CS 4341 Introduction to Artificial Intelligence
A2 Writeup
Nicholas Bradford, Thomas Grimshaw, Lucas Lebrao, Tim Petri
9/25/2015

Your approach to computing a fitness function, selection, child generation, crossover, and mutation. Also, what population size did you use? Provide an example of evaluating a small population, selection, and generating a child for one set of parents. How you handled the problem of illegal children for crossovers. Provide an example, or explain how your representation prevented the problem from occurring.

Fitness:

Puzzle #1: The fitness function takes the sum of the numbers in the list and returns the distance between the sum and the target value. This was selected because sets with a sum close to the target value, even if they are over the target, are more likely to create children that score better.

Puzzle #2: The fitness function for this puzzle is the same as the score of the individual. This was selected over any other approaches due the nature of scoring not having any penalties or invalid combinations. Therefore, score is the best way to determine fitness.

Puzzle #3: Because the Tower structure had such strict rules for validity, it was necessary to provide fitness points for partial towers. This was done from both the bottom and top of the tower working inwards, increasing fitness for every piece that could later be used as the foundation or top of a valid tower. This approach worked well in conjunction with the crossover method, which combines the bottom of one tower with the top of another.

Selection: Selection is performed by taking the population and selecting two random parents without replacement. The two parents generate a child only if the levenshtein distance between them is greater than a threshold. This method prevents incest in the population. The threshold is set to a quarter of the length of the parents to begin with and is decreased when the function is unable to generate any children with the given threshold. This method is a modification of the selection from the CHC algorithm developed by Eshelman.

Crossover:

Puzzle #1: Crossovers are done by taking the two parents and picking random cut points on each of them. The first part of parent 1 and the second part of parent 2 are

then put together to form the child. When the merge happens any values that are used more than the available amount in the initial input are removed. This leads to children generally being shorter than their parents, but extra length is added back in mutation.

Puzzle #2: Crossovers generate a random cutoff value within the array range max and then cuts the first parent's array. This array contains n values out of total. Now using the cut array, it runs through a loop that individually removes every existing value in parent1 from parent2, which leads parent2 to hold max-n values. At the end, parent2 extends parent1, resulting in a total (n) + (max-n) = max values, which results in a valid child.

Puzzle #3: Crossovers involved taking a random number of pieces from the bottom of one parent and the top of the other, and adding them to a child incrementally. During this crossover, the addition of each piece to the child is checked against a static hash table recording which pieces are allowed to be used. If the piece attempting to be added to the child has already been added, then a random unused piece is "mutated" into the child, making sure that the child ends up with the same number of pieces. If this random piece addition was not implemented, the average height of towers in the population would trend downwards over time as children have on average equal or fewer pieces than their parents, which would create serious problems for finding a valid solution.

Mutation: Mutation is performed by the genetic algorithm on 1% of the children it produces. Each puzzle has its own mutation implementation.

Puzzle #1: Mutation for this puzzle selects a random element in the array and substitutes a random available.

Puzzle #2: Mutation for this puzzle picks two arbitrary different indexes and swaps them around in the array

Puzzle #3: Mutation for this puzzle selects two random pieces in the tower and swaps their positions, and does not enforce any rules of validity.

Population size: The default population size used by the genetic algorithm is 100. However for puzzle 3 a population size of 500 was used as the initial population needed to be large enough to handle a large number of possible pieces.

Examples:

Puzzle #1:
Population: [5, 3, 3, 12, 6, 7, 12], [4, 2, 6, 8]

Selection: Parent 1 = [5, 3, 3, 2, 6, 7, 12], Parent 2 = [4, 2, 6, 8]

Because levenshteinDistance(P1, P2) > threshold we accept the parents, generate a child from them.

Randomly generated cutoff point 1: 6
Random generated cutoff point 2: 1

Subarray of parent 1 = [5, 3, 3, 12, 6, 7] (first 6 elements)
Subarray of parent 2 = [2, 6, 8] (last 3 elements)
Child = [5, 3, 3, 12, 6, 7, 2, 8]

It is assumed that 6 is only allowed once and 3 can appear twice. Then, ‘6’ already in subarray of parent 1 did not get added from the subarray of parent 2, as that would violate the properties of the puzzle.

Puzzle #2:
Population: [‘-3.2’, ‘5.0’, ‘6.7’, ‘-4.2’, ‘9.8’, ‘7.3’], [‘‘5.0’, ‘-4.2’, ‘9.8’, ‘6.7’, -3.2’, ‘7.3’]

Selection:
Parent 1: [‘-3.2’, ‘5.0’, ‘6.7’, ‘-4.2’, ‘9.8’, ‘7.3’]
Parent 2: [‘‘5.0’, ‘-4.2’, ‘9.8’, ‘6.7’, -3.2’, ‘7.3’]

Because levenshteinDistance(P1, P2) > threshold we accept the parents, generate a child from them

Cutoff value: 4
Parent 1: [‘-3.2’, ‘5.0’, ‘6.7’, ‘-4.2’]
Remove Parent 1 values from Parent 2:
Parent 2: [‘~~5.0~~’, ‘~~-4.2~~’, ‘9.8’, ‘~~6.7’~~, ~~-3.2’~~, ‘7.3’] = [‘9.8’, ‘7.3’]
Extend Parent 1 using Parent 2:
Child = [‘-3.2’, ‘5.0’, ‘6.7’, ‘-4.2’] + [‘9.8’, ‘7.3’] = [‘-3.2’, ‘5.0’, ‘6.7’, ‘-4.2’, ‘9.8’, ‘7.3’]

Puzzle #3:

Population: ['Door 100 100 2', 'Wall 3 3 2', 'Wall 4 2 2'], ['Wall 3 5 2', 'Lookout 3 1 2'], ['Wall 3 5 2', 'Wall 3 5 2', 'Wall 3 5 2', ‘Door 12 2 3’, ‘Wall 21 3 4’, 'Lookout 3 1 2'], [Door 3 5 2', 'Wall 3 5 2', 'Wall 3 5 2', ‘Door 12 2 3’, ‘Wall 21 3 4’, 'Lookout 3 1 2']

Selection: P1 = ['Wall 3 5 2', 'Wall 3 5 2', 'Wall 3 5 2', ‘Door 12 2 3’, ‘Wall 21 3 4’, 'Lookout 3 1 2'], P2 = [Door 3 5 2', 'Wall 3 5 2', 'Wall 3 5 2', ‘Door 12 2 3’, ‘Wall 21 3 4’, 'Lookout 3 1 2']

Threshold = max(len(p1), len(p2))*.35 = 2
Because levenshteinDistance(P1, P2) < threshold we reject these two parents and dont generate children from them and we select new parents.

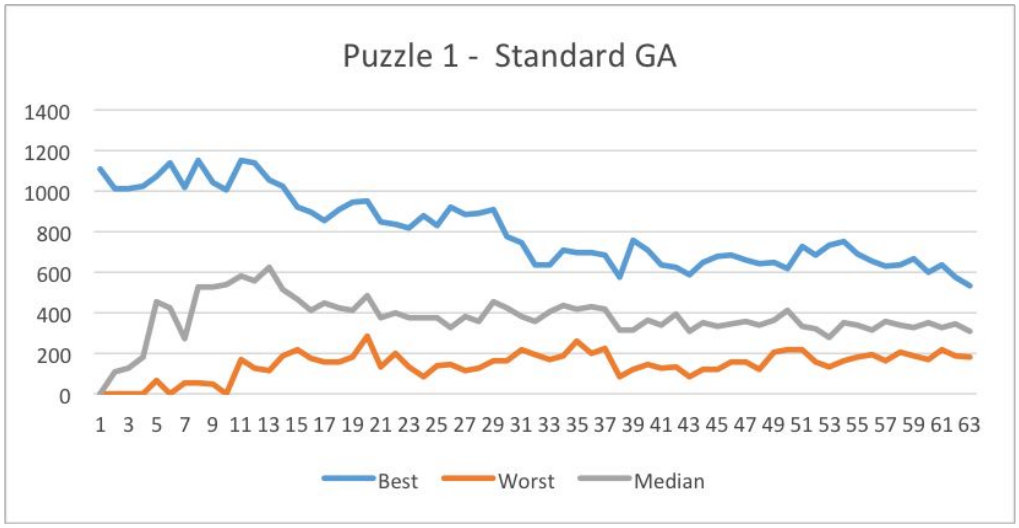
Selection: Parent 1: ['Door 100 100 2', 'Wall 3 3 2', 'Wall 4 2 2'], Parent 2: ['Wall 3 5 2', 'Lookout 3 1 2']

