Metaheuristic Optimization – Assignment 1

Name: Shane Quinn Date: 27/10/2020

Student Number: R00144107

Part 1: NP-Completeness

1. Convert F to 3SAT (Final number in student ID = 7 – Question C on assignment specification)

$$F = (\neg p_1 \vee \neg p_3) \wedge (p_1 \vee \neg p_2 \vee p_3 \vee \neg p_4 \vee p_5 \vee \neg p_6)$$

$$F' = (\neg p_1 \vee \neg p_3 \vee U_1) \wedge (\neg p_1 \vee \neg p_3 \vee \neg U_1) \wedge (p_1 \vee \neg p_2 \vee U_2) \wedge (\neg U_2 \vee p_3 \vee U_3) \wedge (\neg U_3 \vee \neg p_4 \vee U_4) \wedge (\neg U_4 \vee p_5 \vee \neg p_6)$$

$$Solution => p_1 = F, p_2 = F, p_3 = T, p_4 = F, p_5 = T, p_6 = F, U_1 = T, U_2 = T, U_3 = T, U_4 = T$$

$$F' = (\neg F \vee \neg T \vee T) \wedge (\neg F \vee \neg T \vee \neg T) \wedge (F \vee \neg F \vee T) \wedge (\neg T \vee T \vee T) \wedge (\neg T \vee \neg F \vee T) \wedge (\neg T \vee T \vee \neg F)$$

$$F' = (T) \wedge (T) \wedge (T) \wedge (T) \wedge (T) \wedge (T)$$

$$F' = T$$

$$F = (\neg p_1 \vee \neg p_3) \wedge (p_1 \vee \neg p_2 \vee p_3 \vee \neg p_4 \vee p_5 \vee \neg p_6)$$

$$F = (\neg F \vee \neg T) \wedge (F \vee \neg F \vee T \vee \neg F \vee T \vee \neg F)$$

$$F = (T \vee F) \wedge (F \vee T \vee T \vee T \vee T \vee T)$$

$$F = (T) \wedge (T)$$

$$F = T\ (Valid\ Solution)$$

2. First letter of first name = 'S' – Convert the second and third clauses of F' to a 3Col graph.

$$F' = (\neg p_1 \vee \neg p_3 \vee U_1) \wedge (\neg p_1 \vee \neg p_3 \vee \neg U_1) \wedge (p_1 \vee \neg p_2 \vee U_2) \wedge (\neg U_2 \vee p_3 \vee U_3) \wedge (\neg U_3 \vee \neg p_4 \vee U_4) \wedge (\neg U_4 \vee p_5 \vee \neg p_6)$$

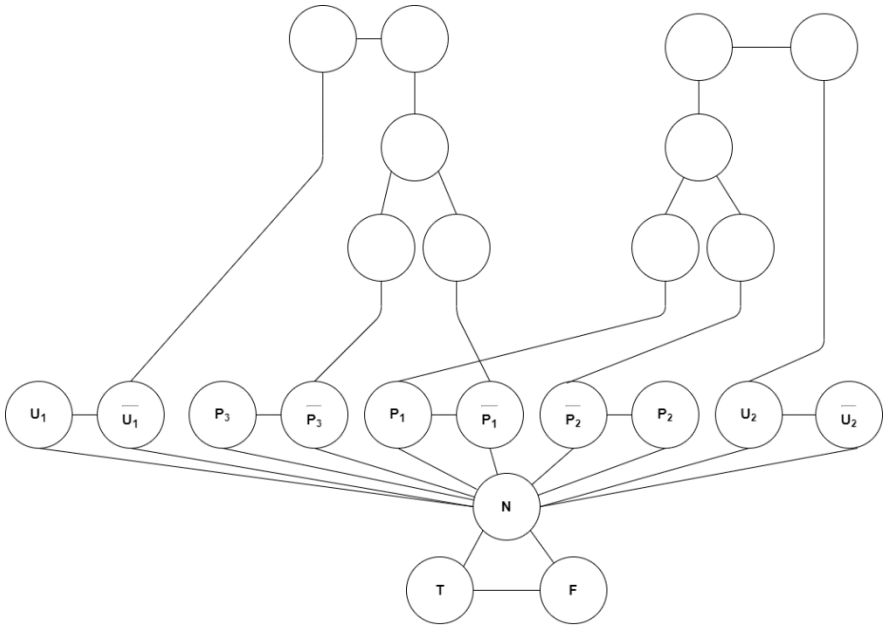

Figure 1 – 3Col Graph

Part 2: Genetic Algorithms

Genetic Algorithms are used to solve computationally difficult problems such as combinatorial optimization problems, which are a class of NP-hard problems (they cannot be described in polynomial time). The NP-hard problem that we are looking at for this assignment is the travelling salesman problem which can be described as the following:

"If a traveling salesman wishes to visit exactly once each of a list m cities (where the cost of traveling from city i to city j is cij) and then return to the home city, what is the least costly route the traveling salesman can take." [1]

Figure 2 shows the basic genetic algorithm flow that we implemented.

1. We start by creating an initial population (the size of the initial population is specified by the user).
2. We calculate the fitness of each individual in the population, their fitness is measured by the total distance to visit each city and return to the first city.
3. Using a proportional selection method, we create our mating pool, the method we use to achieve this is stochastic universal sampling.
4. 2 parents are then selected; we use binary tournament selection in our implementation.
5. The 2 selected parents are then fed into a crossover function which amalgamates the two parents to create 2 offspring. We use 'Order-1' and 'Uniform' crossover.
6. The offspring are then mutated. The probability of mutation occurring for each child is set by the user as the mutation rate. We use 'Scramble' and 'Inversion' mutation.
7. The fitness of each child is computed
8. The individual with the best fitness is saved
9. The offspring then replace current individuals in the population
10. Steps 3 – 9 are repeated until we reach the max iterations (which is set by the user)

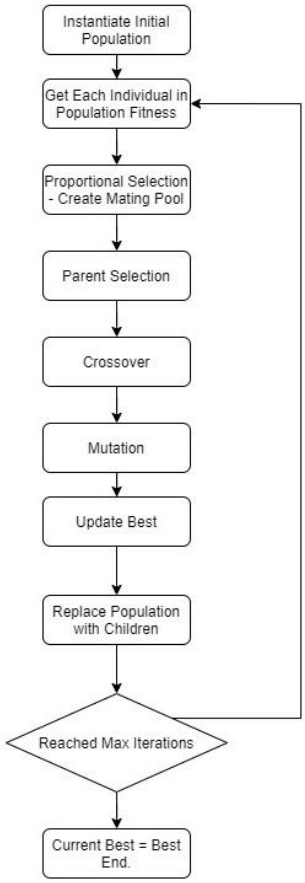

Figure 2 - Genetic Algorithm Flow

The travelling salesman problem requires our crossover and mutation operators to interchange their current chromosomes, never adding or removing chromosomes. This is because the problem stipulates that each city must be visited once (except the first city, the distance to the first city from the last city visited is added to the fitness of each Individual)

In the next couple of sections, we'll explore the different methodologies implemented in our solution. After this we'll evaluate the differences, pros and cons and how experimentation can lead to a better overall solution.

Crossover

Crossover is a method of combining 2 parent genes to create 1 or 2 child genes (or in our case 'routes' for our travelling salesman). Here we will discuss the types of crossover being implemented in this project. For both crossover types implemented, we create 2 offspring.

Order-1 Crossover

Order-1 crossover is achieved by selecting range from parent A, removing these chromosomes from parent B and appending them onto parent B in the order the appear in parent A creating child A. In the example below we will examine creating one child from Parent A and Parent B

$$Parent\ A - [0,5,7,4,6,8,3,9,1,2]$$

$$Parent\ B - [1,8,6,4,9,7,0,2,3,5]$$

We select a random index range (in this case 3 – 5, highlighted) from parent A. These chromosomes are removed from parent B giving the following:

$$Child\ A\ (Parent\ B) - [1,4,9,7,0,2,5]$$

The removed chromosomes are then appended to parent B (in our case in the order they appear in parent A).

$$Child\ A - [1,4,9,7,0,2,5,6,8,3]$$

The same steps are taken to create a second child but the other way around.

Uniform Crossover

Uniform crossover is achieved by randomly selecting chromosomes in our parent genes, removing them and appending them onto the chromosomes left, in the order that they appear in the other parent. Below in parent A, we remove the highlighted chromosomes.

$$Parent\ A - [0,5,7,4,6,8,3,9,1,2]$$

These are removed giving us the following:

$$Parent\ A - [0,7,4,8,3,1,2]$$

The removed chromosomes are then appended onto the remaining chromosomes in the order that they appear in the other parent creating the Child A

$$Parent\ B - [1,8,6,4,9,7,0,2,3,5]$$

$$Child\ A - [0,7,4,8,3,1,2,6,9,5]$$

This process is repeated for parent B, giving us child B.

