Eliga Franks

Dr Wainwright

Evolutionary Computation

Sunday April, 23

Evolutionary Algorithms and its effect on 0/1 Knapsack

Introduction: The knapsack problem is a fundamental optimization problem that involves selecting a subset of items to maximize the value of the selected items while ensuring that the total weight of the selected items does not exceed a given capacity. This problem is motivated by scenarios where we have limited resources and must choose a subset of items to carry or pack in a way that maximizes their value. It must select a subset of items from a larger set to maximize a certain objective while satisfying certain constraints. In this essay we will be applying the knapsack problem to a SGA, SA, and FHC algorithm.

A SGA is a simple genetic algorithm that uses natural selection and genetics to solve problems. The problems evolve potential solutions over generations using techniques such as selection, crossover, and mutations. It uses a population of potential solutions that are randomly generated, and the best solutions are take and out fourth for new solutions. These new solutions are then mutated and added to the population and then repeated for generations until the optimal solution is found or that is what you're hoping for. FHC is an efficient algorithm for solving optimization problems, but most often get stuck in local optima. SA is used to find the global minimum of the given function. It uses heating and cooling to develop the optimal from an initial temperature. The speed of cooling or heating will determine the efficiency of the search potential for the optimal. The slower the temperature cools the better the solution can be. This allows the search space to be explored and escape local optimal. Then a new solution is accepted with probability based on temperature and the difference between the new and the current solution.

Chromosomes and Fitness: The 0/1 knapsack in our case is looked at as a chromosome representation where each gene represents an item that can be included in the knapsack or not. The chromosome has a length of n, where n is the number of items in the problem. Each gene can take on one of two possible values: 0, indicating that the corresponding item is not included in the knapsack, or 1, indicating that the corresponding item is included in the knapsack. The fitness of a chromosome is evaluated by calculating the total value of the items included in the knapsack while ensuring that the total weight of the items does not exceed the capacity of the knapsack. The goal is to maximize the fitness value of the chromosome. Infeasible solutions, which are solutions with a total weight exceeding the maximum allowed weight, are simply given a fitness score of 0, indicating that they are not valid solutions.

Operators: Tournament and rank selections were used. Tournament selection is performed by randomly selecting two individuals from the population and choosing the one with the highest fitness to be a parent. This process is repeated until the desired number of parents is obtained. Rank selection assigns a rank to each individual based on its fitness relative to others in the population. The probability of selecting an individual for reproduction is proportional to its rank. To perform rank selection, the code

first calculates the selection probabilities based on the fitness scores of each individual, then uses the random choices function to select parents based on these probabilities. These methods ensure that individuals with high fitness have a higher probability of being selected for reproduction. This increases the chances of producing offspring with better fitness.

For single-point crossover I set a random point along the chromosome to be selected and the genetic material from one parent up to that point is combined with the genetic material from the other parent after that point. In the code, this is accomplished by setting crossover_point1 to a random integer between 1 and chromosome_length - 1, where chromosome_length is the length of the chromosomes being crossed over. The offspring1 and offspring2 variables are then created by concatenating the genetic material from the two parents using this crossover point. For two point crossover I set two random points along the chromosome to be selected, and the genetic material between those two points is swapped between the parents. This is accomplished in the code by first selecting two random points, crossover_point1 and crossover_point2, as above. Then, if a random number is less than 1 (I use this to decide if I want single or double point), the two crossover points are swapped so that crossover_point1 is always less than crossover_point2. The offspring1 and offspring2 variables are then created by concatenating the genetic material from the parents using the two crossover points.

For my two mutataion operator I used insertion and flip. My method for insertion start off with a randomly generated bit, which can be either 0 or 1, with an equal probability. Then, a random position within the offspring chromosome is chosen to insert the new bit. The position is chosen uniformly at random from all possible positions within the chromosome. For flip mutation a random bit within the offspring chromosome is selected. This bit is then flipped, which means that its value is changed from 0 to 1 or from 1 to 0. The position of the bit to flip is chosen uniformly at random from all possible positions within the chromosome.

For my perturb functions I used flip and swap to implement small changes toward the local optima. My flip technique takes as input a current solution and an index, and creates a new solution by flipping the value of the bit at the specified index. The current_solution argument is a binary string or a list of binary values representing the solution. The index argument is an integer indicating the position of the bit to be flipped. The function creates a copy of the input solution using the copy method, flips the value of the bit at the specified index by subtracting it from 1, and returns the modified solution. For swap the function takes as input a current solution and two indices, and creates a new solution by swapping the values of the bits at the specified indices. The current_solution argument is a binary string or a list of binary values representing the solution. The index1 and index2 arguments are integers indicating the positions of the bits to be swapped. The function creates a copy of the input solution using the copy method, swaps the values of the bits at the specified indices using a tuple unpacking technique, and returns the modified solution.

Parameters used: I found that a 300 population would allow me to effectively create a diverse of solution by exploring more space, but not causing my program to run for a long time. For crossover rate I chose a 0.8 because I felt that passing on the desired traits could help lead to a stable and optimal solution. For mutation rate I just chose the standard 0.1. For termination I used generations because it would allow me to increase the search for the optimal when the problem size would increase. For α I chose 0.95, β to be 1.0 T0 and I0 to be 100 in the toy, small, and medium, but in the large a choose To and I0 to be 1000 because I wanted to have my simulated annealing program take its time

searching for the optimal based on the size of the number of values and weights. But I did want my program to computer forever, so I found what I believe to be the sweet spot. For β = (delta T) / (delta t) I thought that a constant cooling rate would solve the problem at the slowest cooling rate. This would help the problem not get stuck in sub optimal. I may touch on some details later when the time is right in terms of how they effect the data.

Dataset: For my SGA toy problem, listed as Data 1, I ran a set combination of two selections, crossovers, and mutations. I used a generation of 5 on this set even though it could find it in 1 generation. I found out that this allowed the solution to beat out majority of every once and a blue moon low fitness value. I kept a population of 300 but this was just a preference rather than needed. This allowed me to receive a optimal of 175 on all and only not receiving 5/5 optimal on the combination1. Keep in mind that I took a toy problem of size 11 (without the solution randomly inserted) and doubled that for a small size. Then the optimal solution randomly inserted again. This would be repeated for the toy problem small, medium, and large. The corresponding values would be 15, 26, 37, and 48. For the small size we changed the population size to 300. Again, this was more for a strategy to weed out the random errors, but not necessarily needed. We saw all the combinations get a 175 optimal and two not receive 5/5 optimal of each run. Please refer to Data 2 for this one. Moving on to medium, although a population size of 350 would solve this problem no problem I kicked it up a notch at 500 to help with random low fitness returns since this is semi randomly generated. Once again it received a best optimal of 175 and nearly all optimal received. Refer to Data 3 for this one. For large I put it at 1000 thinking it would need more generations to solve a bigger set problem, but it turned out that it's not so big in terms of NP size. I received a best optimal of 175 and nearly received all optimal for each run. Follow the general scheme and refer to the data below. You can also find the average fitness for each but in our case since they nearly are perfect it wasn't needed to explain though. Rather got a robust understanding of the runs through the best optimal and how many out of 5 runs were the optimal obtained. Keep in mind that in the case the combination of methods for this were hard to evaluate since found the optimal in similar results. If I could redo it all over again and if the time allowed, I would like to do small at 37, a medium at 103, and a large at 169 sizes. This would hopefully allow me to see results that differ from each other in terms of the techniques used for each.

Moving on to my Foolish Hill Climb the data really isn't worth going over since it wasn't optimal nearly 100 percent of the time, but I would like to say it does give the optimal, but this would maybe be like 2 out of 100 runs. From a data analysis perspective this is almost useless, but cool to interpret.

The SA was personally my favorite to implement. For my toy problem flip received average of 167 with the optimal being obtained 4/5. Keep in mind the optimal is still 175. Now swap was perfect all around. It received the optimal perfectly every time. For The medium set the same data was obtained for each outside of a one-point increase in the flip mutations average. This was replicated the same in the medium and large with only the flip mutations average increasing or decreasing by one or two points.