Because levenshteinDistance(P1, P2) > threshold we accept these two parents a generate a child from them

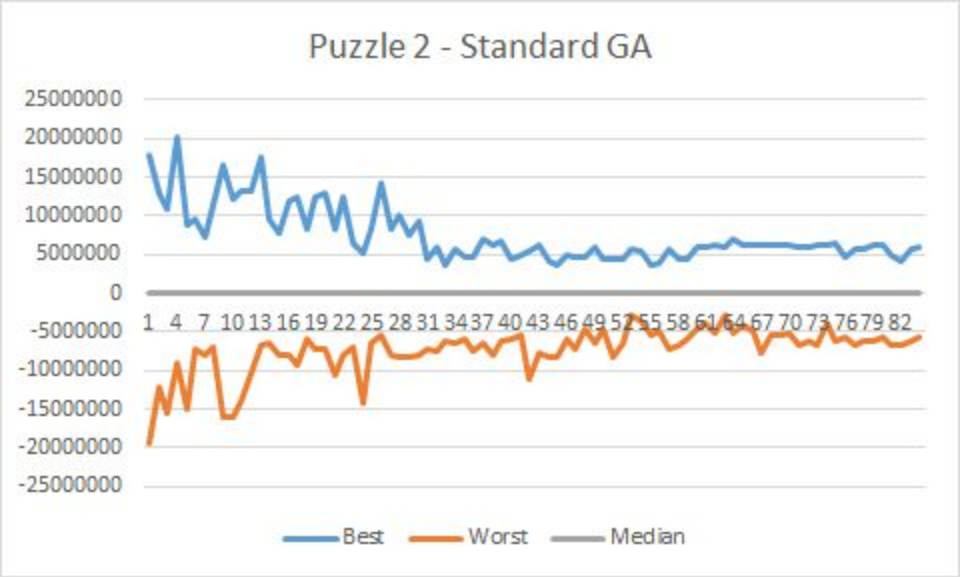
generate Random1() = 2
generate Random2() = 1
So, take the bottom 2 pieces of parent1, and the top 1 piece from parent2.
→ child: ['Door 100 100 2', 'Wall 3 3 2', 'Lookout 3 1 2']

For each of the 3 puzzles, provide a graph showing how solution quality varies as the number of generations increases:

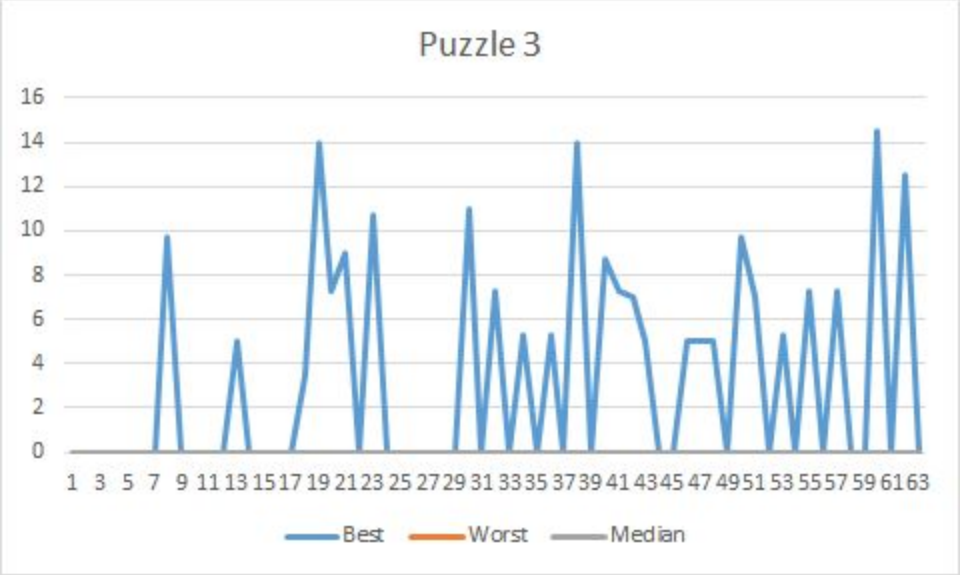
Puzzle #1: Number Packing



Puzzle #2: Number Allocation



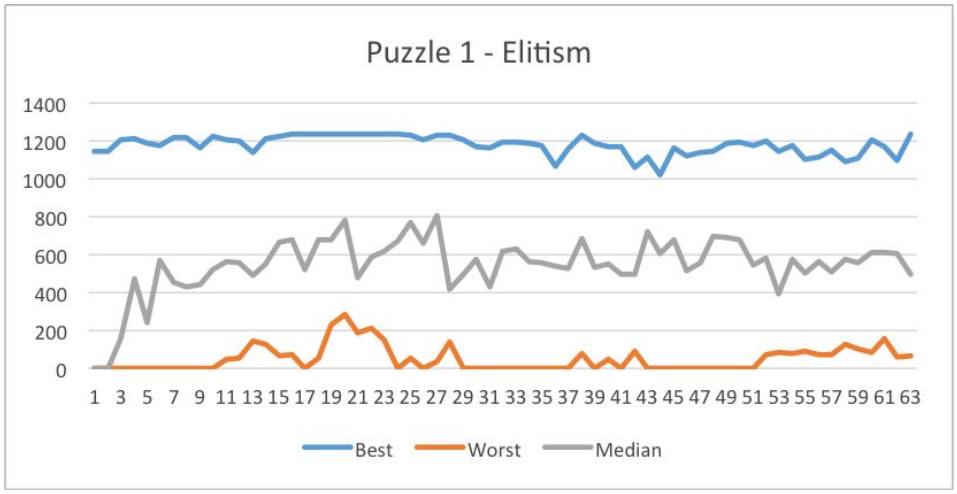
Puzzle #3: Tower Building



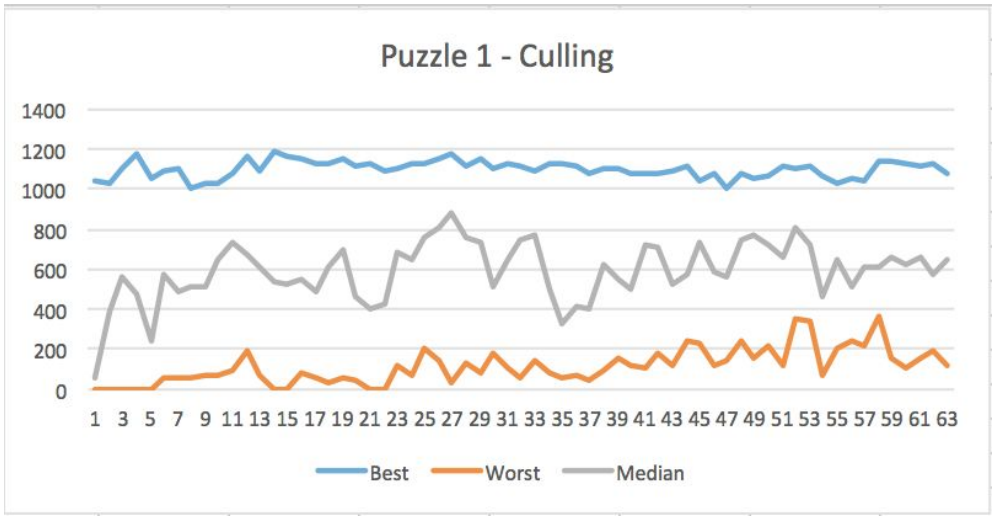
You must also experiment with a few simple techniques with GA (elitism and culling) and explain how they impact performance on the 3 puzzles.

