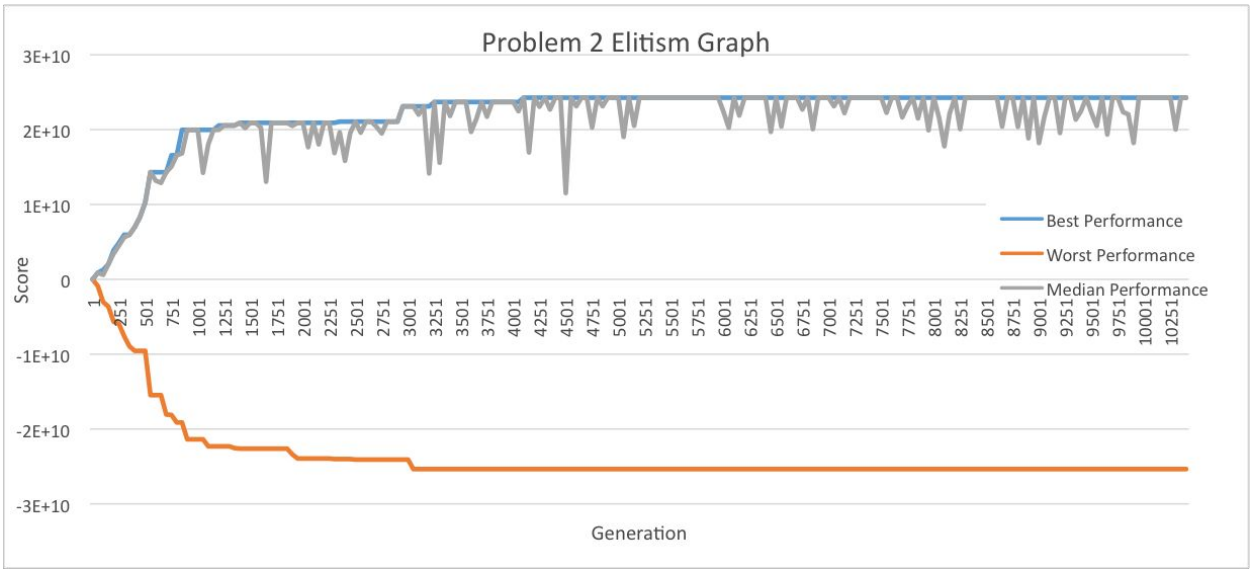
Problem 1 Fitness over generations (average of 5 runs) with elitism and no culling:



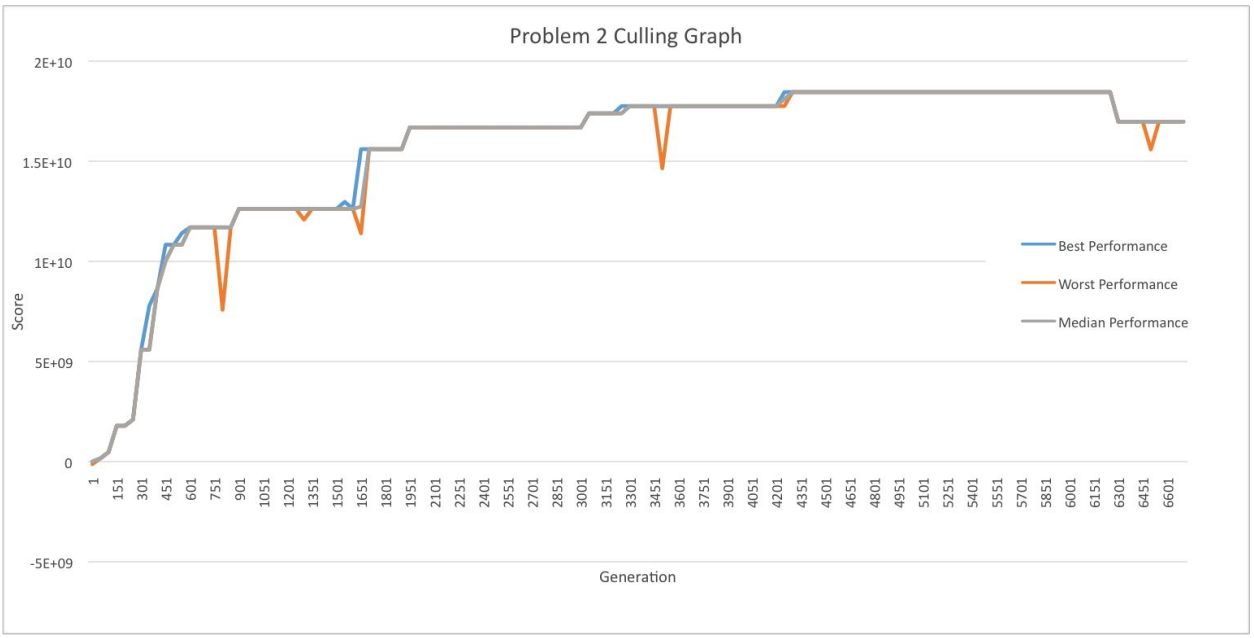
Problem 1 Fitness over generations (average of 5 runs) with culling and no elitism:



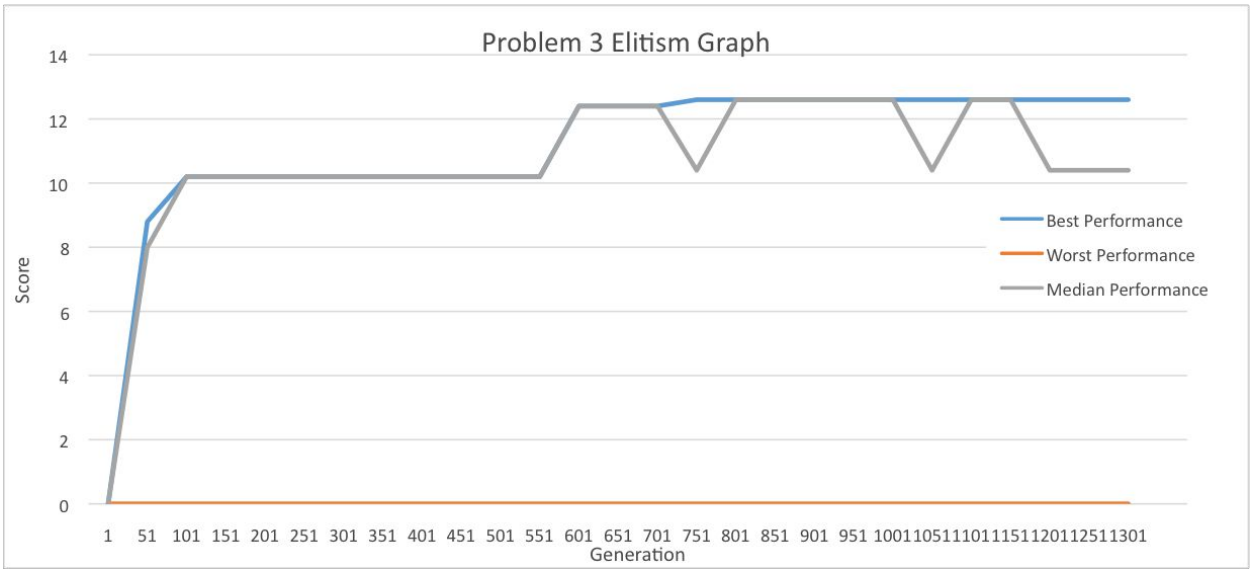
Problem 2 Fitness over generations (average of 5 runs) with elitism and no culling:



Problem 2 Fitness over generations (average of 5 runs) with culling and no elitism:



Problem 3 Fitness over generations (average of 5 runs) with elitism and no culling:



Problem 3 Fitness over generations (average of 5 runs) with culling and no elitism: