Genetic Algorithm for Travelling Salesman Problem

Report

R00195734

Contents

Part 1: NP Completeness

$$F = (-\omega_3 \vee \omega_4) \wedge (\omega_1 \vee -\omega_2 \vee \omega_3 \vee -\omega_4 \vee -\omega_5 \vee \omega_6)$$

$F' =$

1. $(-\omega_3 \vee \omega_4 \vee y_1)$
2. $\wedge\ (-\omega_3 \vee \omega_4 \vee -y_1)$
3. $\wedge\ (\omega_1 \vee -\omega_2 \vee y_2)$
4. $\wedge\ (-y_2 \vee \omega_3 \vee y_3)$
5. $\wedge\ (-y_3 \vee -\omega_4 \vee y_4)$
6. $\wedge\ (-y_4 \vee -\omega_5 \vee \omega_6)$.

One Solution for F':-

$\omega_1 = F$	$y_1 = T$
$\omega_2 = T$	$y_2 = T$
$\omega_3 = F$	$y_3 = T$
$\omega_4 = T$	$y_4 = T$.
$\omega_5 = T$	
$\omega_6 = T$.	

Clause 1 of F is satisfied because $w_4 = T$

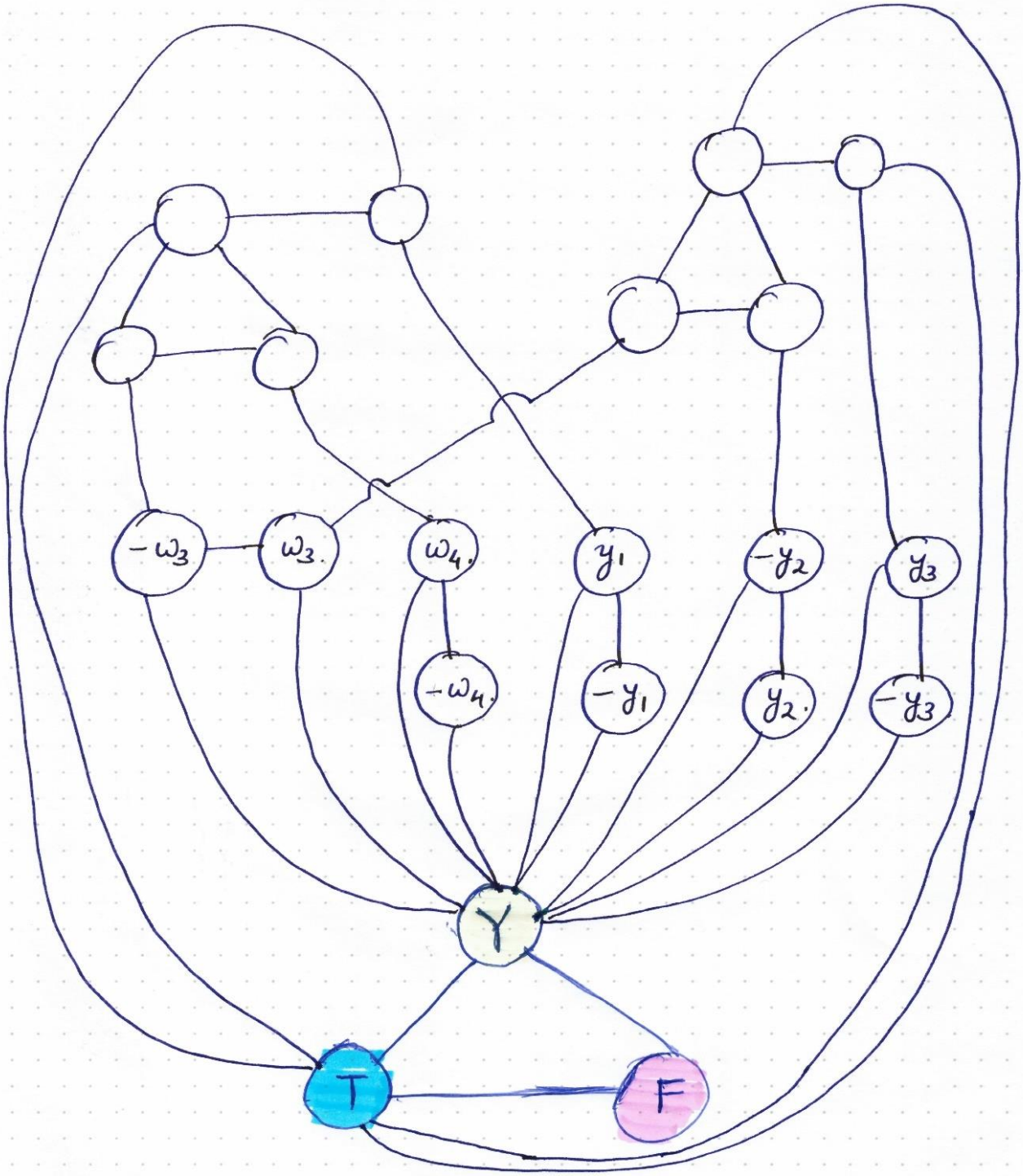
Clause 2 of F is satisfied because $\omega_6 = T$

Therefore this solution for F' satisfies F

3SAT → 3COL

① $(-w_3 \vee w_4 \vee y_1)$

② $(-y_2 \vee w_3 \vee y_3)$



Part 2

A genetic algorithm was created based on the skeleton code provided by Dr. Grimes. Tests were done with configurations 1-6.

The insertion heuristic used was the solution to the code provided by Dr. Diarmuid Grimes.

A multi-processing approach was used to run different runs in parallel. The default configuration is to do 7 runs in parallel. This can be changed by changing the global variable *g_n_processes*.

All graphs plotted were saved as *cpickle* in files. These graphs can be re-rendered using the script plotagain.py.

Genetic Algorithms draw inspiration from evolution and are used to solve optimization problems. They are randomized algorithms. An initial population is chosen. A fitness function is applied, and then individuals from the initial population are mated or crossed over, and mutations applied. This forms the next generation. The older generation is replaced completely or partially in the population with the new generation. The steps are repeated

1. **Select Initial Population**
2. **Repeat**
    a. **Calculate fitness**
    b. **Mate**
    c. **Mutate**
    d. **Select Survivors**

Genetic Algorithms can get stuck on local optimas, and to mitigate them, random restart is used. In our runs, 5 random restarts were used.

Note on debug statements and performance

For debugging, several asserts were added, and these asserts slow down the code by a factor between 2x and 10x. To turn these asserts off, the code can be run with the python option -OO, which turns of the __debug__ built-in variable.

Also, when profiled, the most expensive operation was calculation of Euclidan distance. Since the same distances are calculated multiple times, an optimization would be to use a hash-lookup instead of calculating it each time. This would come at the expense of increased memory usage, but keeping in mind the time taken for the runs this may be well worth it. This was not implemented, but would be an area of further development.

I also tried using manhattan distance instead of Euclidean distance as it is computationally less intensive. However, the results obtained with manhattan distances were not very good and the approach was abandoned early on.

Design of Experiments

The objective of the experiment was to evaluate the performance of different parameters of the GA and their suitability to the traveling salesman problem. A GA was coded, and run for 1000 to 1500 executions, and 5 runs for each city. The parameters (mutation rate, and population size) were varied independently of each (keeping the other fixed), and compared. Each parameter was compared with each of the configurations.

Furthermore, within each run, the mean, median and best fitness in each iteration was plotted. This was done to give a sense of how quickly GA improves (or makes worse, as we saw in some configurations) with iterations, and how that changes over a period of time.

The results were mostly consistent with theory. Initially the GA was able to improve the solution very quickly, but as iterations went on, it became more difficult to find better solutions. The trend was the same where GA found worse solutions with time. Initially, it made the solution worse very quickly, but later it stabilized.

Three files were used for the experiment – these three files had a different number of cities, and also had different sequences.

Implementation Details

Selection and population

Binary tournament selection was implemented.

Binary Tournament Selection is performed before cross over and mating. Each time, two random individuals are picked from the pool, and their fitness compared. The winner is chosen for crossover.

Two methods for population replacement were used.

1. **Complete replacement**
2. **A variant of Elitist replacement (This can be specified by specifying the ratio of parents with the option -epr)**

Most of the detailed comparison was done using complete replacement. However, when the initial population was arrived at using insertion heuristic, a complete replacement didn't perform very well.

Elitist replacement performed much better when insertion heuristic was used. However, due to time constraints, only basic comparisons were made with Elitist replacement. Future work would involve varying the other parameters with Elitist replacement.

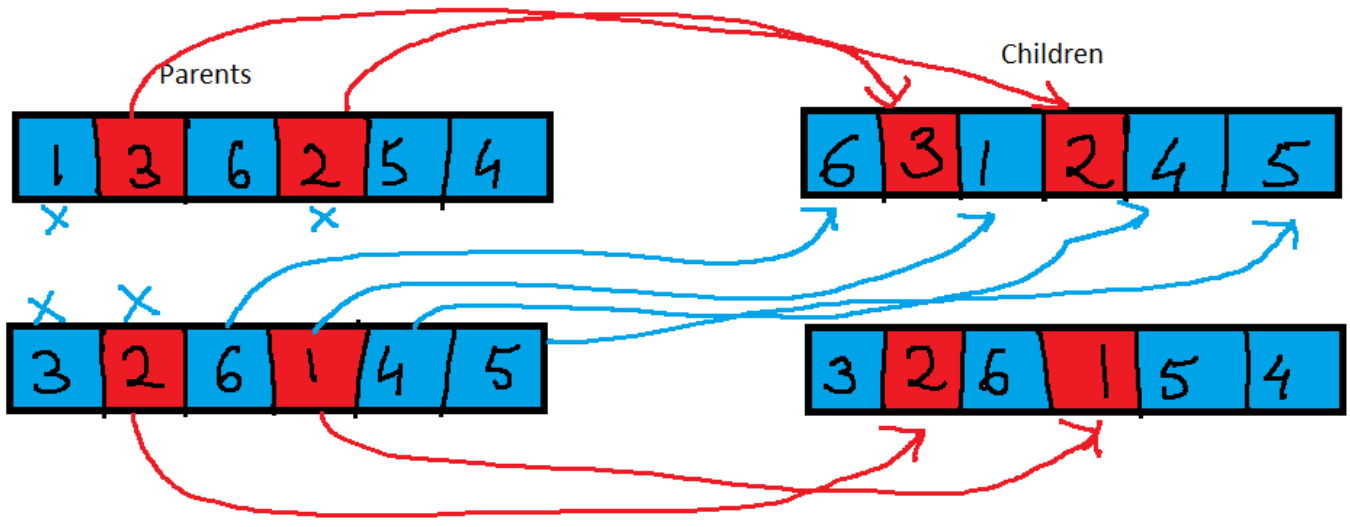
A heap was used to select k parents with the lowest distances. Since heapify runs in linear time, the overall complexity of this operation is O(n + k log(n)).

Crossover

Uniform Crossover

In Uniform Crossover, two parents are crossed and they can produce either one or two children, based on the variant. A set of locations are chosen randomly and they are fixed position in both the parents. These positions are not necessarily contiguous.

The child corresponding to each parent will have the same genes at the same fixed locations. The remaining genes will be copied from the other parent in order.



Uniform Crossover

Two variants of uniform crossover were implemented:

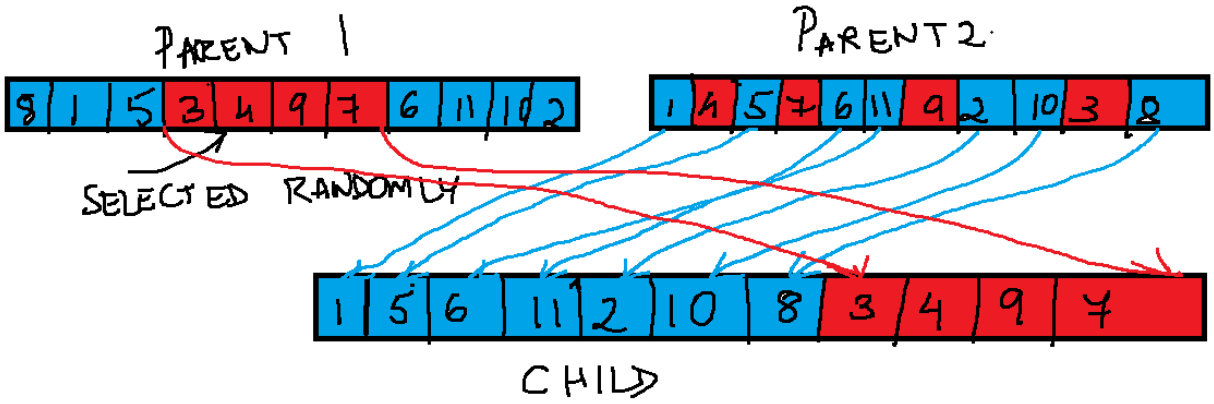
1. **Random selection of number of elements to splice, with a low value set to 5**
2. **Number of elements to splice is between 50 and 75% of the number of genes**

No significant difference was observed between the two variants. Further work would include running comparisons with the second variant as well.

Furthermore, a frozen set was used to avoid an O(n^2) cost, to keep the cost down to O(n).

Order 1 Crossover

Order-1 cross-over looks at 2 individuals, and produces one child. A set of consecutive genes are chosen at random from parent 1. The child will inherit these genes in the same order as the parent 1. All other genes not in this set are first chosen from parent 2, and added to the child. Following this, this set of selected genes is appended to the child.



Order 1 Crossover

Two variants of Order-1 crossover were implemented:

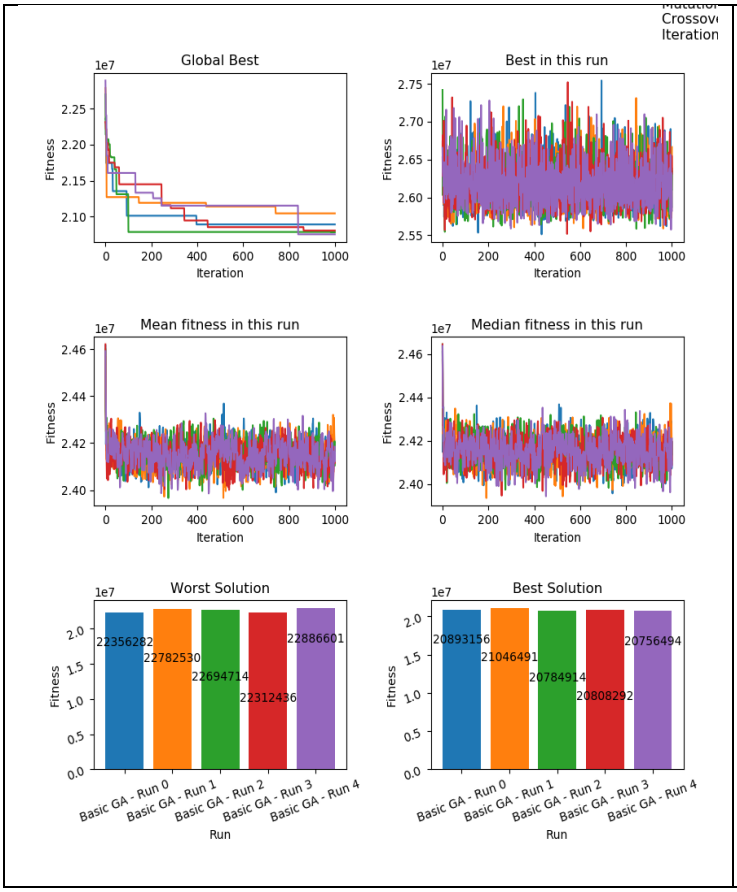
1. **That selects a random number of fixed genes**
2. **That selects a random number of fixed genes with a minimum of 50% of total genes, and a maximum of 75% of total genes.**

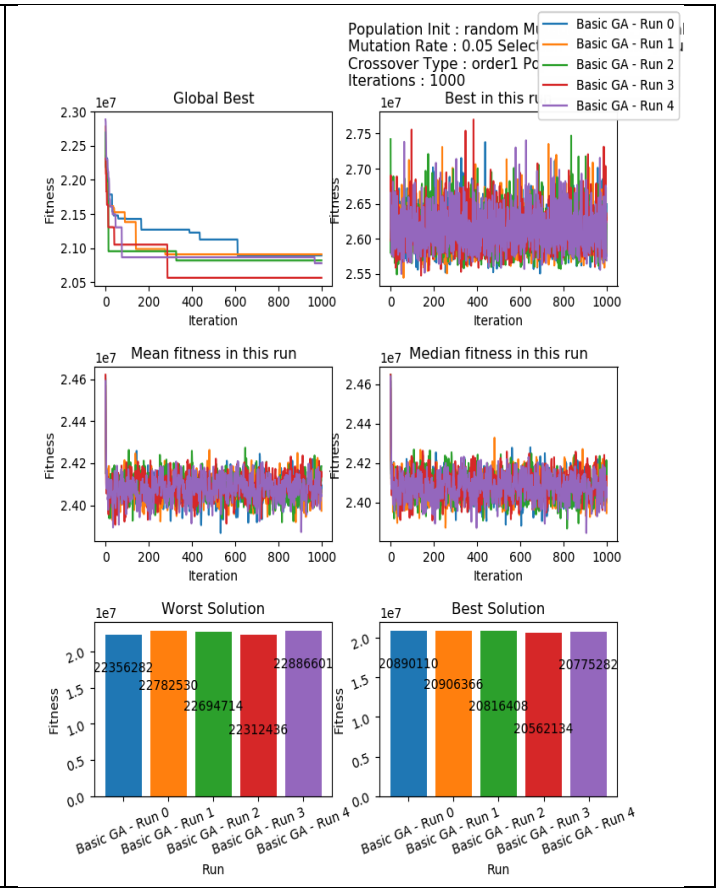
The second variant was coded later as an experiment when the first variant did not perform very well when using an insertion heuristic. Most of the comparisons, however, were run with the first variant. Further work would include running comparisons with the second variant as well.

A third variant of order-1 crossover was also implemented, in which two parents are crossed-over to form two children, with the same fixed-indices. However results are not included for the third variation.

No significant difference was found between the two variants. The unconstrained number of genes performed slightly better (the solution was better by 0.93%).

Between 50 and 75% of genes	Unconstrained Random Number of Genes

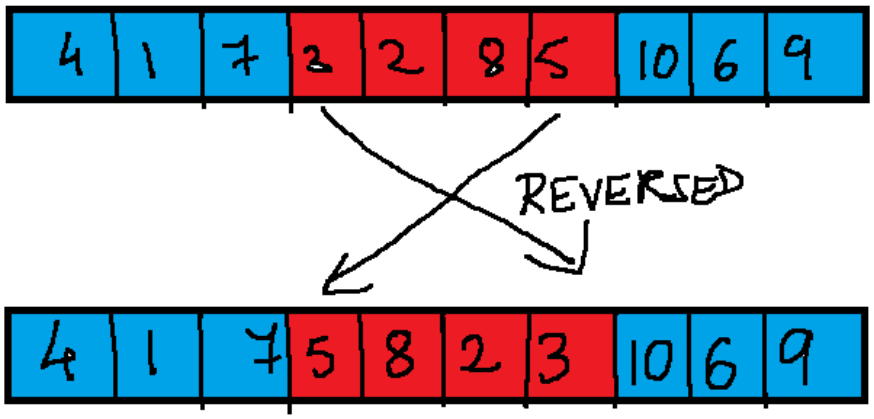




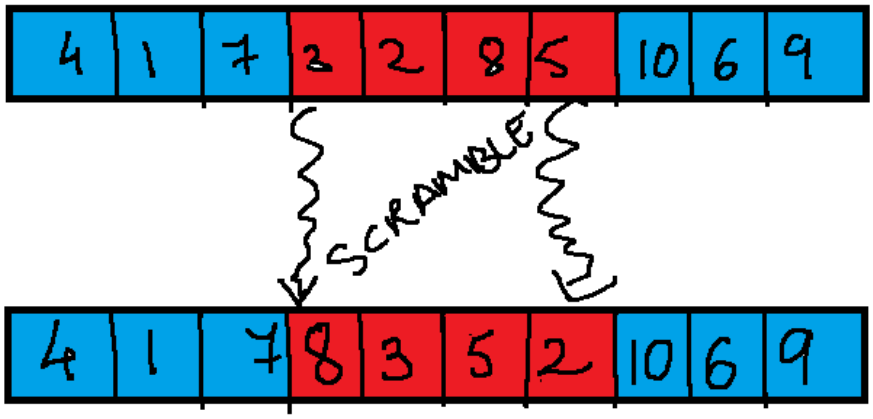
Mutation

Scramble and inversion mutation were implemented

In inversion mutation, a random set of consecutive locations are chosen and those are reversed. The example below can help explain that.



Scramble mutation is more destructive, and it scrambles all genes with in the selected range.



Whether or not a mutation happens at all or not is controlled by a Mutation-Rate. A random number is chosen, and if it is less than the mutation rate, mutation is performed. Ideally the mutation rate should be low.

A variant of mutation was also tried where at least 50% of the genes were mutated. However, that, also couldn't break the bad solutions the GA got stuck on when the initial population was selected with insertion heuristic. Results for this have not been presented.

Creating the Mating Pool

Two versions were implemented:

1. **A version that copies all individuals from the population to form the mating pool**
2. **A version that selects individuals based on a probability distribution that varies with the inverse of the total distance. However, all comparisons were performed with the former variant, because binary tournament selection also applies a probability and chooses better solutions with a higher probability, and if the second version was used, it would result in applying the probability twice. This version has not been included in the results.**

Results

In the first are the results of the basic runs. In this section, I only discuss each configuration individually, and the characteristics of each configuration between each iteration.

In the second section, I will present a comparison of the various runs in the same place.

In the third section, I will present the effect of varying the different parameters (mutation rate and population size).

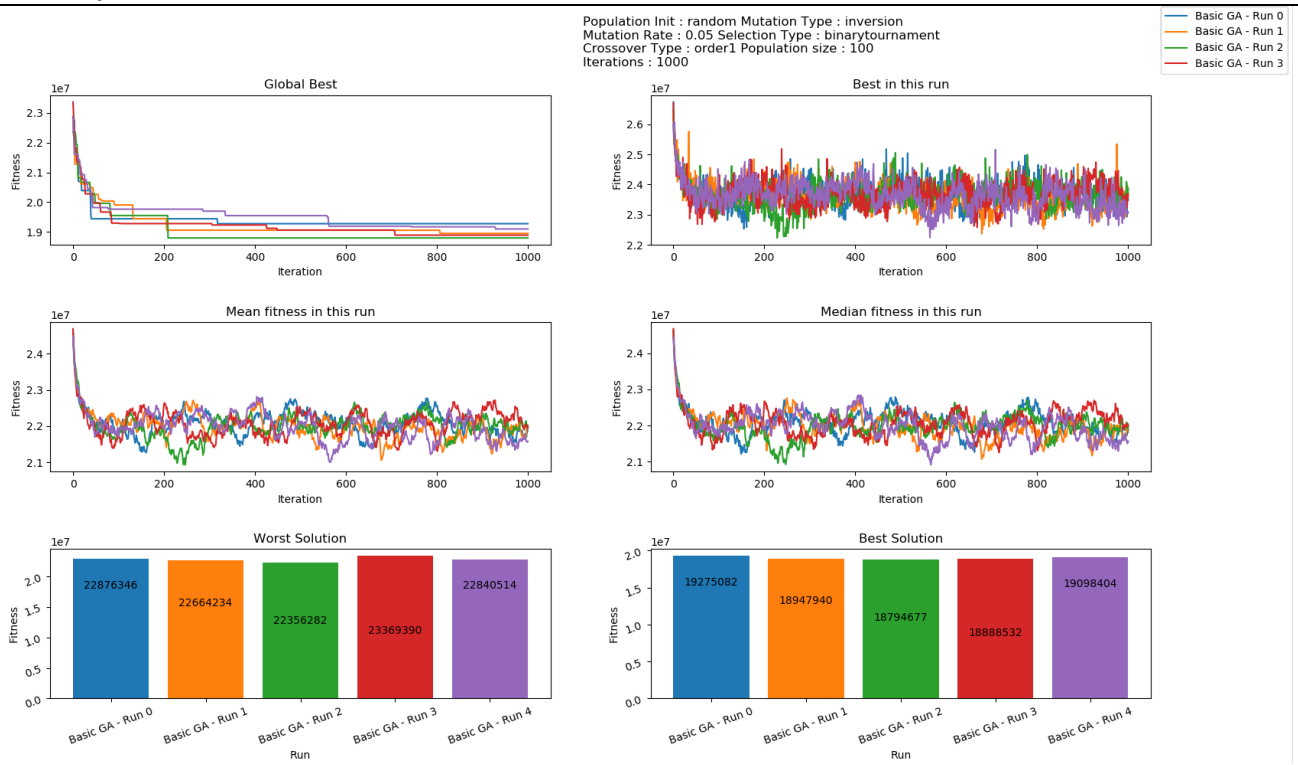
In the last section, I will present the results from the additional experiments with elitism and variants of different crossovers. This last section has been performed with fewer number of runs due to time constraints.

Basic Runs

Configuration 1

Initial Population: Random
Crossover: Order-1
Mutation: Inverse
Selection: Binary Tournament

File: inst-0.tsp, 184 Cities

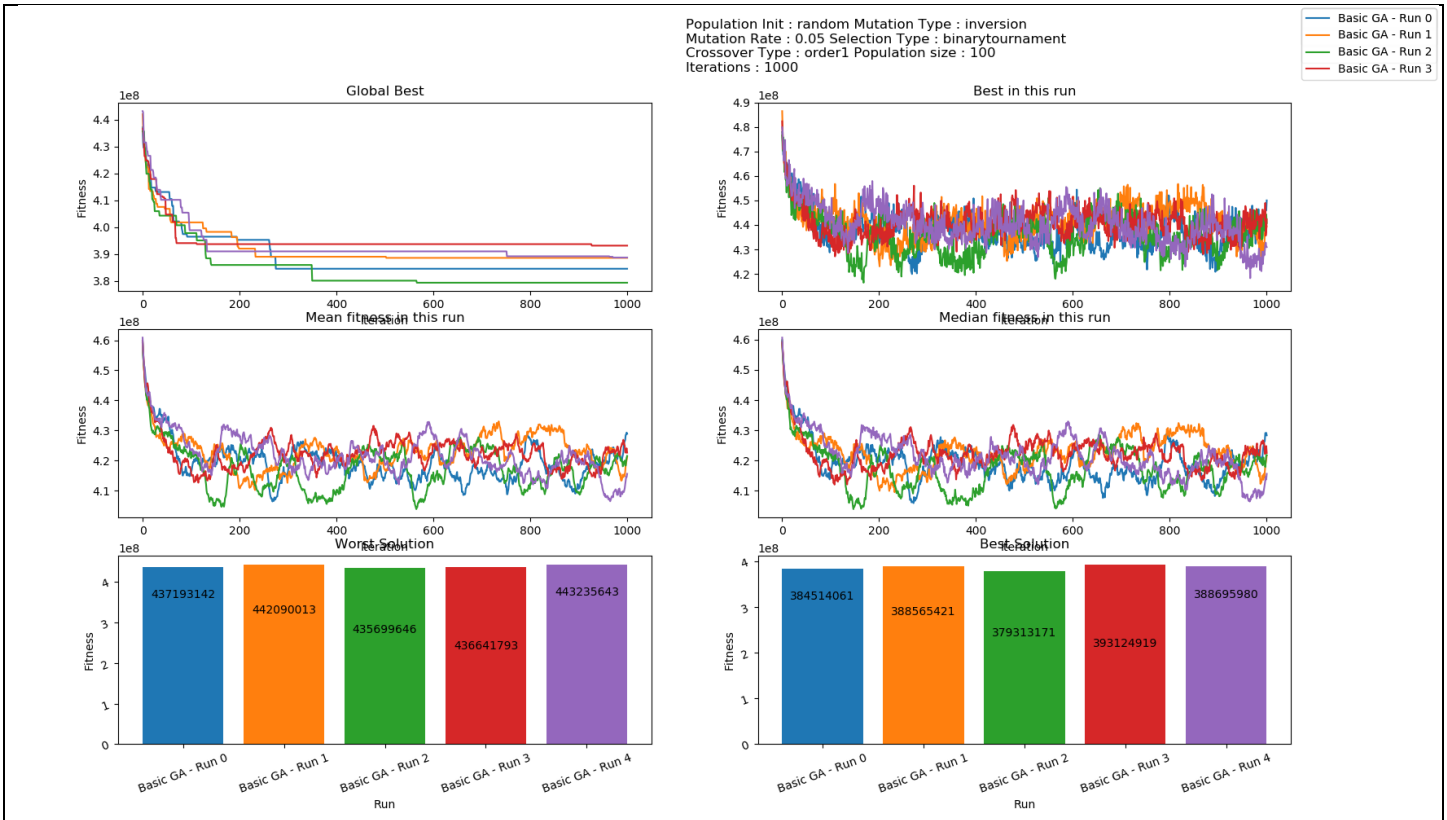


The legend is incomplete, there are 5 runs, but the legend only shows four.

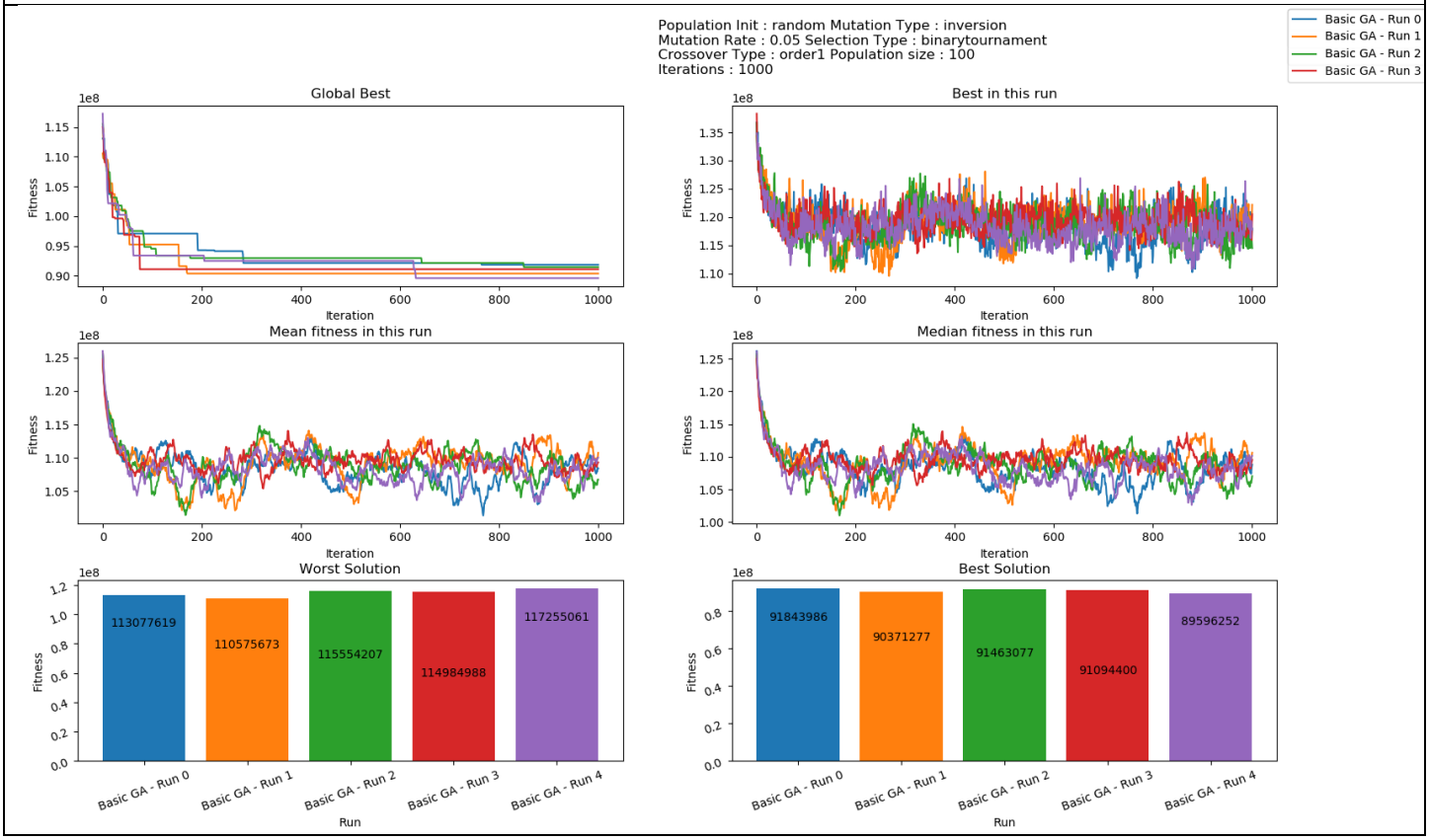
The above chart shows the fitness in every run, and the global best fitness. The shows a sharp decrease till it flattens out after around 100 iterations. This is consistent with theory as GA's tend to improve the solution very quickly in the beginning, till flattening out, or becoming very slow to find any improvements.

Similar trends were also observed with files inst-5 and inst 13, but the solution flattened after a different number of iterations.

Inst-5.tsp, 819 cities



Inst-13.tsp, 352 cities



Configuration 2

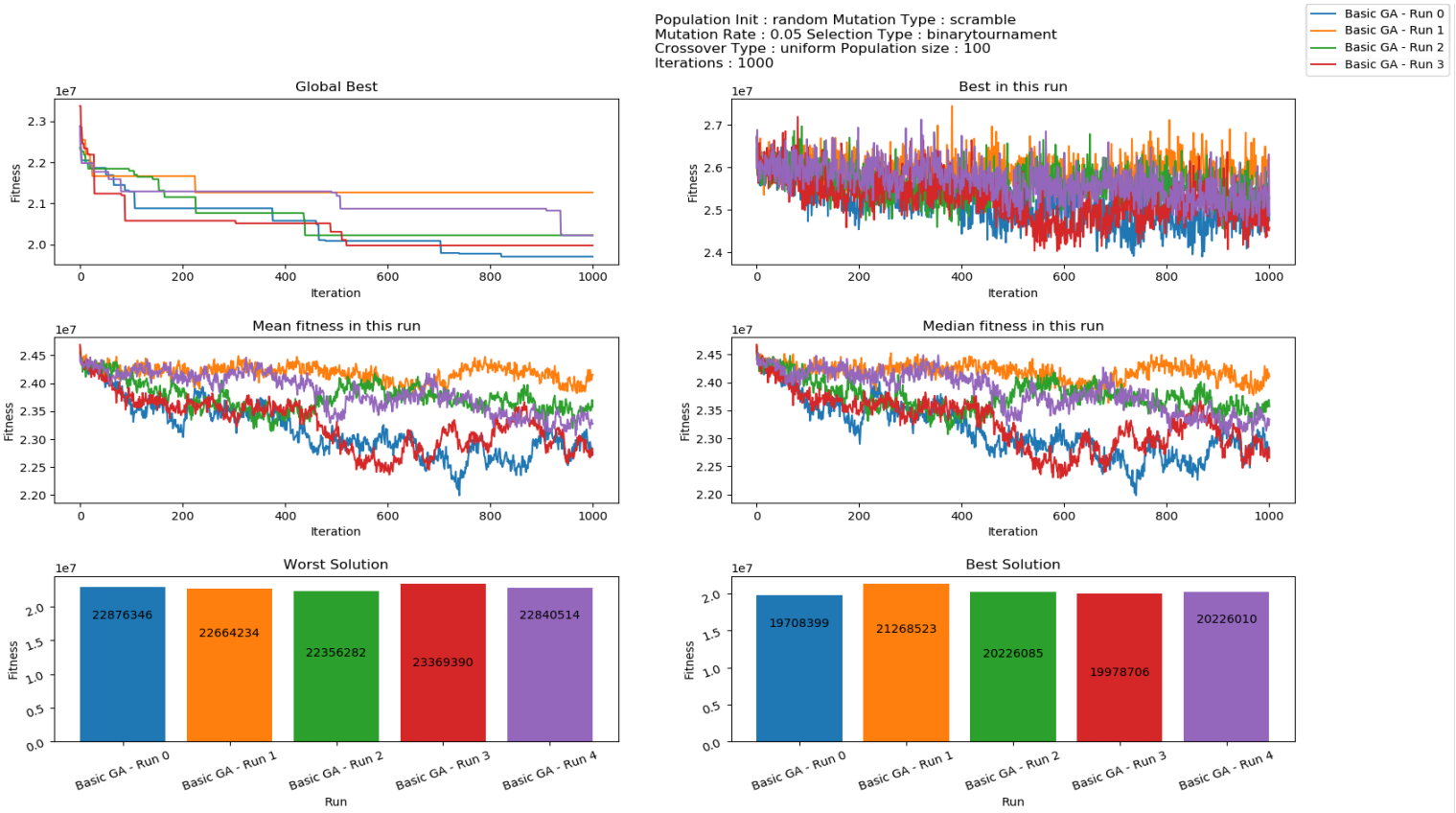
Scramble Mutation, Binary Tournament Selection, Random Initial Population Selection, Uniform Crossover.

Again, as before, a sharp fall in fitness was seen in the beginning till the graph levelled off. This is consistent with theory.

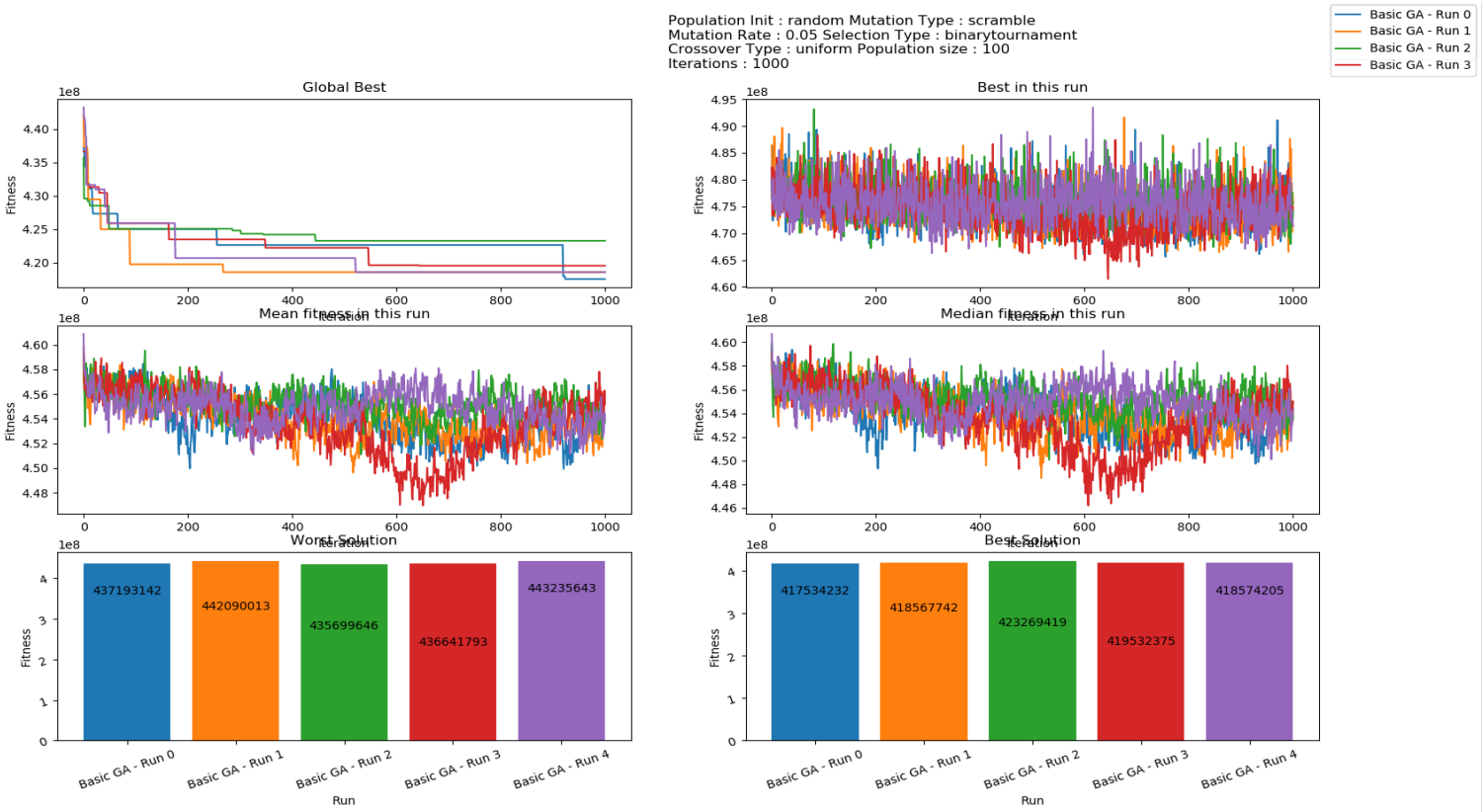
However, at the phase of the graph that levelled off, the median fitness in each run showed more variability than configuration 1. This effect could be due to the scramble mutation, which tends to produce more variance in the paths.

The graph for inst-13.tsp was somewhat different than the other two. In the beginning the median variance in each run was quite flat, after which it dropped and kept dropping consistently.

Inst-0.tsp, 184 cities

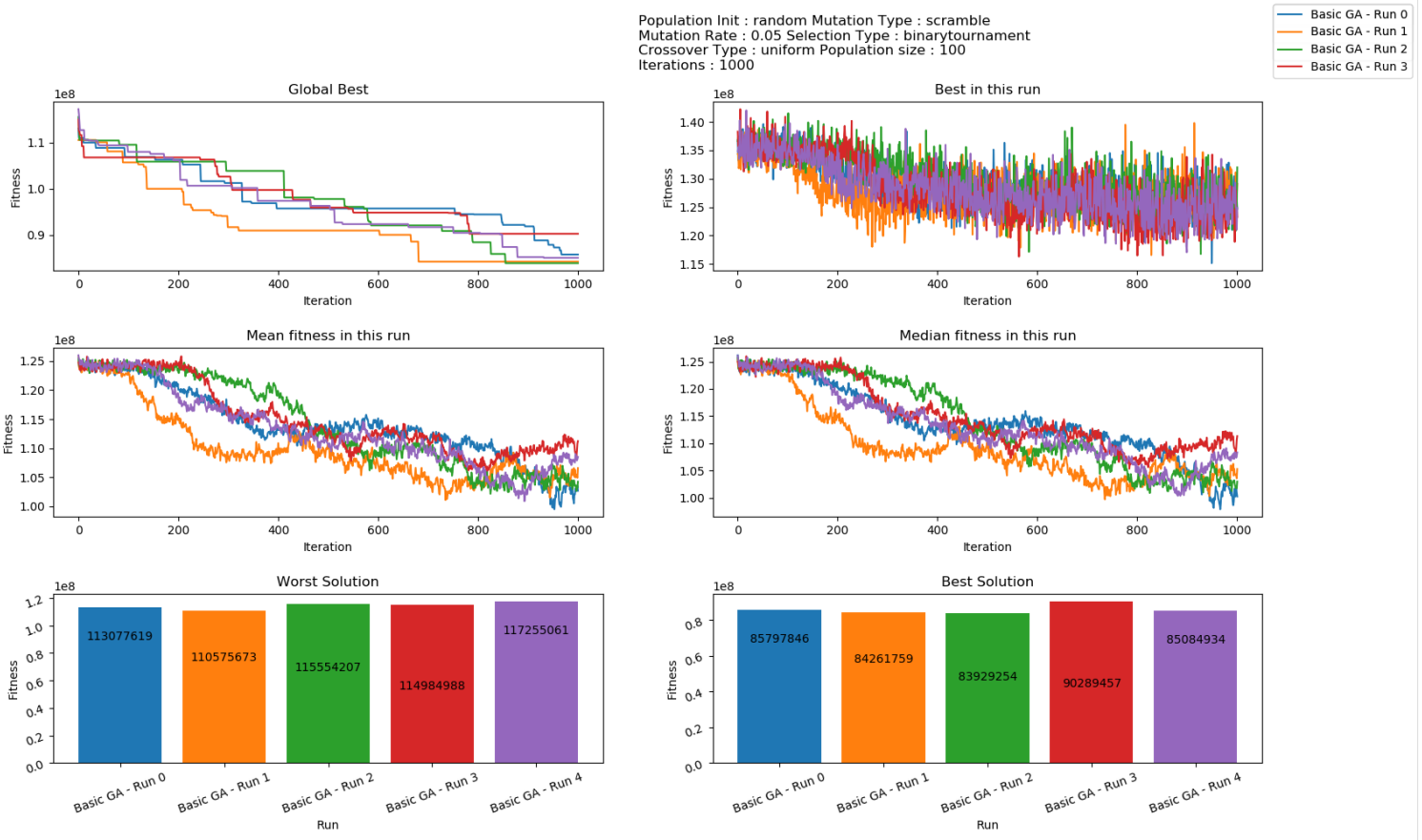


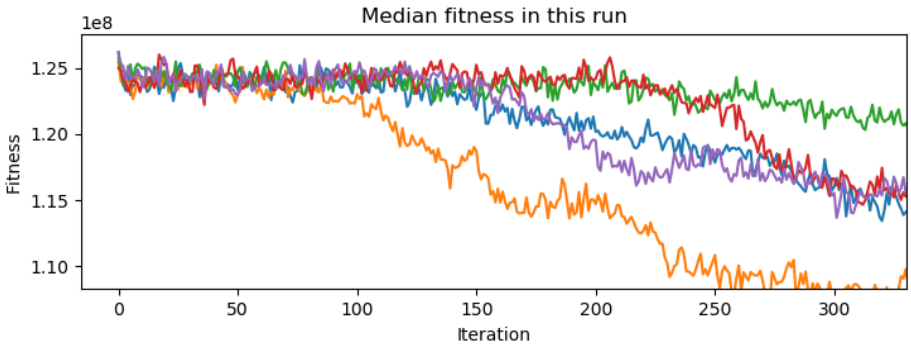
Inst-5.tsp, 819 cities



Inst-13.tsp 352 cities

Again, as noted earlier, this is somewhat different as the GA was initially unable to reduce the median fitness for the first 100 iterations, after which it kept dropping.



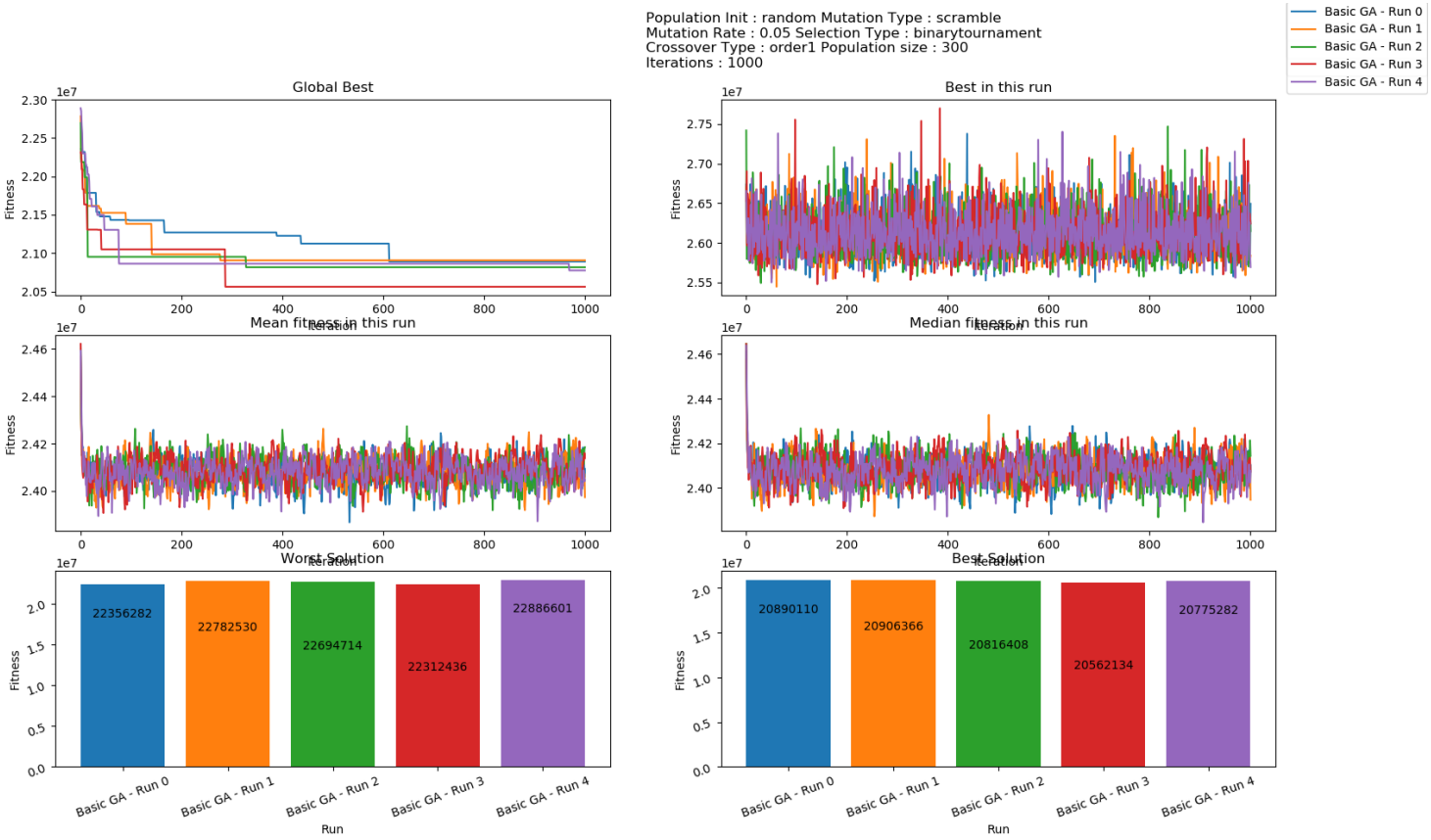


Configuration 3

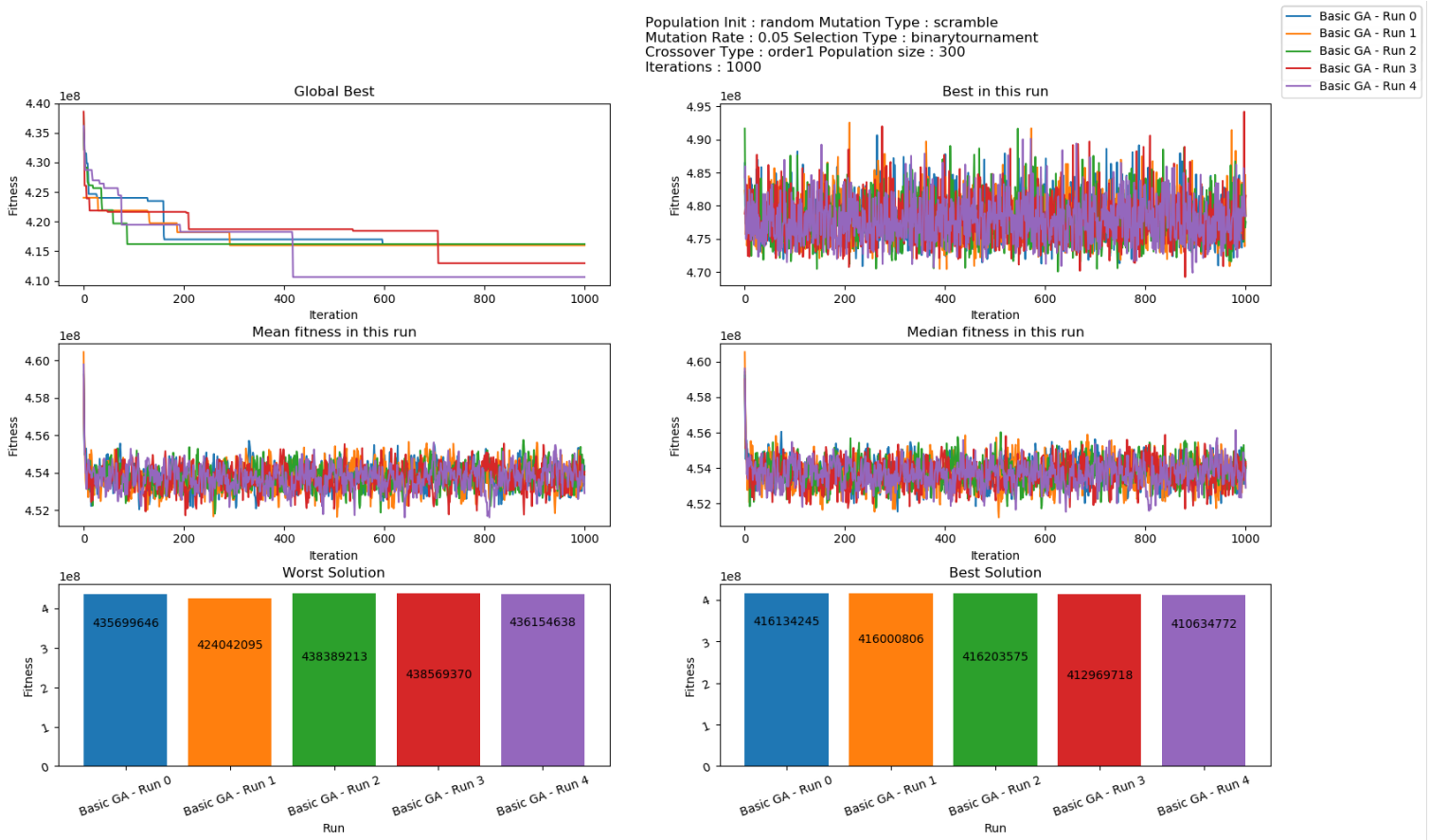
Random Initial Selection, Order-1 Crossover, Scramble Mutation, Binary Tournament Selection

The results in this run was also similar to configuration 2 in terms of trends. However the best fitness in every run showed much more variance than the previous configurations.

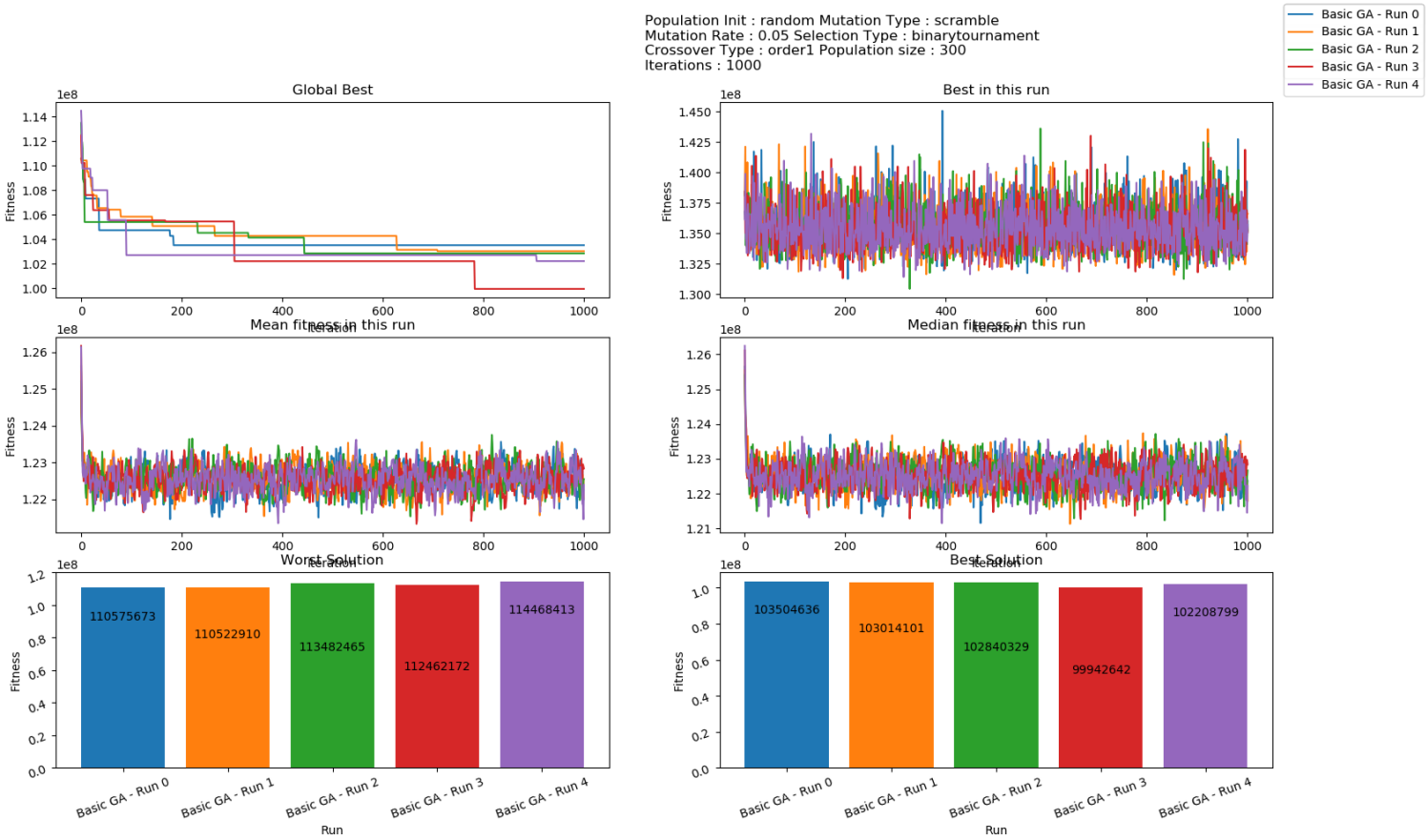
Inst-0.tsp, 184 cities



Isn't-5.tsp


Population Init : random Mutation Type : scramble
Mutation Rate : 0.05 Selection Type : binarytournament
Crossover Type : order1 Population size : 300
Iterations : 1000
Basic GA - Run 0
Basic GA - Run 1
Basic GA - Run 2
Basic GA - Run 3
Basic GA - Run 4
Global Best
Best in this run
Mean fitness in this run
Median fitness in this run
Worst Solution
Best Solution
Fitness
Iteration
Run
1e8
435699646
424042095
438389213
438569370
436154638
416134245
416000806
416203575
412969718
410634772

Inst-13.tsp

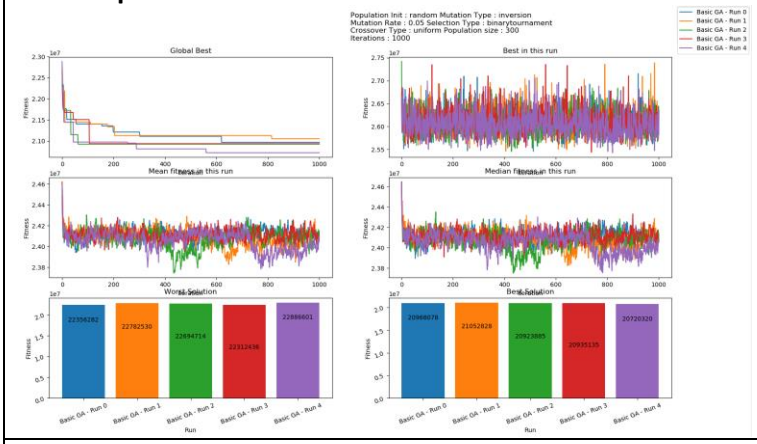
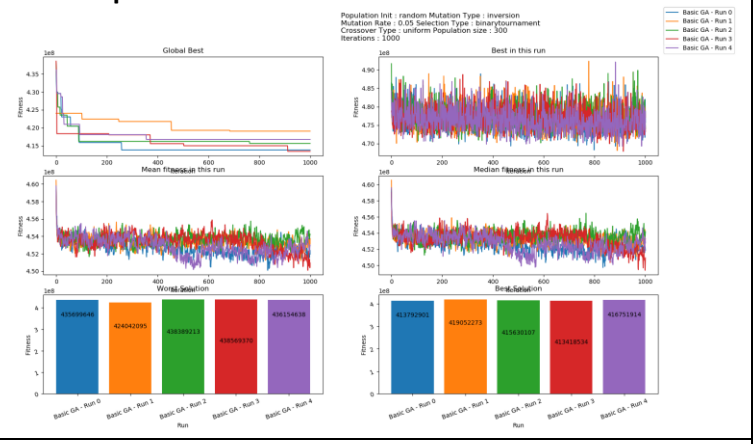


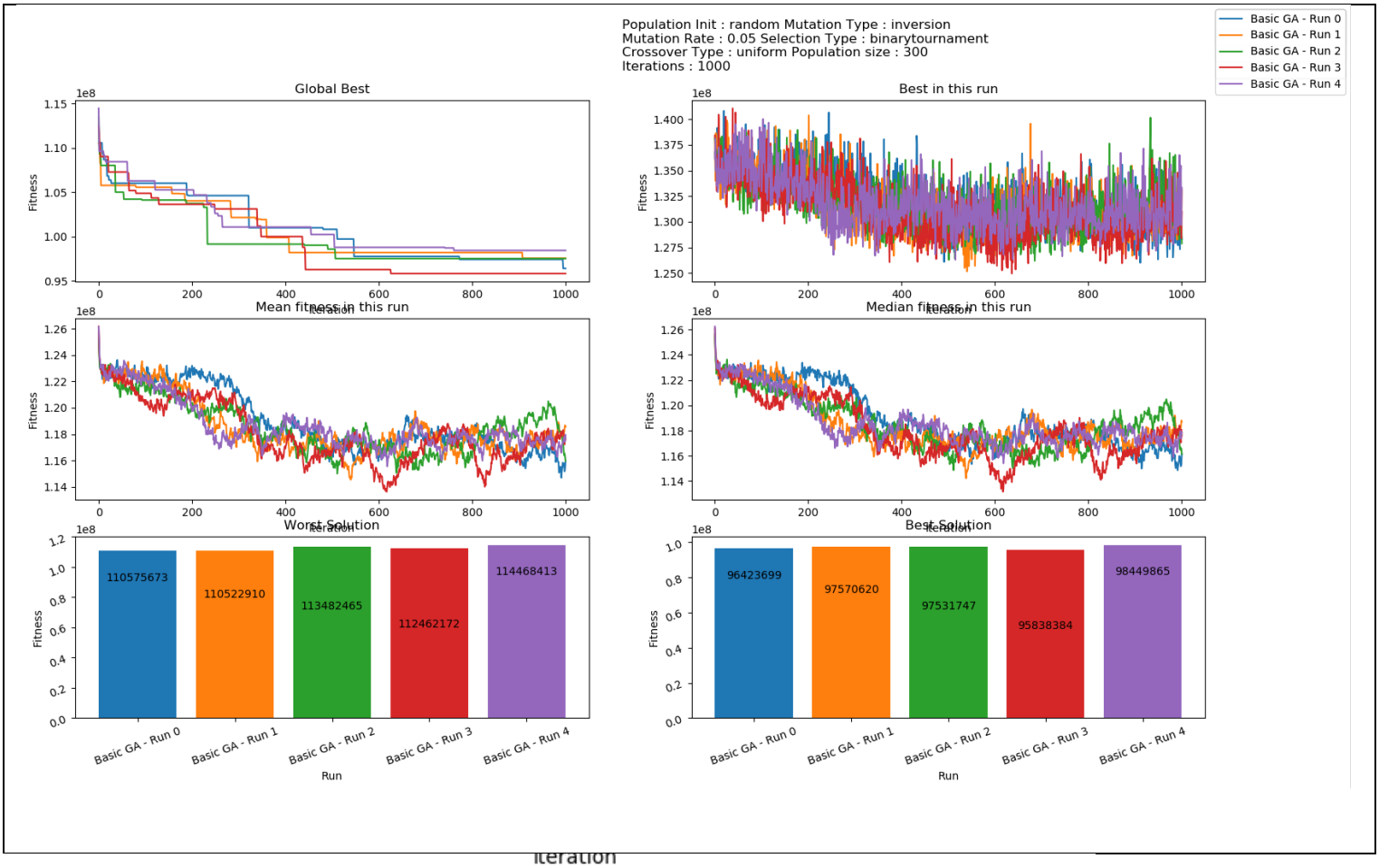
Configuration 4

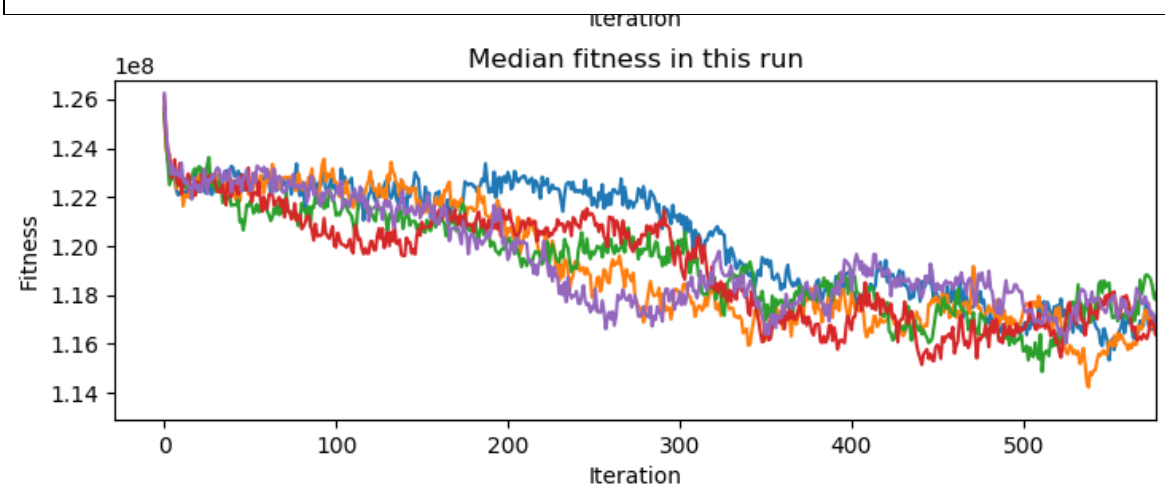
Random Population Selection, Inversion Mutation, Uniform Crossover, Binary Tournament

Again, this configuration had similar characteristics to configuration 4. The graphs are presented below, but there is nothing noteworthy here, hence the graphs have been made smaller in size.

inst-13.tsp showed a trend similar to configuration2. Initially the GA was not able to find good solutions for some time, and then it kept dropping continuously and consistently.

Inst-0.tsp	**Inst-5.tsp**
	
Inst-13.tsp	





The above graph shows a shar drop in the first few iterations, then a flattening till iteration100, then a gradual drop between iteration 150 and iteration 350, and then a flattening.

Configuration 5

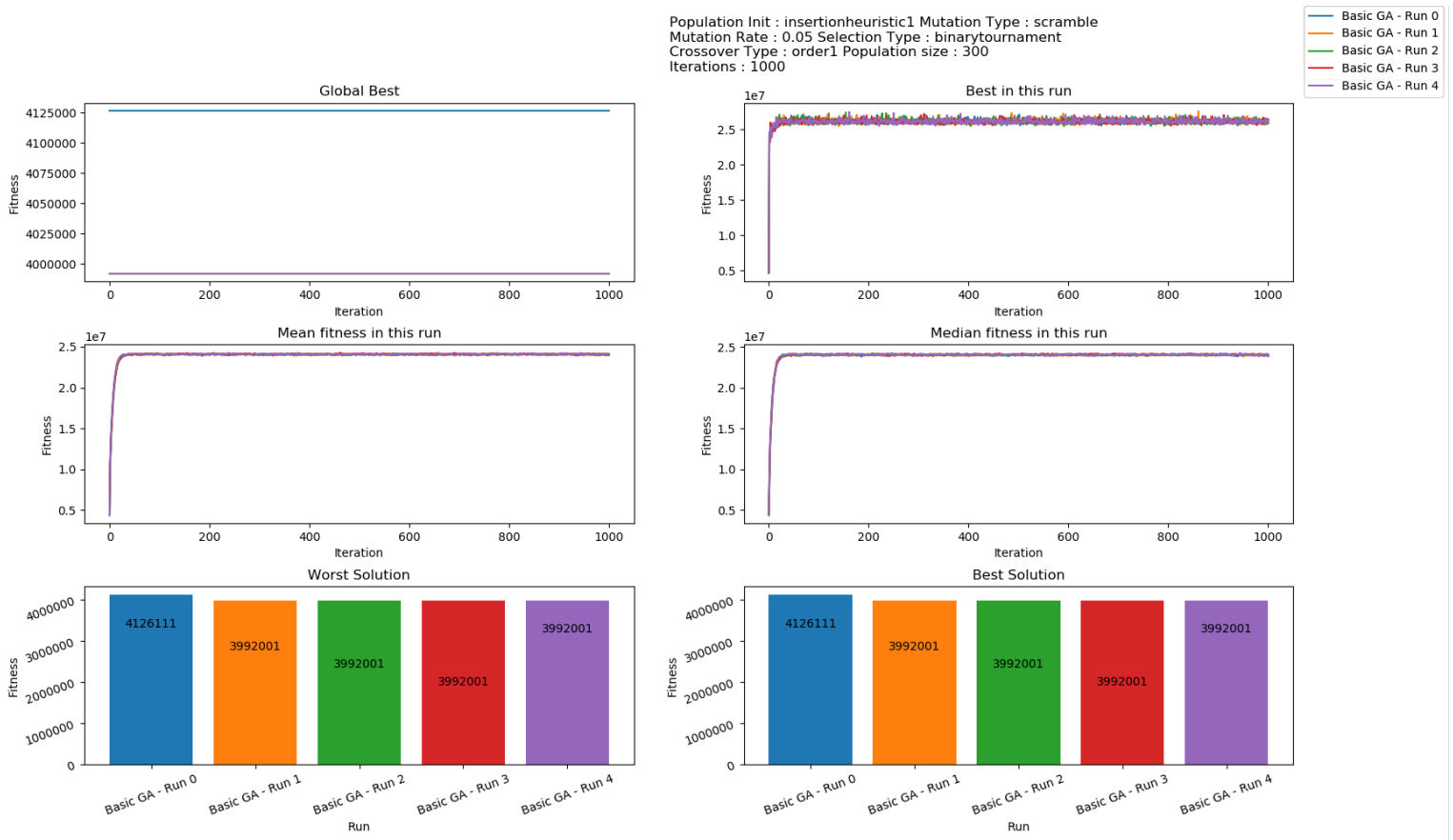
Insertion Heuristic population initialization, scramble mutation, order-1 crossover, binary tournament selection.

In this configuration, the initial population selection produces a solution that is an order of magnitude better than random population selection.

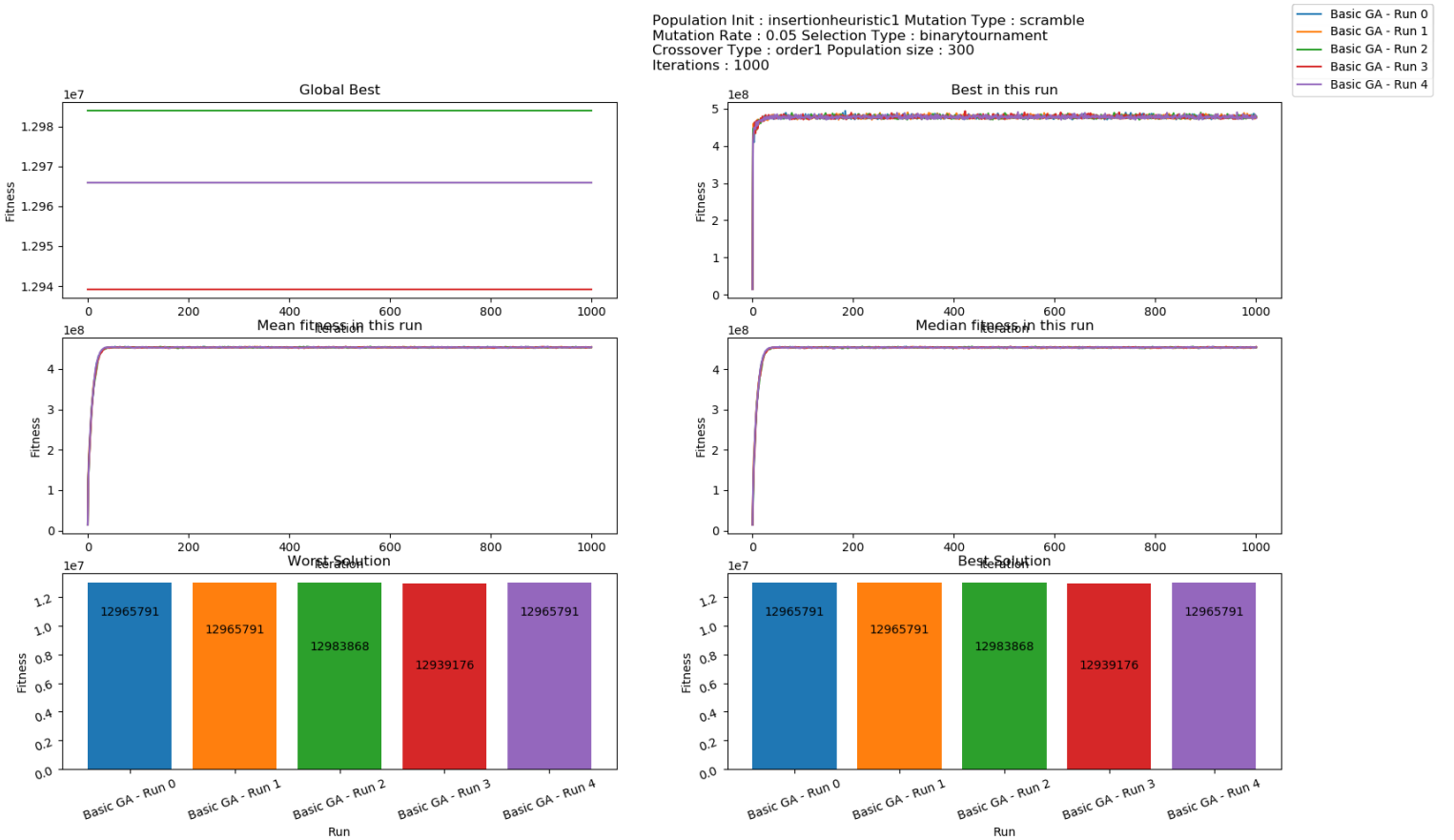
The GA, however, is not able to improve upon this solution, and it consistently found worse solutions. The fitness rose sharply, till stabilizing at a value about 5x the initial solution.

We will see later that some of this was mitigated using elitist selection.

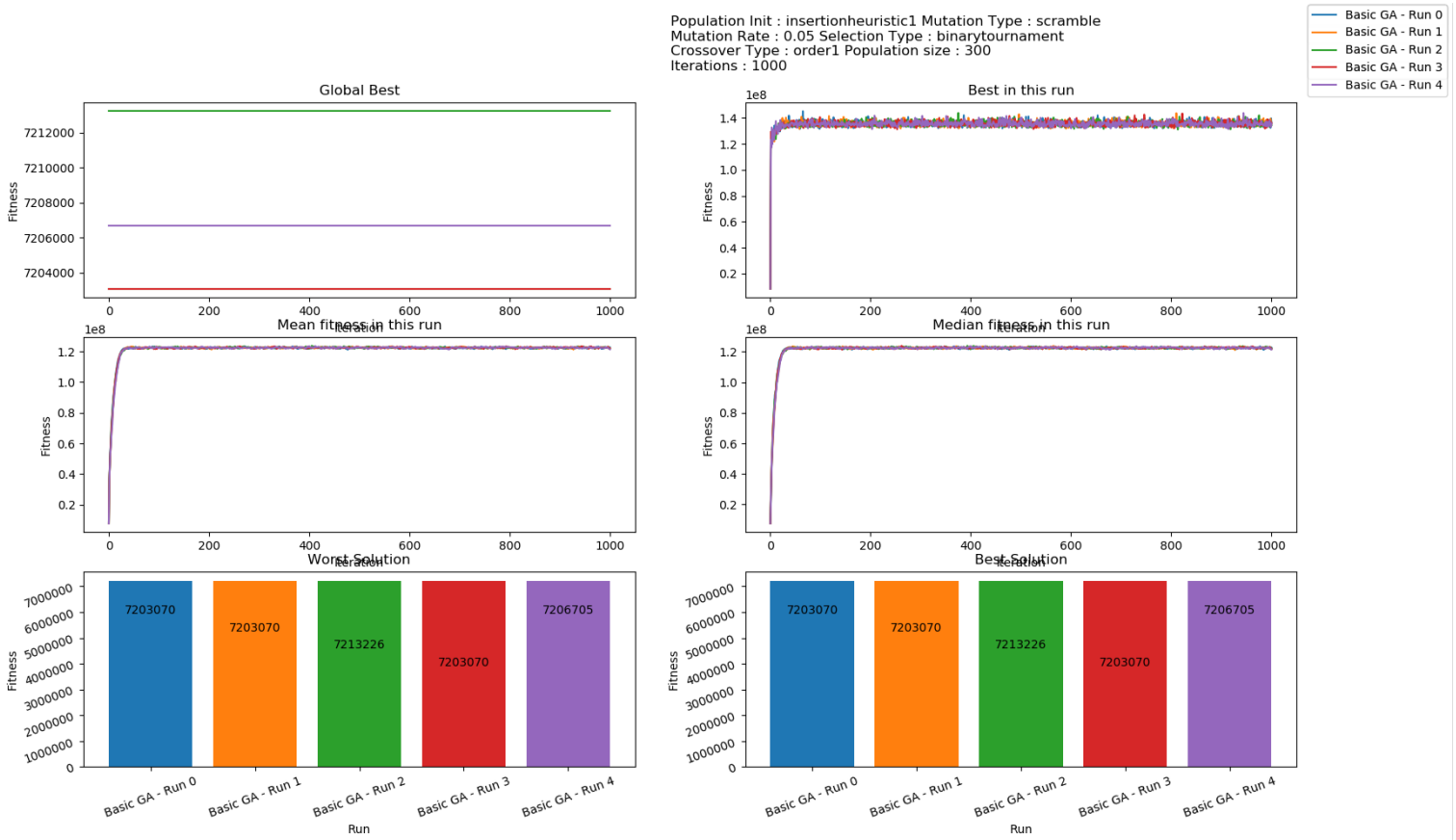
Inst-0.tsp



Inst-5.tsp



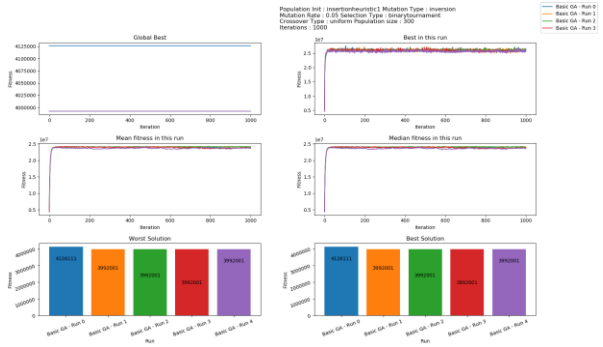
Inst-13.tsp



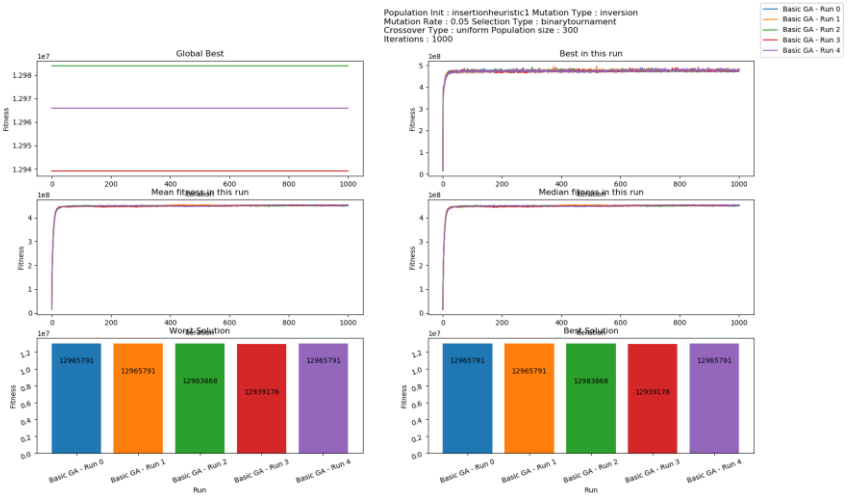
Configuration 6

Again, the results were similar to configuration 5. All the observations in configuration 5 are also valid here.

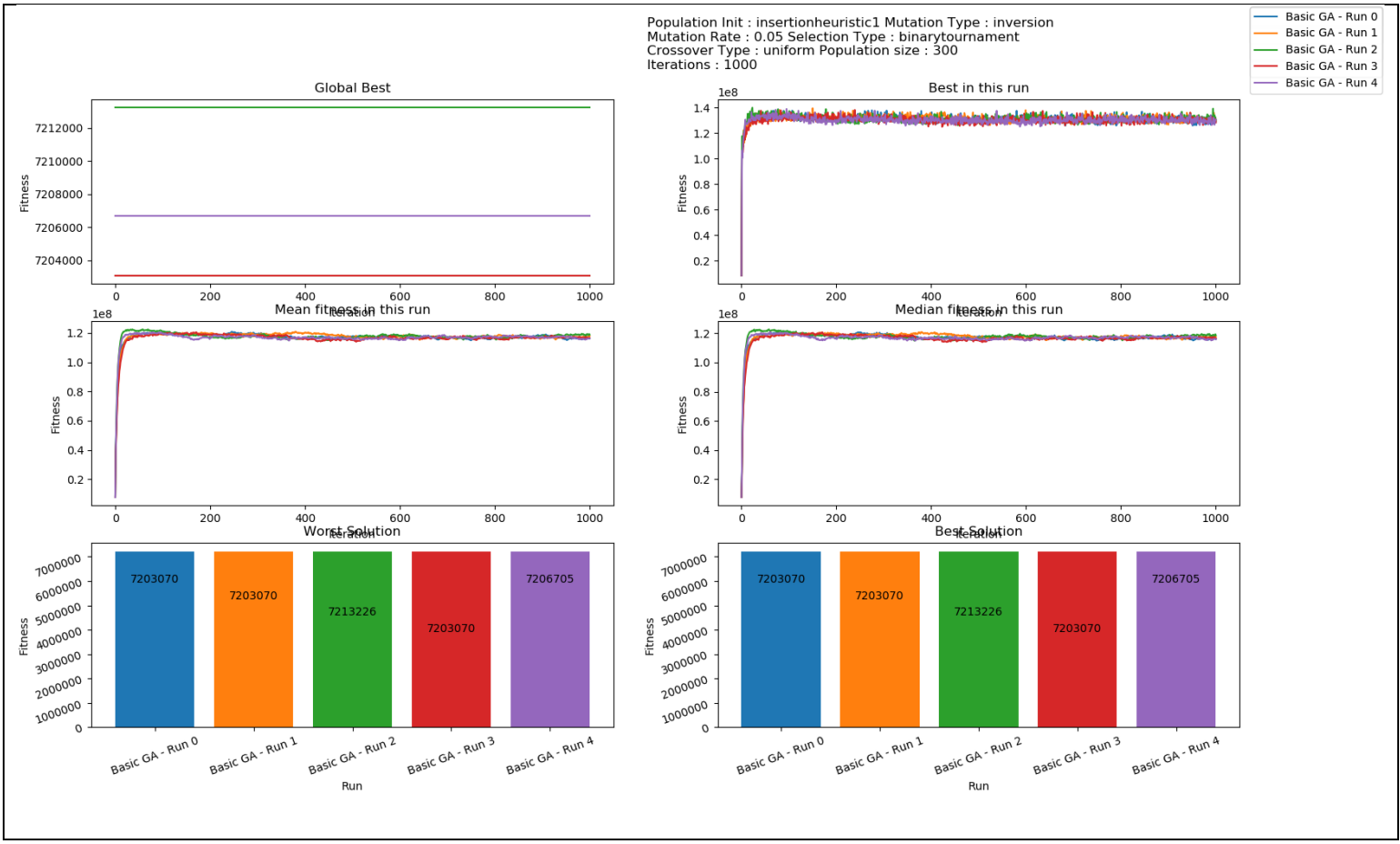
Inst-0.tsp



Inst-5.tsp



Inst-13.tsp



Comparison between Configurations

Speed: Configurations 2, 4 and 6 are much faster than 1, 3 and 5.

This would indicate that order-1 cross-over is faster than uniform crossover on an average.

Configurations 5 and 6 produced results that were an order of magnitude better than the other configurations. But that is because of the initial population selection heuristic. *The GA didn't perform much better in both of these configurations, and in the last chart, we show that all of the GA's ended up roughly in the same place after 1000 iterations.* The difference in optimality was solely due to a better initial population selection.