Puzzle #1: Number Packing

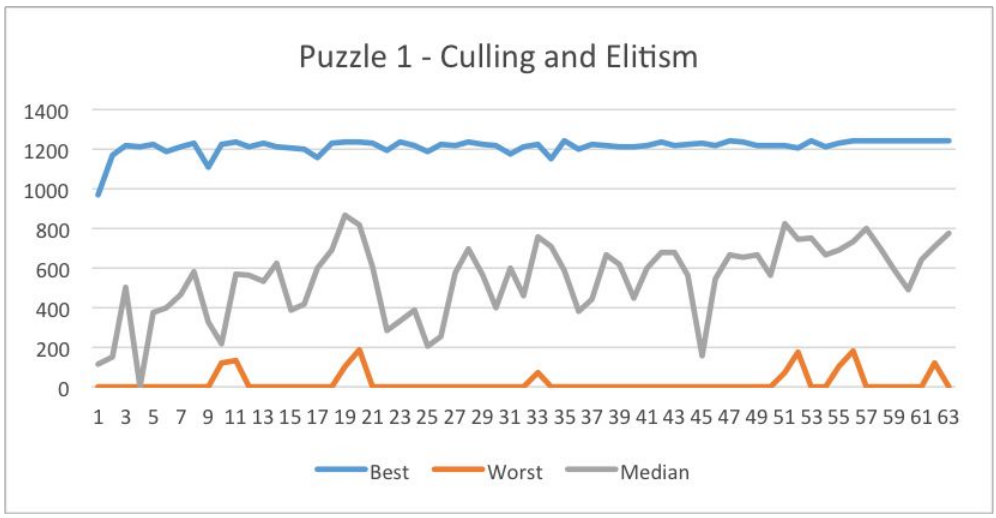
a. Elitism Graph



b. Culling Graph



c. Elitism/Culling Graph



d. Effect on 3 metrics

The simplicity and non-fixed length aspect of this problem made it difficult to clearly show the effects of the 3 metrics. The constraint of a total of 50 integers makes this a computationally easy problem which resulted in runs where the number of generations could reach 100,000. It is still possible to see some trends in the above graphs, even though the highest scoring solution often was found in the first 10% of the generations.

In this problem, elitism allowed the GA to quickly find solutions that were very close to the target by retaining the very best children that were often just a few points of the maximum. It was then a matter of getting these points by a crossover or a small mutation. Once the maximum had been found, this would also always be kept through subsequent generations. The best case would rise and stay around the target value. The worst case remains around 0 and the median would be right in between.

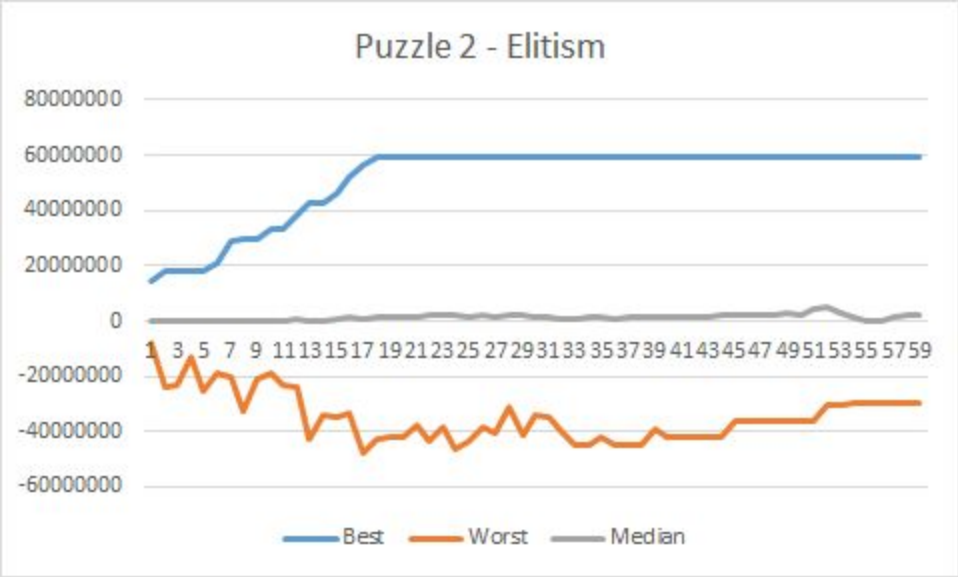
Culling, on the other hand, eliminates the worst children. This results in good solutions are bad solutions being mixed and only occasionally will the crossover of these result in something that is close to the maxima. The worst case would gradually get better the the best case would slowly approach the target value.