Computational Results: For viewing the graphical analysis, please view the figures below that correspond to the order of the algorithms listed. For the comparisons of the different sizes taken on the SGA we could not determine which methods would be better than the other since the data collected was very similar. If time would permit, I would go on to look at larger lengths to see if the data would vary. You can assume that as these lengths get large that the problem would get increasingly hard but as an NP we must declare what would that large threshold be. For the FHC we also cannot compare the

methods since the data is sporadic and only returns the optimal maybe 2 out of 100 runs. We can conclude that our values were getting cause in sub optimal level since we know that it does give the fitness back at a very small percentage. Evaluating the SA we can clearly see the winner between flip and swap mutation. Swap was found to find the optimal solution 100 percent of the time while flip is only 80 percent accurate. This means that the better mutation method would be swap. Flip mutation generally isn't a very good solution in terms of problems that require a specific value in a solution, which is the knapsack problem in a nutshell. Also, if the bit string is too small or too large it could mess with the calculations.

Conclusions: In conclusion, the knapsack problem is a fundamental optimization problem that involves selecting a subset of items to maximize the value of the selected items while ensuring that the total weight of the selected items does not exceed a given capacity. In this project, we explored various techniques for solving the 0/1 knapsack problem using genetic algorithms. We used chromosome representations, fitness evaluations, selection operators, crossover operators, mutation operators, and perturb functions to evolve a population of candidate solutions toward the optimal solution.

We experimented with different parameters to determine the best settings for the genetic algorithm, and found that a population size of 300, crossover rate of 0.8, mutation rate of 0.1, and generations as the termination criteria were effective for finding good solutions. We also used a cooling schedule with α=0.95, β=1.0, and initial temperature T0=100 to explore the solution space and converge toward the optimal solution.

The results obtained from the experiments showed that genetic algorithms can be an effective method for solving the knapsack problem, especially for larger problem sizes. However, the algorithm's performance is highly dependent on the choice of parameters and the quality of the initial population. Further research can explore other techniques for initializing the population and adjusting the parameters to improve the algorithm's performance.

Data 1: Toy
SGA

5 runs per	Selection	Crossover	Mutation	Population	Max generations	Best Optimal	Average	Optimal Received
Combination1	Tournament	One Point	Insert	300	5	175	171	(4/5)
Combination2	Tournament	One Point	Flip	300	5	175	175	(5/5)
Combination3	Tournament	Two Point	Insert	300	5	175	175	(5/5)
Combination4	Tournament	Two Point	Flip	300	5	175	175	(5/5)
Combination5	Rank	One Point	Insert	300	5	175	175	(5/5)
Combination6	Rank	One Point	Flip	300	5	175	175	(5/5)
Combination7	Rank	Two Point	Insert	300	5	175	175	(5/5)
Combination8	Rank	Two Point	Flip	300	5	175	175	(5/5)

Data 2: Small
SGA

5 runs per	Selection	Crossover	Mutation	Population	Max generations	Best Optimal	Average	Optimal Received
Combination1	Tournament	One Point	Insert	300	250	175	175	(5/5)
Combination2	Tournament	One Point	Flip	300	250	175	175	(5/5)
Combination3	Tournament	Two Point	Insert	300	250	175	175	(5/5)
Combination4	Tournament	Two Point	Flip	300	250	175	175	(5/5)
Combination5	Rank	One Point	Insert	300	250	175	171	(4/5)
Combination6	Rank	One Point	Flip	300	250	175	175	(5/5)
Combination7	Rank	Two Point	Insert	300	250	175	171	(4/5)
Combination8	Rank	Two Point	Flip	300	250	175	175	(5/5)

Data 3:
Medium SGA

5 runs per	Selection	Crossover	Mutation	Population	Max generations	Best Optimal	Average	Optimal Re
Combination1	Tournament	One Point	Insert	300	500	175	175	(5/5)
Combination2	Tournament	One Point	Flip	300	500	175	175	(5/5)
Combination3	Tournament	Two Point	Insert	300	500	175	171	(4/5)
Combination4	Tournament	Two Point	Flip	300	500	175	175	(5/5)
Combination5	Rank	One Point	Insert	300	500	175	171	(4/5)
Combination6	Rank	One Point	Flip	300	500	175	175	(5/5)
Combination7	Rank	Two Point	Insert	300	500	175	175	(5/5)
Combination8	Rank	Two Point	Flip	300	500	175	175	(5/5)

Data 4: Large
SGA

5 runs per	Selection	Crossover	Mutation	Population	Max generations	Best Optimal	Average	Optimal Received
Combination1	Tournament	One Point	Insert	300	1000	175	175	(5/5)
Combination2	Tournament	One Point	Flip	300	1000	175	175	(5/5)
Combination3	Tournament	Two Point	Insert	300	1000	175	175	(4/5)
Combination4	Tournament	Two Point	Flip	300	1000	175	175	(5/5)
Combination5	Rank	One Point	Insert	300	1000	175	171	(4/5)
Combination6	Rank	One Point	Flip	300	1000	175	175	(5/5)
Combination7	Rank	Two Point	Insert	300	1000	175	171	(4/5)
Combination8	Rank	Two Point	Flip	300	1000	175	175	(5/5)

Data 5: FHC Toy

5 runs per	Perturbations	Best Optimal	Average	Optimal Received
Combination1	Flip	170	127	(0/5)
Combination2	Swap	160	210	(0/5)

Data 6: FHC Small

5 runs per	Perturbations	Best Optimal	Average	Optimal Received
Combination1	Flip	130	152	(0/5)
Combination2	Swap	175	196	(1/5)

Data 7: FHC Medium

5 runs per	Perturbations	Best Optimal	Average	Optimal Received
Combination1	Flip	155	158	(0/5)
Combination2	Swap	160	222	(0/5)

Data 8: FHC Large

5 runs per	Perturbations	Best Optimal	Average	Optimal Received
Combination1	Flip	80	167	(0/5)
Combination2	Swap	160	155	(0/5)

SA 8: Toy

5 runs per	Perturbations	α	β	T_0	I_0	Best Optimal	Average
Combination1	Flip	1.0	0.95	100	100	175	167
Combination2	Swap	1.0	0.95	100	100	175	175
Optimal Received							
(4/5)							
(5/5)							

SA 9: Small

5 runs per	Perturbations	α	β	T_0	I_0	Best Optimal	Average
Combination1	Flip	1.0	0.95	100	100	175	166
Combination2	Swap	1.0	0.95	100	100	175	175
Optimal Received							
(4/5)							
(5/5)							

SA 10: Medium

5 runs per	Perturbations	α	β	T_0	I_0	Best Optimal	Average
Combination1	Flip	1.0	0.95	1000	1000	175	167
Combination2	Swap	1.0	0.95	1000	1000	175	175
Optimal Received							
(4/5)							
(5/5)							

SA 11: Large

5 runs per	Perturbations	α	β	T_0	I_0	Best Optimal	Average
Combination1	Flip	1.0	0.95	1000	1000	175	165
Combination2	Swap	1.0	0.95	1000	1000	175	175
Optimal Received							
(4/5)							
(5/5)							

Figure 1: SGA Toy Problem

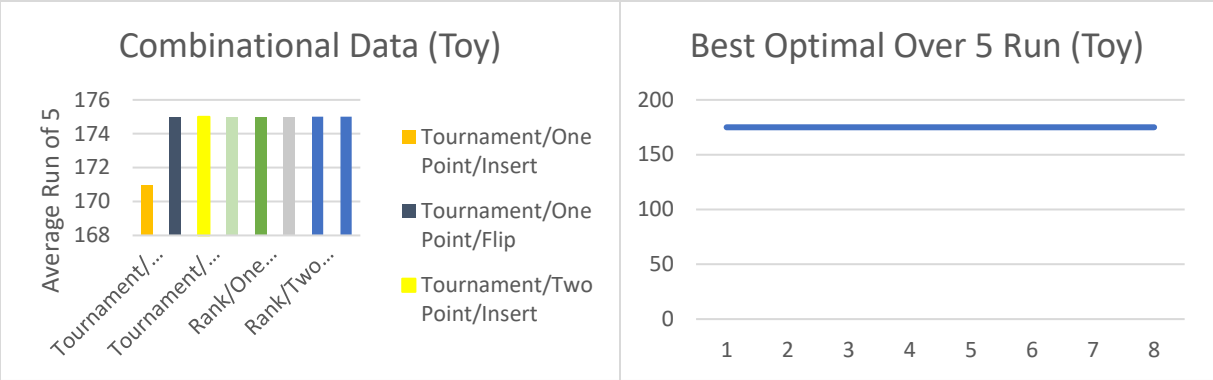

Figure 2: SGA Small Problem

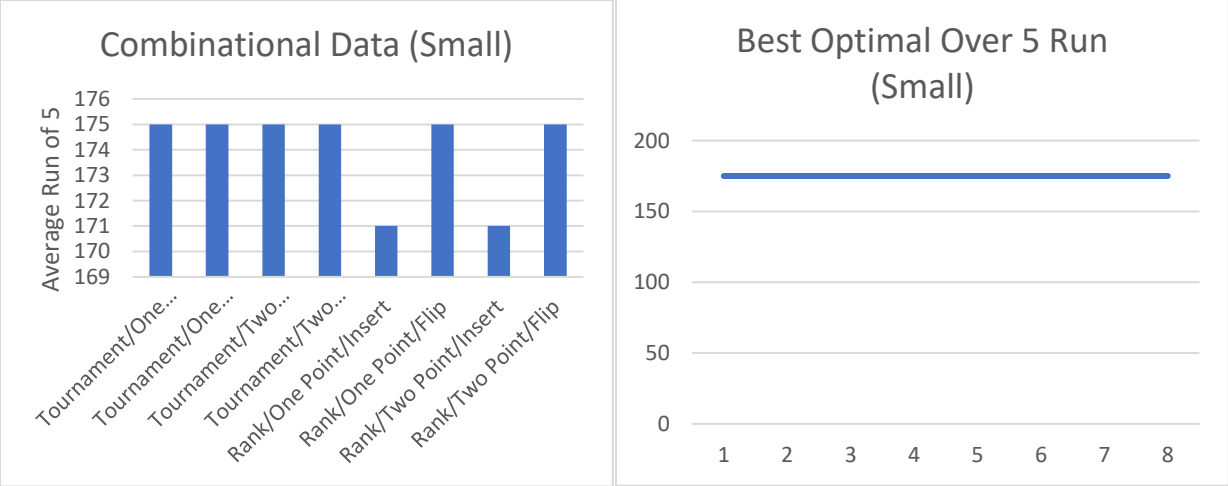

Figure 3: SGA Medium Problem

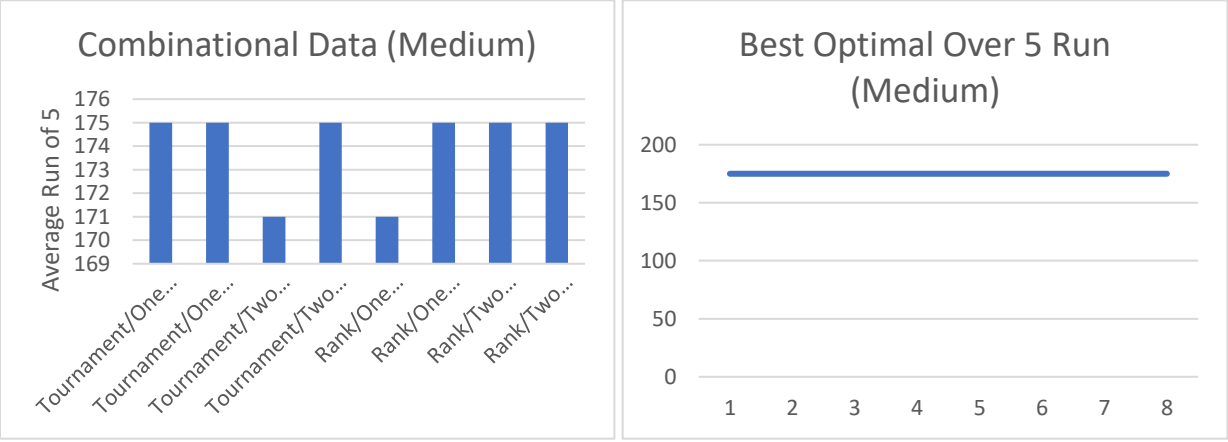

Figure 4: SGA Large Problem

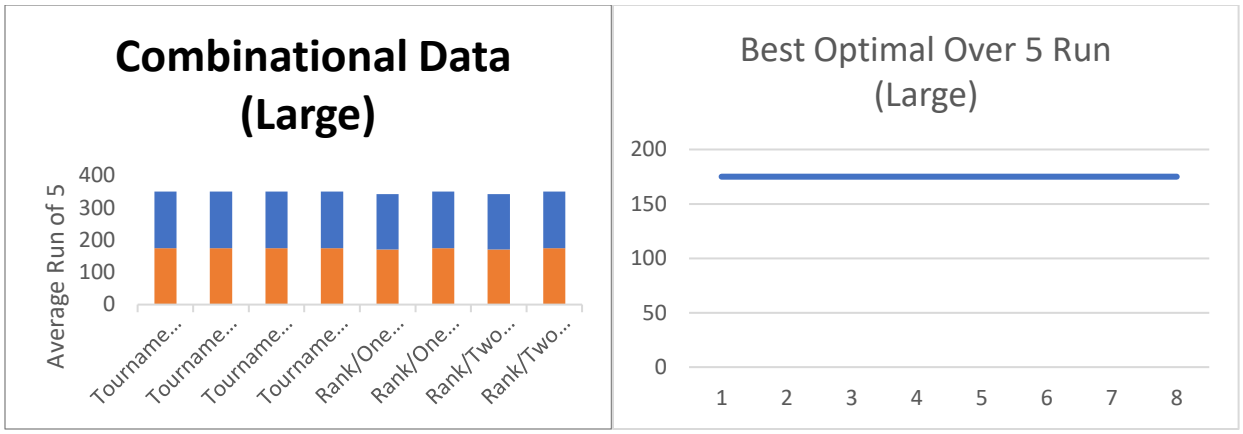

Figure 5: FHC Toy Problem

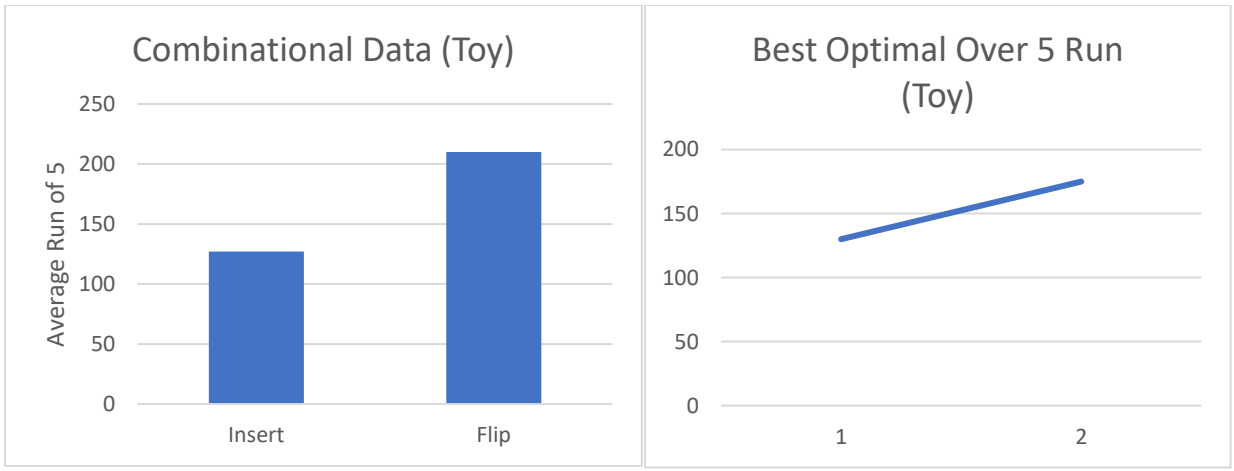

Figure 6: FHC Small Problem

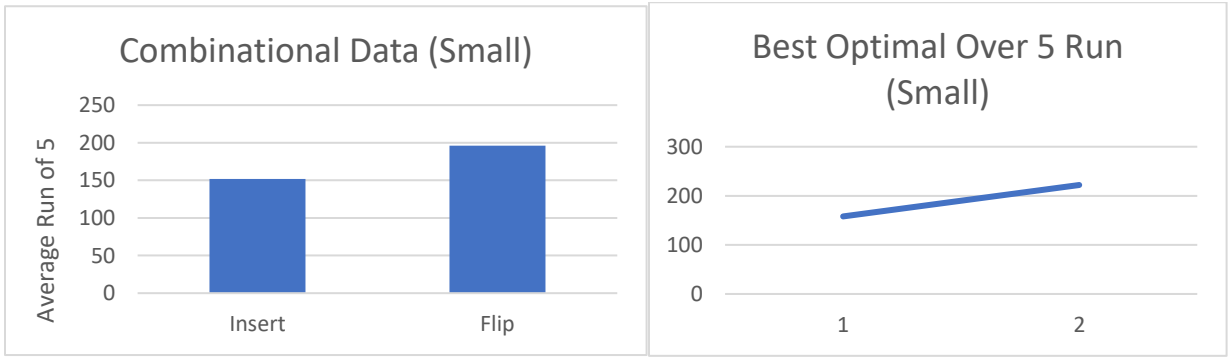