Mutation

Mutation is the process of slightly changing the genes of one individual, the reason for mutation is to introduce a certain degree of randomness into our algorithm. This can help prevent our results from plateauing at a false optimal solution.

Too high a mutation rate can lead to more of a random solution where we can lose optimality. A general rule of thumb is to set the mutation rate to a small number at first, like 1/N, where N is the number of individuals in the population (on average mutation only occurs once for each iteration), the developer can then experiment and alter this depending on results. With a mutation rate of 0.05, each child has a 5% chance of being mutated as described below.

Inversion

Inversion mutation is a form of mutation where a range of chromosomes are selected and reversed. Consider the following.

$$Individual - [0,5,7,4,6,8,3,9,1,2]$$

The highlighted chromosomes are inverted.

$$Individual\ after\ mutation - [0,5,7,9,3,8,6,4,1,2]$$

Scramble

Scramble mutation is a form of mutation where a range of chromosomes are selected and shuffled. Consider the following:

$$Individual - [0,5,7,4,6,8,3,9,1,2]$$

The highlighted chromosomes are inverted.

$$Individual\ after\ mutation - [0,5,7,8,6,9,3,4,1,2]$$

Initial Population

We explore two ways of generating our initial population. The first is randomly, in this method we pick an initial location at random and select every location thereafter at random, consider the diagram below.

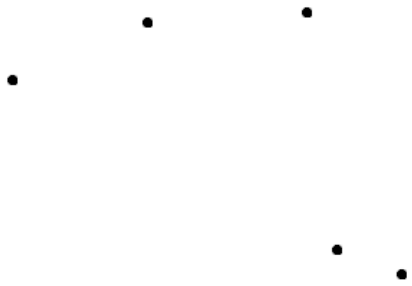

Figure 3 – TSP Route

If each dot above represents a location, a random initial population selection might look something like this.

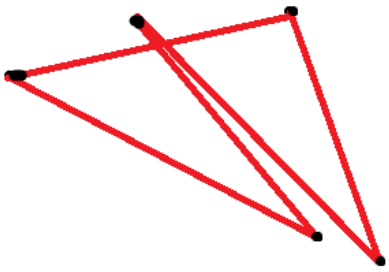

Figure 4 – Random Initial Population

As the goal of the travelling salesman problem is to find the shortest route possible, we can implement a heuristic to find a more optimal initial solution. The heuristic we implemented was nearest neighbour. We pick an initial location at random, for example calling it 'LocA' and we find the closest location to 'LocA' that has not been visited yet, we repeat this until we've visited each location and then return to the starting location. A nearest neighbour initial population solution might look something like below,

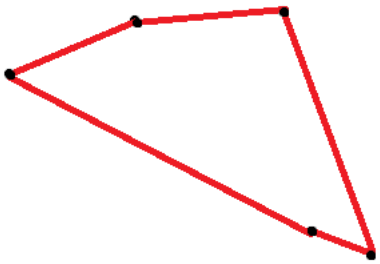

Figure 5 – Nearest Neighbour Initial Population

Our nearest neighbour heuristic uses the Euclidean distance to compute each nearest location. The clear benefit of using a heuristic such as nearest neighbour to generate our initial population is that the initial solution is closer to optimal than it would be if we used random population selection. The drawback is nearest neighbour heuristic takes slightly longer than random population selection to execute, we discuss this further and investigate empirical results in a later section.

Stochastic Universal Sampling

Stochastic universal sampling is a method by which the probability of parent individuals being chosen is dependent on their fitness. The higher an Individual's fitness is, the more likely it is to be entered into the mating pool. Below, in Figure 6, we can see how stochastic universal sampling is used. In this example we are selecting 6 individuals, notice how each of the 6 'Pointers' are evenly spaced, while the line is divided unevenly. Each line segment represents an individual to be selected, the higher their fitness the larger the line segment. The first 'Pointer' position is randomly selected as a point between 0 and 1/N (where N is the number of Individuals to be selected, in this case 6). Each other pointer is the sum of the previous pointer(s) and 1/N.

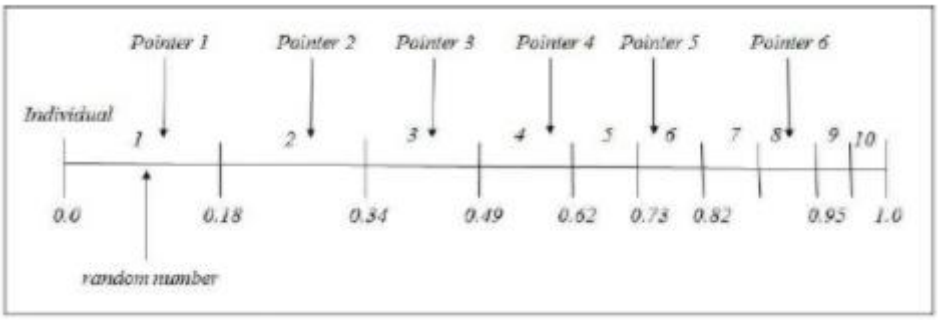

Figure 6 – Stochastic Universal Sampling [1]

The approach I took instead of using the range 0 – 'Total Population Size', I used the range 0 – 1 and segmented this line relative to each individual's fitness. Each point on the 'Sampling' line is then 1/'Total Population Size' distance apart from each other (see accompanied code).

Some alternatives include random sampling and roulette wheel sampling. Random sampling is where the mating pool is chosen at random from the population. Roulette wheel sampling is where a 'roulette wheel' is divided into segments, each segment represents an individual and the size of the segment is representative of the individual's fitness. The wheel is the spun N times (where N is the number of Individuals to be selected).

The benefits of Stochastic Universal Sampling ensure that 'fitter' Individuals are chosen for mating on each iteration.

Selection

When the mating pool is created, we then focus on selecting 2 parents for feeding into our crossover function of choice, detailed above, we do this N/2 times. Each of our crossover implementations generate 2 children, so to generate N children we must call our chosen crossover function N/2 times for each iteration.

Random Selection

Random selection is self-explanatory and the simplest form of selection, 2 parents are chosen from the mating pool.

Binary Tournament Selection

Binary Tournament Selection is slightly more complicated, here we pit two individuals against each other, the fitter individual is then selected. We do this twice in our case, to select 2 parents. So, 4 individuals are chosen at random from our mating pool. The fitter of the first two individuals and the fitter of the second 2 individuals are selected for crossover.

Binary Tournament Selection, while being slightly more taxing on compute resources, has the benefit choosing the fitter individuals for crossover, leading to a more optimized solution.

Basic Evaluation

Here we'll evaluate some of the configurations we implemented, paying attention to the impact of using different crossover and mutation methods. Note, for each configuration we use Binary Tournament Selection to generate our mating pool from the population. We also used the following as standard throughout each evaluation so we could make valid comparisons

- Population Size = 100
- Mutation Rate = 0.05
- Max Iterations = 100
- Number of Runs = 5

Below are configurations 1 and 2.

Config	Initial Population	Crossover	Mutation
1	Random	Order-1	Inversion
2	Random	Uniform	Scramble

Table 1 – Configurations 1 and 2

As specified in the assignment outline, the first letter of my surname is 'Q' therefore I used inst-4.tsp, inst-16.tsp, and inst-6.tsp for my analysis. For reference:

- Inst-4.tsp = Small
- Inst-16.tsp = Medium
- Inst-6.tsp = Large

For config 1 and 2 we randomly create our first population; this has the benefit of being faster than using a heuristic to determine our first population however this method gives us a less optimal starting position than when using a heuristic such as Nearest Neighbour (discussed below).

Below in Table 2 we can see the impact of each configuration on our solution

Problem File	Config	Mean Best Solution	Mean Runtime (s)
inst-4.tsp	1	15009105	35.8
	2	20568551	45.1
inst-6.tsp	1	229359954	319.6
	2	332374575	268.4
inst-16.tsp	1	59758756	67.2
	2	100842526	88

Table 2 – Config 1 vs Config 2 Comparison for all problem files

As we can see, Config 1 generates a far better mean best solution, with a shorter runtime in the case of the small and medium dataset and a larger runtime in the case of the large dataset.

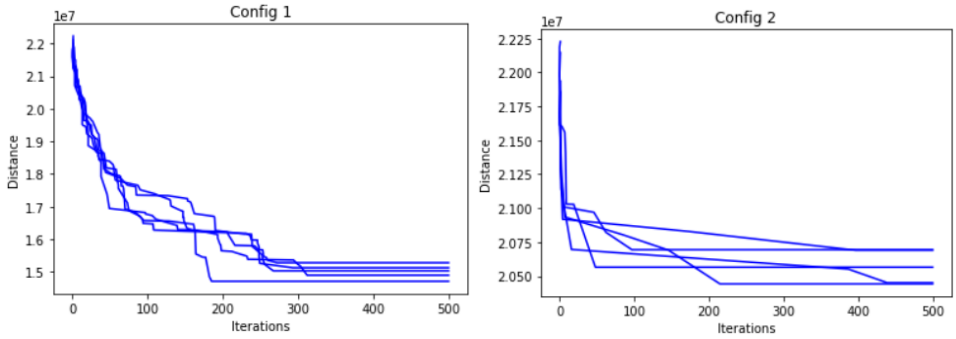

Figure 7 – Config 1 vs Config 2 comparison (inst-4.tsp)

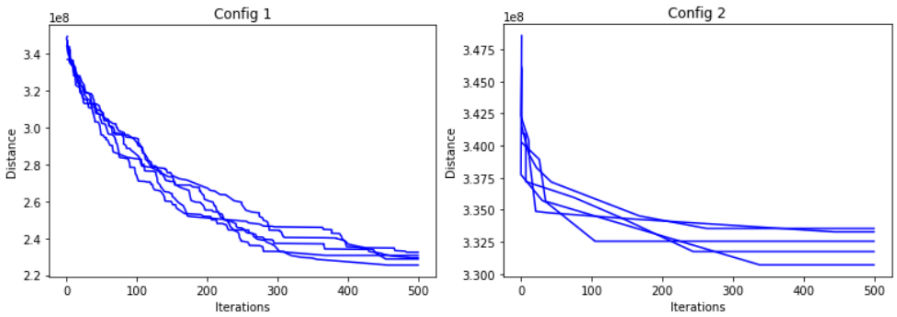

Figure 8 – Config 1 vs Config 2 Comparison (inst-6.tsp)

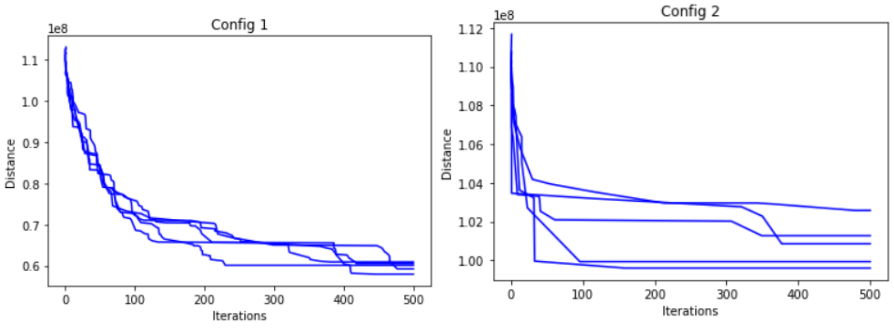

Figure 9 - Config 1 vs Config 2 Comparison (inst-16.tsp)

Looking at the graphs above, we can see the impact on the size of the dataset on the solution. Comparing config 1 across all datasets we can see that for inst-4 (small) our solution plateaus around the 300 iterations mark where for inst-6 (large) it has yet to plateau even after 500 iterations.

When we compare Config 1 to Config 2 across all datasets, we can see that Config 2 generates a far less optimal solution than Config 1

Evaluation of Genetic Algorithm

Below are the configurations used to evaluate the effect of certain crossover and mutation combinations on the overall results. We also explored the effect of implementing a heuristic to generate of initial sample.

Config	Initial Population	Crossover	Mutation
3	Random	Order-1	Scramble
4	Random	Uniform	Inversion
5	Nearest Neighbour	Order-1	Scramble
6	Nearest Neighbour	Uniform	Inversion

Table 3 – Configurations 3, 4, 5 and 6

We ran these configurations on each of our datasets 5 times to generate the mean best solution and mean runtime. We used the same populations size, mutation rate and max iterations as specified in the previous section for our experiments here. Below are our results,

Problem File	Config	Mean Best Solution	Mean Runtime (s)
inst-4.tsp	3	15149516	36
	4	20362857	45.2
	5	15050734	39.9
	6	20595998	49.1
inst-6.tsp	3	235825033	317.9
	4	332353765	465.7
	5	238831478	393.9
	6	332518147	534.2
inst-16.tsp	3	61723979	67.5
	4	101114531	88
	5	63404137	77.1
	6	100659437	97

Table 4 – Config 3,4,5 and 6 comparison

For each dataset, configuration 5 and 6 takes longer to execute than configuration 3 and 4 respectively. The only difference between configurations 5 and 6 and configurations 3 and 4 is that 5 and 6 uses or nearest neighbour heuristic (discussed in a previous section) to determine its initial population. This leads to a better starting mean best starting position. We can see from the results that for the largest dataset this has very little effect. Below we see the impact on the initial best solution.

Population Size	File	Random Best Initial	Nearest Neighbour Best Initial
100	inst-4.tsp	22227402	21925889
500		21841064	21058637

Figure 10 – Impact of heuristic on best initial solution

As we can see implementing a heuristic can lead to a better initial solution. However, this is no guarantee that we will reach our optimal solution faster.

I've attached graphs and tables for each dataset/configuration results in the appendix.

Population Size

Here we'll look at the impact altering the population size has on our mean best solution and mean completion time. We used inst-4.tsp to carry out these experiments, with a mutation rate of 0.05, with 500 iterations over 5 runs.

Population Size	File	Config	Mean Completion Time	Mean Best Solution
300	inst-4	3	72	16003468
		4	90	20200202
		5	80	16048691
		6	98	20252184
100		3	36	15149516
		4	45	20362857
		5	40	15050734
		6	49	20595998
50		3	18	14928030
		4	22	20706959
		5	20	15205044
		6	25	20477999

Table 5 – Impact on altering population size

As we can see an increase in population size causes an increase in completion time. This is because the extra processing required to create, crossover, mutate and evaluate each individual. An increase in population size doesn't seem to have a large impact on our mean best solution for the small sized dataset (inst-4.tsp).

Mutation Rate

Below we can see the effect of altering the mutation rate on our solutions. As before, each run uses a mutation rate of 0.05, with 500 iterations over 5 runs.

Mutation Rate	File	Config	Mean Completion Time	Mean Best Solution
0.5	inst-4	3	37	18716546
		4	45	20437150
		5	40	18964602
		6	49	20562498
0.1		3	36	15929074
		4	45	20471867
		5	40	15430312
		6	49	20296064
0.05		3	36	15149516
		4	45	20362857
		5	40	15050734
		6	49	20595998
0.05		3	36	14880677
		4	44	20478268
		5	40	14745571
		6	49	20505792

Figure 11 – Impact on altering mutation rate

We can see that an increase in mutation rate can lead to a drop in performance. This is because mutating 'good' individuals can lead to a loss in performance and this leads to more of a random solution. As mentioned in a previous section a good rule of thumb is to design your GA in such a way that approximately 1 individual is mutated on each iteration.

Inversion Change probability

Next, we investigated the effect of changing the probability of changing a chromosome in our Uniform crossover. We can see that a higher probability of changing a chromosome can lead to better results.

Population Size	Uniform Change Prob	File	Config	Completion Time	Best Solution
100	0.3	inst-4	2	50	20465669
	0.5		2	45.1	20568551
	0.7		2	48	20274191

Table 6 – Impact of Inversion Change Probability

Conclusions

From our results we can see that, for our datasets, 'Order-1' crossover generates a more optimal solution than 'Uniform' crossover.

We also saw that increasing the mutation rate can lead to poorer performance in our algorithm.

We found that increasing the population does not always lead to an increase in performance. For smaller datasets an increase in population has little effect, while in larger datasets it can help lead to an optimal solution quicker.