Culling and elitism work best together, keeping the best children while eliminating the worst scoring ones. One issue that these two combined helped solve was that of the non-fixed length property of the problem. Individuals that were either too long or too short were dropped by the culling and the best ones (usually of right length) were kept to be mutated and crossed over with high scoring individuals. The population would converge to consist of individuals with a length that could give the optimal solution. The best case would like with just elitism quickly reach the target value while the worst case stays around 0.

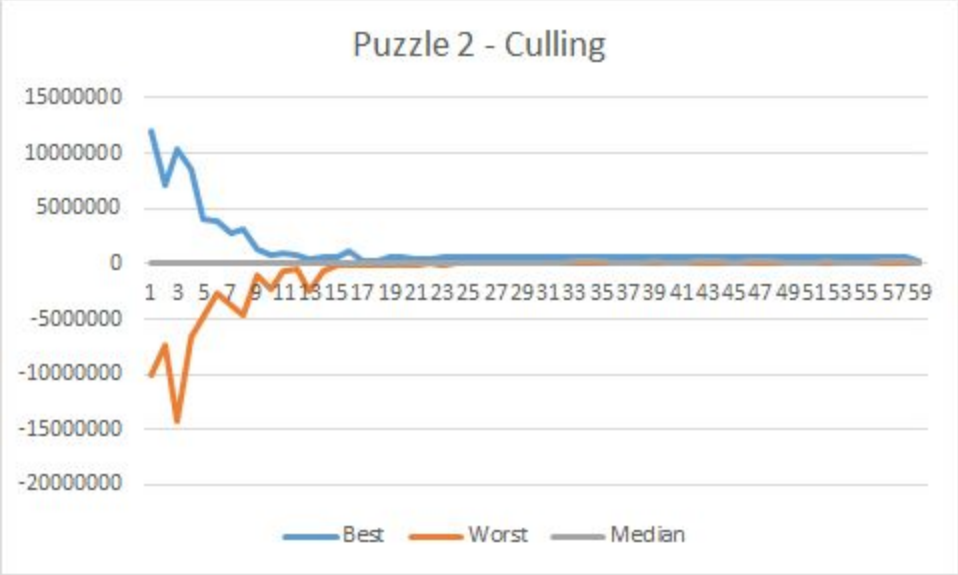
.