Figure 7: FHC Medium Problem

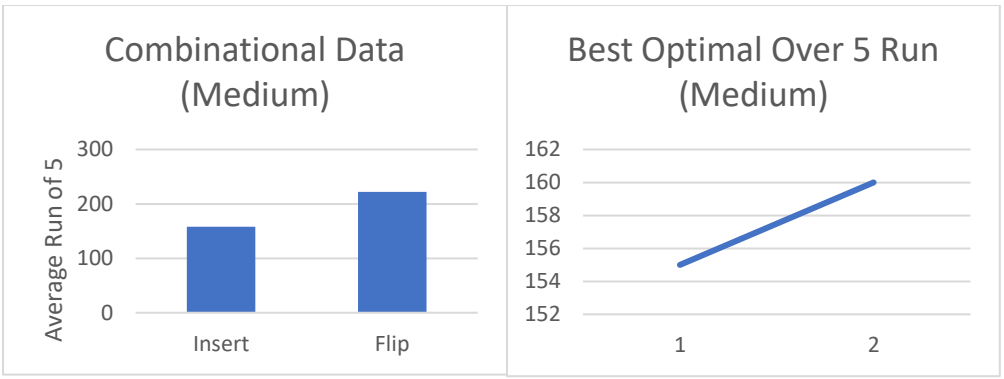

Figure 8: FHC Large Problem

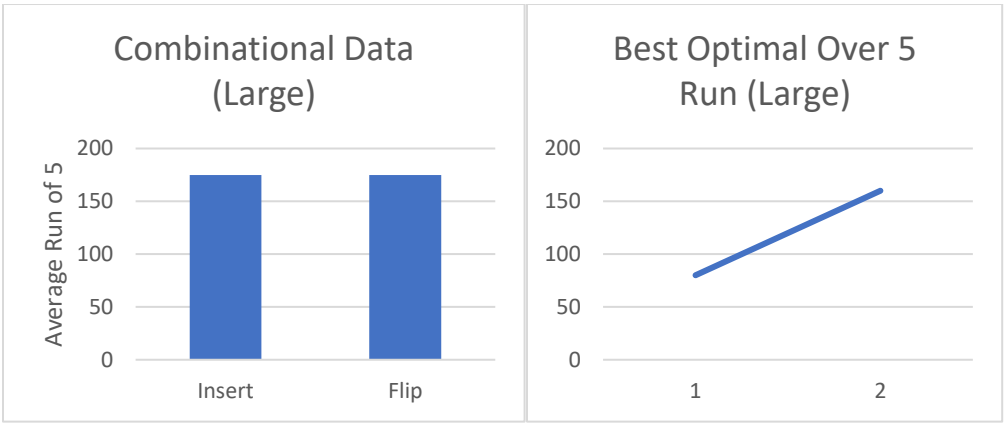

Figure 9: Simulated Annealing Toy Problem

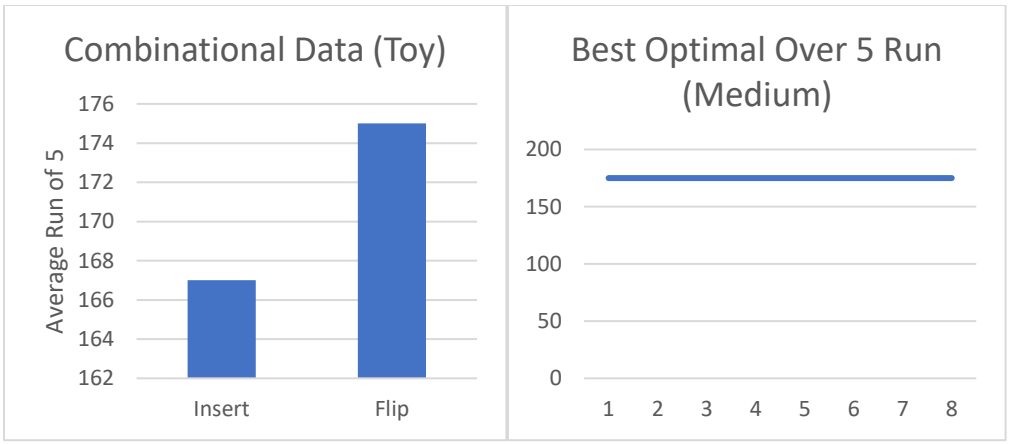