Appendix

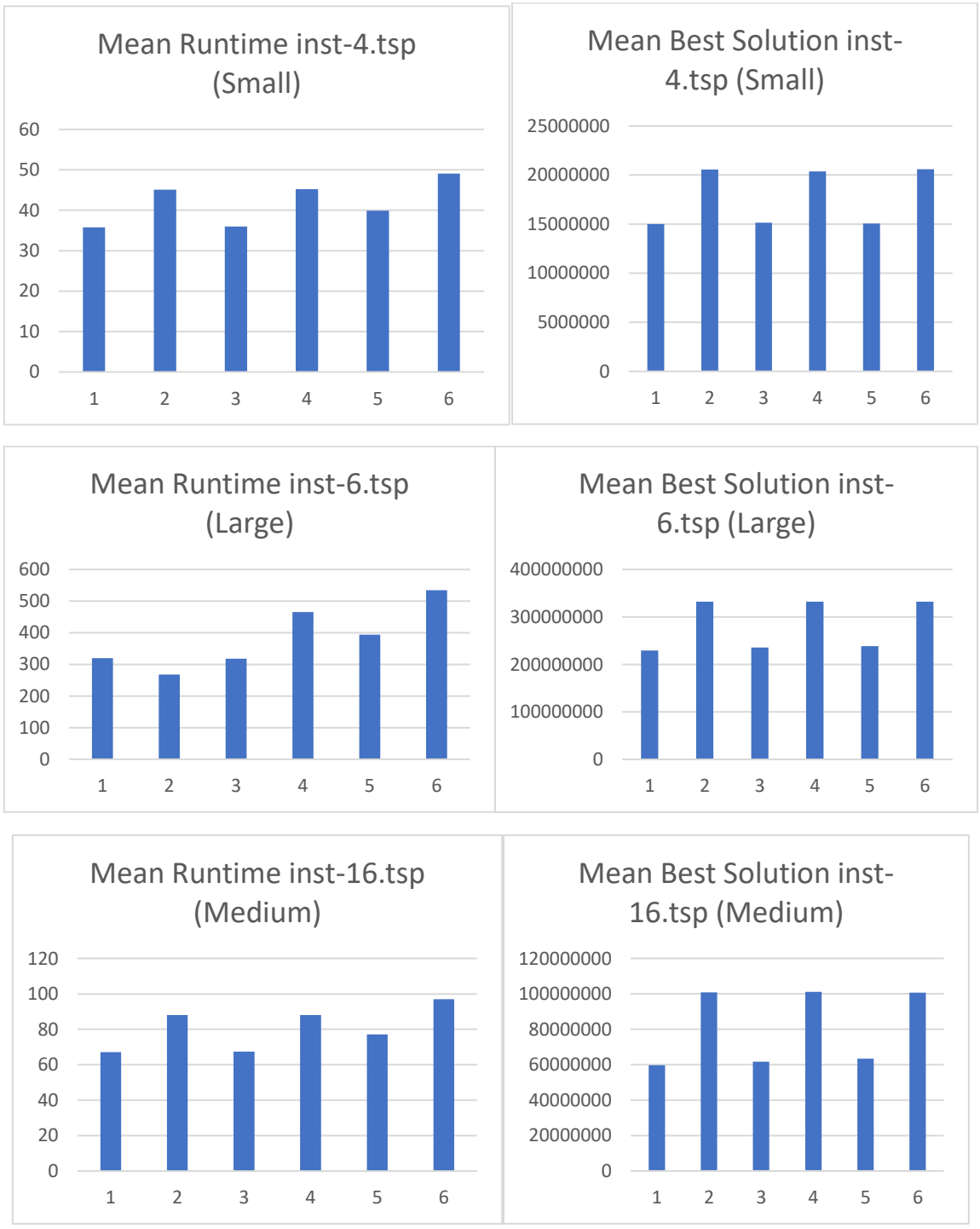

Figure 12 – Results Graphs

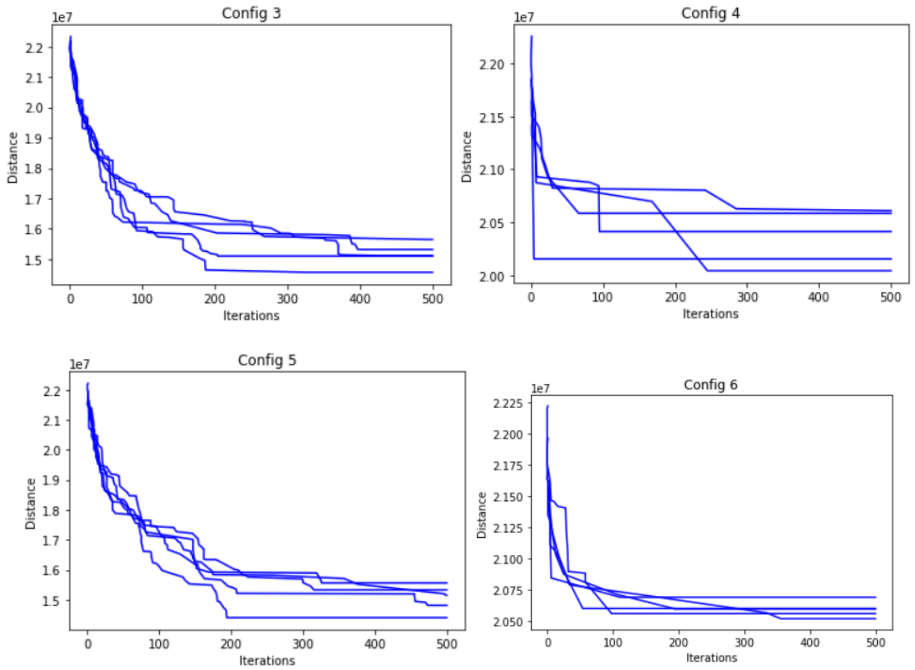

Figure 13 – inst-4.tsp Configuration Comparisons (Best Fitness/Iterations)