Puzzle #2: Number Allocation

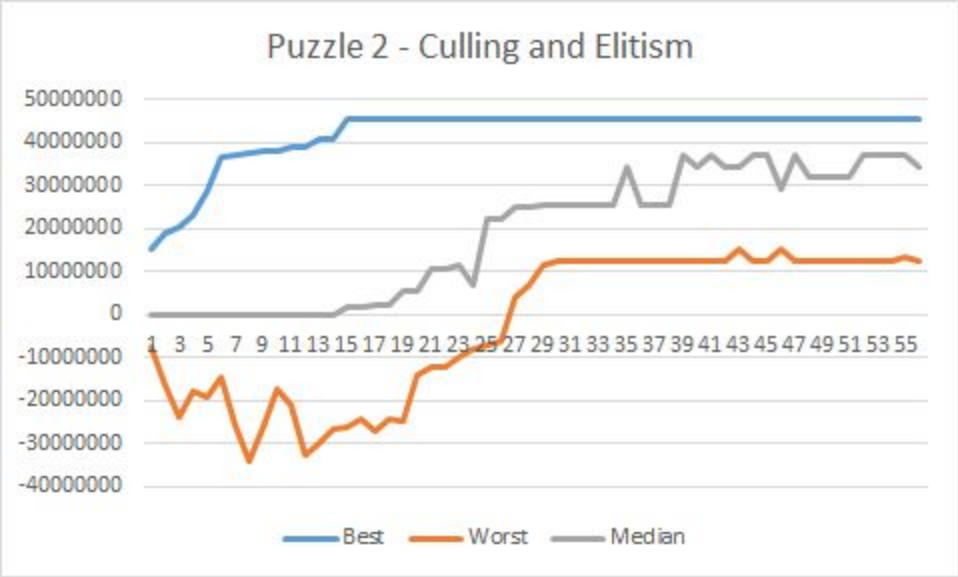
a. Elitism Graph



b. Culling Graph



c. Culling/Elitism Graph



d. Effect on 3 metrics

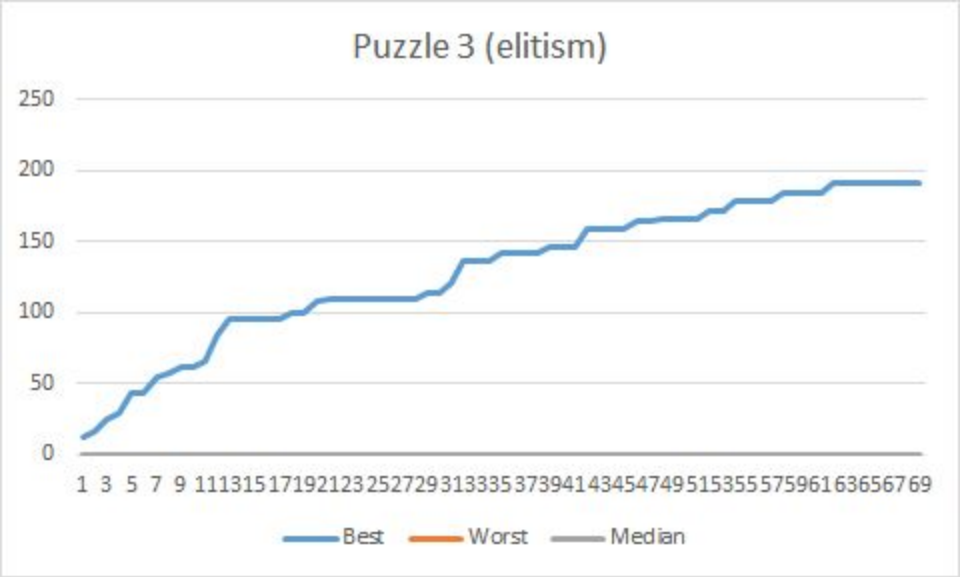
Elitism affects the subsequent generations by keeping better scoring individuals that provide higher maximum Best Scores in the long run, however it does nothing to improve Worst scores, actually allowing to find lower values.

On the other hand, Culling removes worst scoring individuals from the population, however it does nothing to improve the population and leads both Best and Worst scores into converging to the Median line.

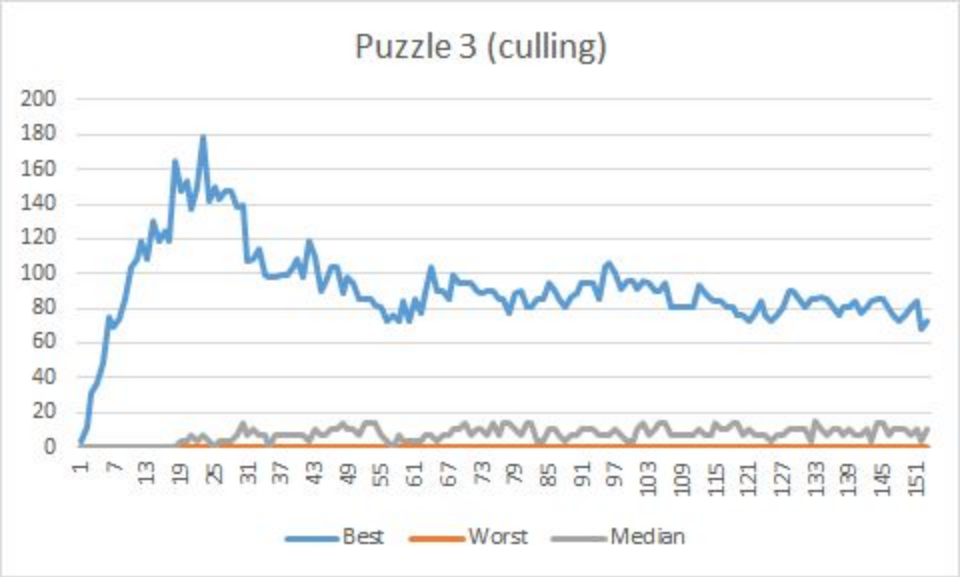
However, both Elitism and Culling metrics combined have an overall higher value for all 3 metrics, where both Best and Worst reach a plateau, with Best having its best scores so far and Worst being at a positive value.