Figure 10: Simulated Annealing Small Problem

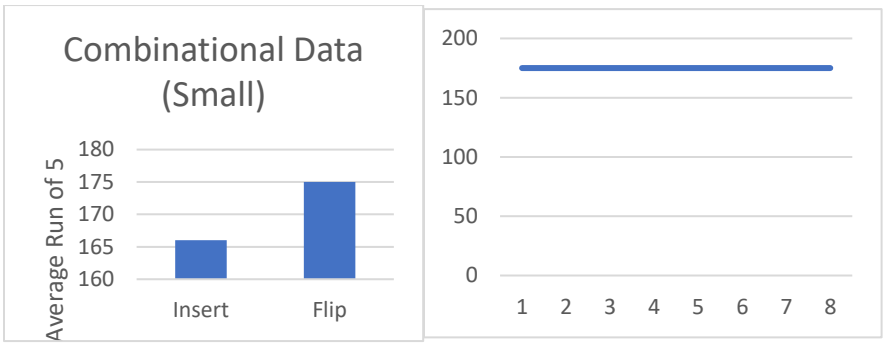

Figure 11: Simulated Annealing Medium Problem

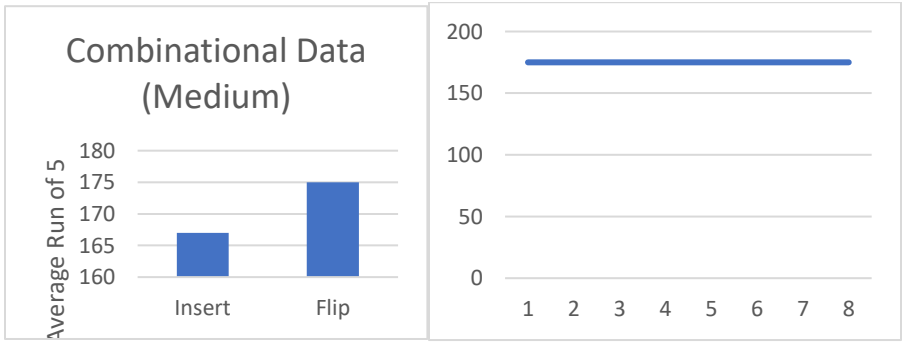

Figure 12: Simulated Annealing Large Problem