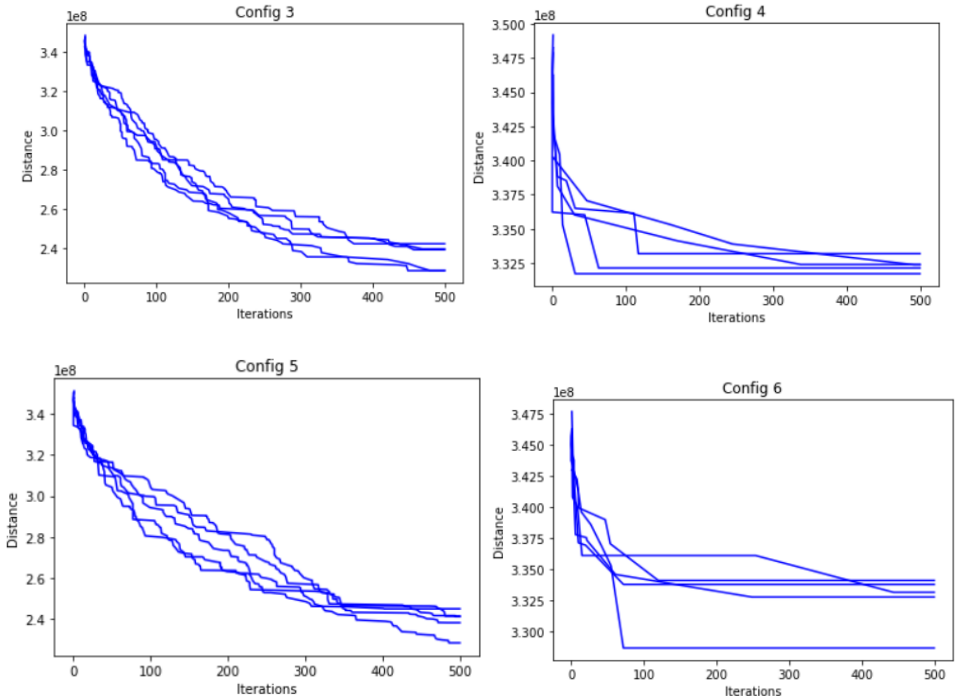

Figure 14 - inst-6.tsp Configuration Comparisons (Best Fitness/Iterations)

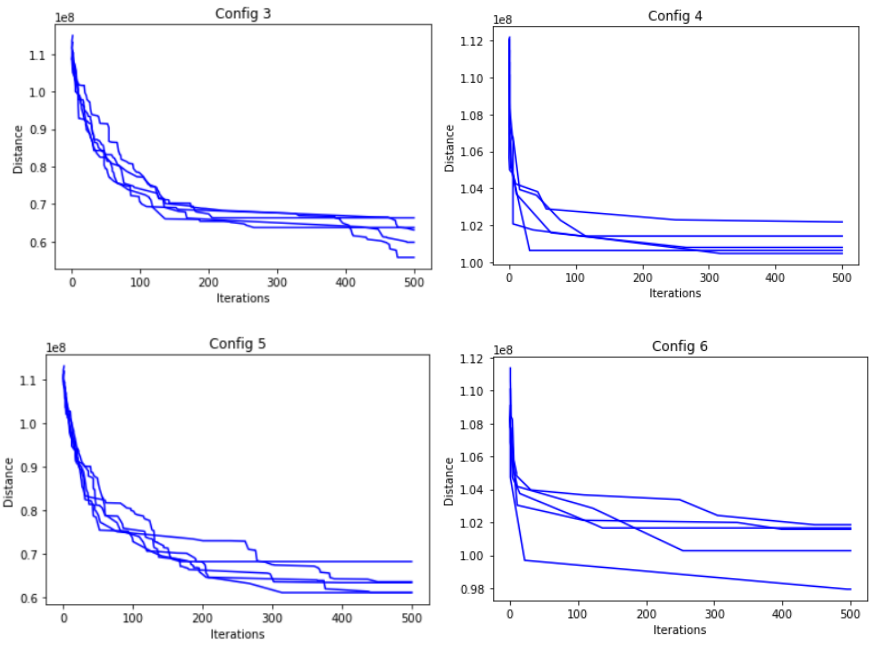

Figure 15 - inst-16.tsp Configuration Comparisons (Best Fitness/Iterations)

References

[1] Dr. D. Grimes, *Assignment 1 Specification.*

[2] Y. S. M. M. N. S. Abid Hussain, “Performance Evaluation of Best-Worst Selection Criteria for Genetic Algorithm,” Science Publishing Group , Islamabad, Pakistan, 2017.