Puzzle #3: Tower Building

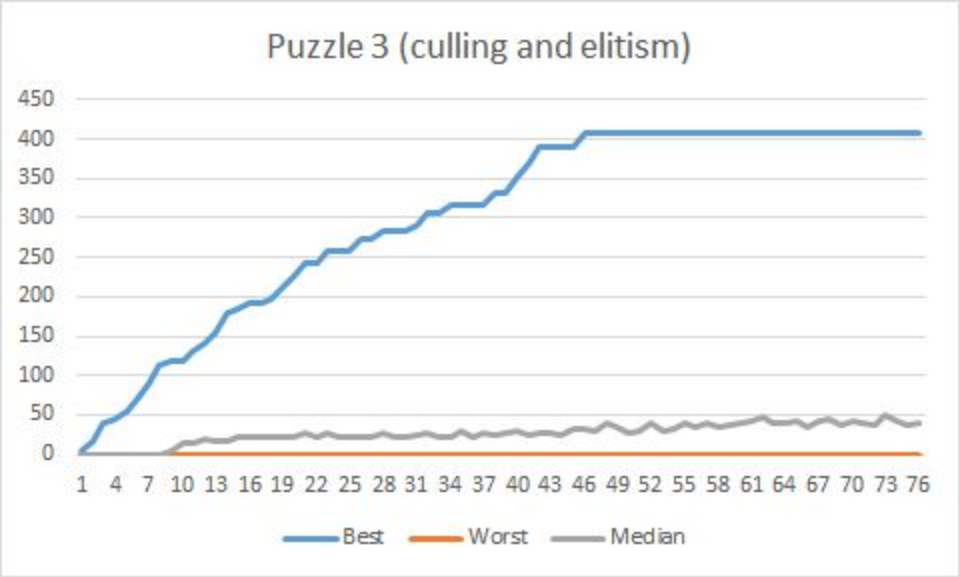
a. Elitism Graph



b. Culling Graph



c. Elitism/Culling Graph



d. Effect on 3 metrics

For all metrics, the minimum score of the generation ("Worst") remained 0, most likely due to the very strict constraints imposed by the scoring causing many children to be invalid.

Elitism allows the GA to gradually trend upwards (in the max) by retaining the very best children, although this slows and does not succeed in reaching the global maximum. However, because it does nothing to eliminate the worst children, both the median and minimum remain very low (0).

Culling has the opposite issue: the worst children are eliminated, causing the median to rise (though not the minimum). However, because the best solutions are mixed with poor solutions instead of being retained, they are diluted, which results in finding a local maximum very quickly, and then getting stuck in a much less optimal region.

Combining the two techniques creates an effective convergence to the optimal solution, as the best children are retained, the worst children are removed, and the maximum and median both are able to trend upwards. This is the only approach that found the global maximum.

You should explain how you implemented elitism and culling, and how you settled on that approach.

Culling: Culling was implemented by sorting the population by fitness and dropping the specified percentage of worst performing individuals. This was chosen because it is a fast method to remove the low performing individuals.

Elitism: Elitism was implemented by taking the sorted population and adding the top specified percentage to the new generation before generating children. This was selected because it is an easy method to maintain the top performing individuals.