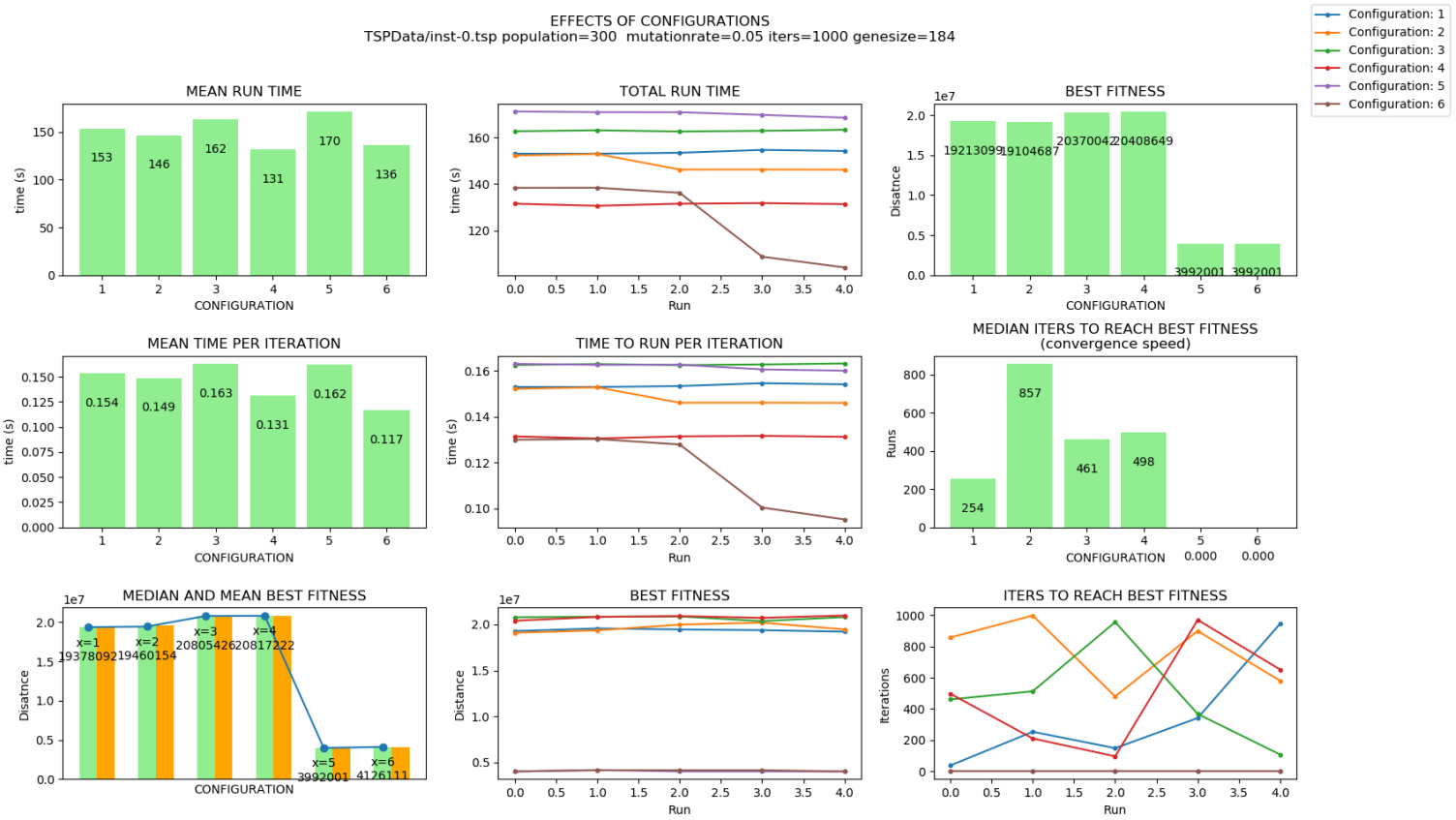
Configurations 1 and 2 performed slightly better than configurations 3 and 4 in terms of giving the better solution.

In terms of the number of iterations needed to converge[1] to the best solution the GA could find, configuration 2 performed the worst. Configuration 1 performed better than configuration 3, which performed better than configuration 4.

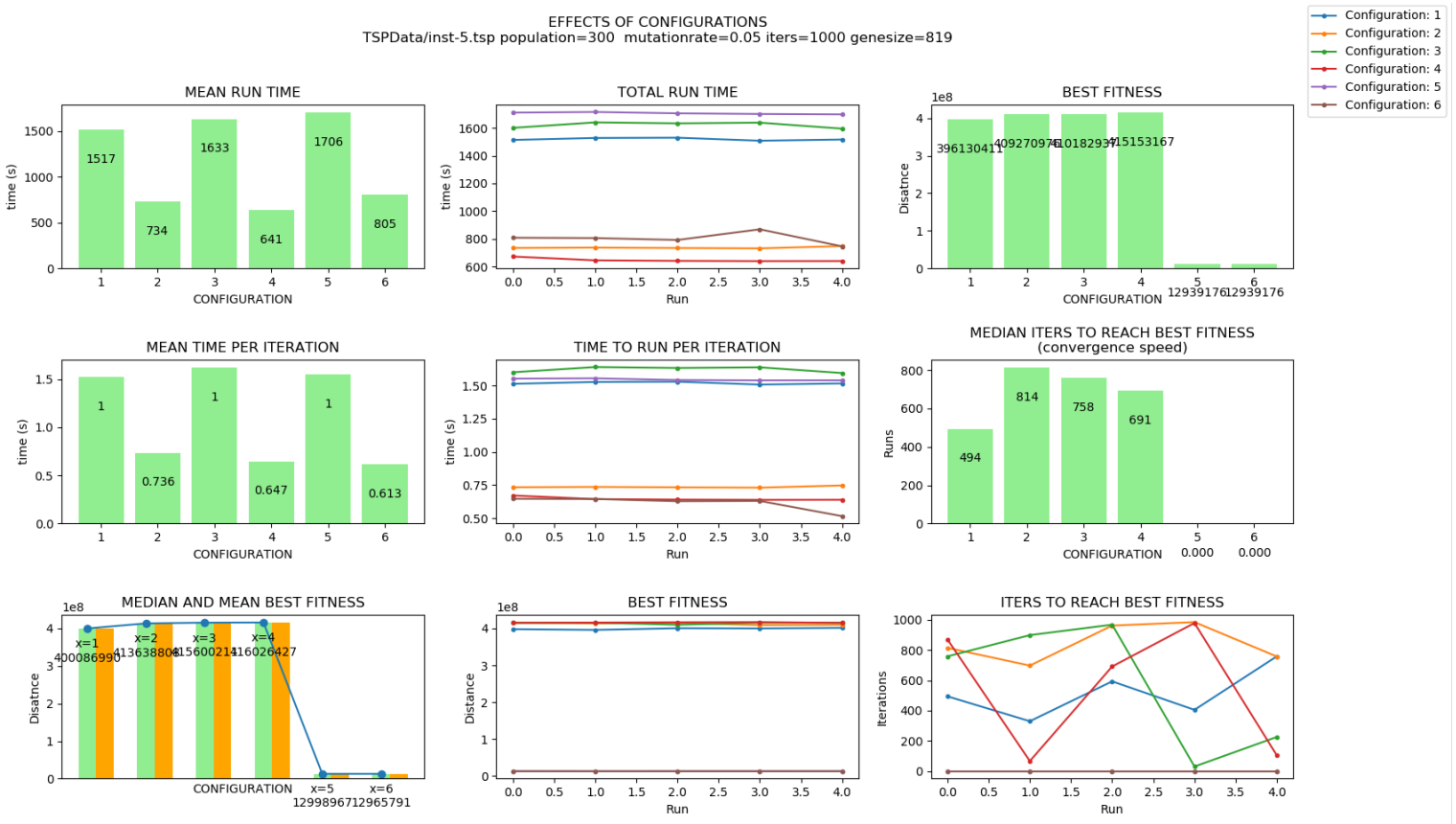
This could be down to the fact that both uniform crossover and scramble mutation produce more variation on each application, and with such variation, the GA takes time to converge. The combination of the two takes the greatest time to converge.

[1] The term "convergence" is used loosely here and elsewhere in this report. It is not used in the same terms as mathematical convergence, although there is a similarity. It is hard to know if a GA has converged, or whether it will find a better solution because it makes a step jump with a mutation. We also do not know the optimal value, so empirically, it is hard to say whether it has reached close to the optimal value. In this report, convergence is used as the last best value found in a run after a reasonable number of runs and where we do not see any significant improvement after a period of time; and the number of the iteration during which it found the value is the speed.

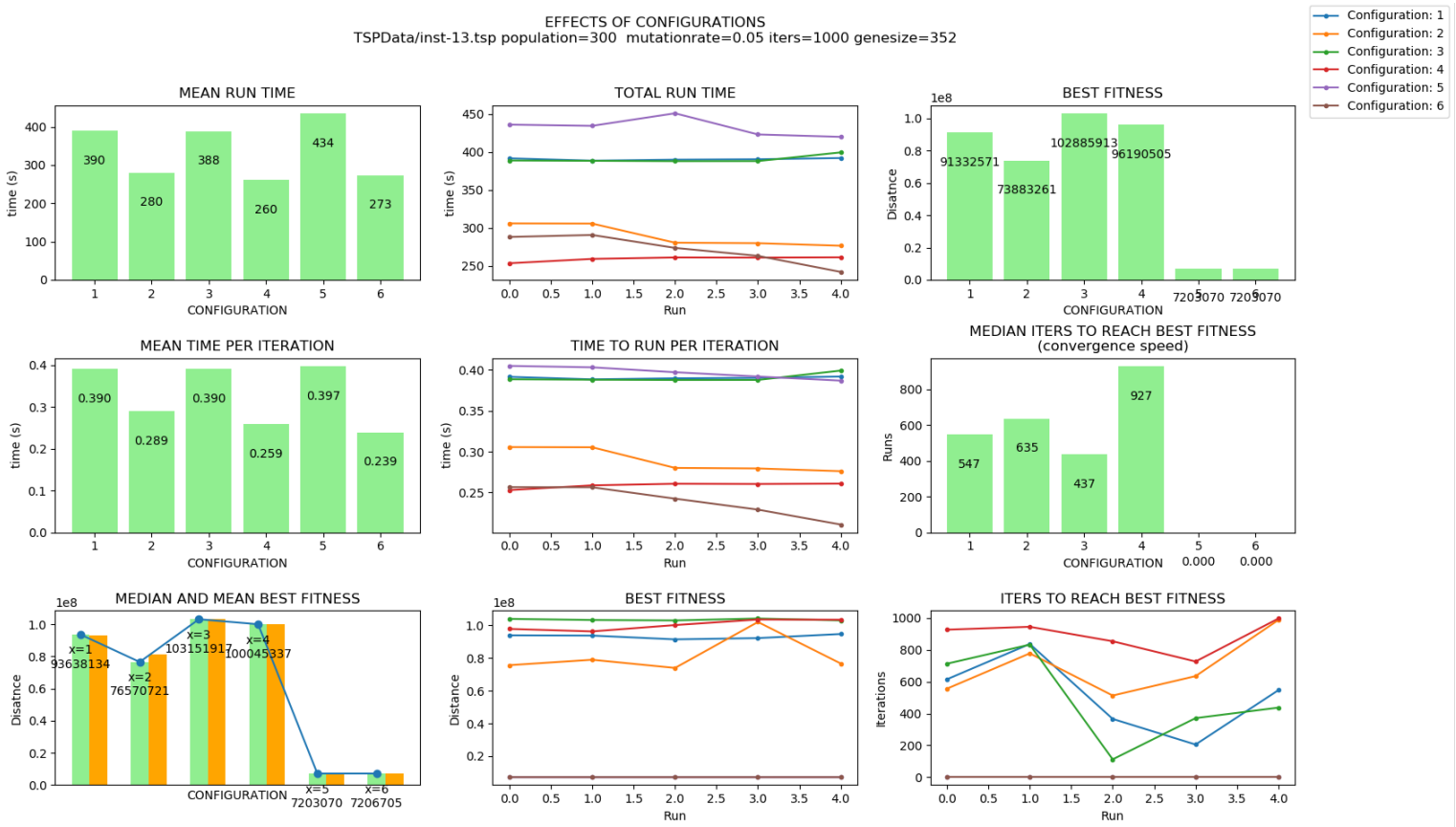
Inst-0.tsp



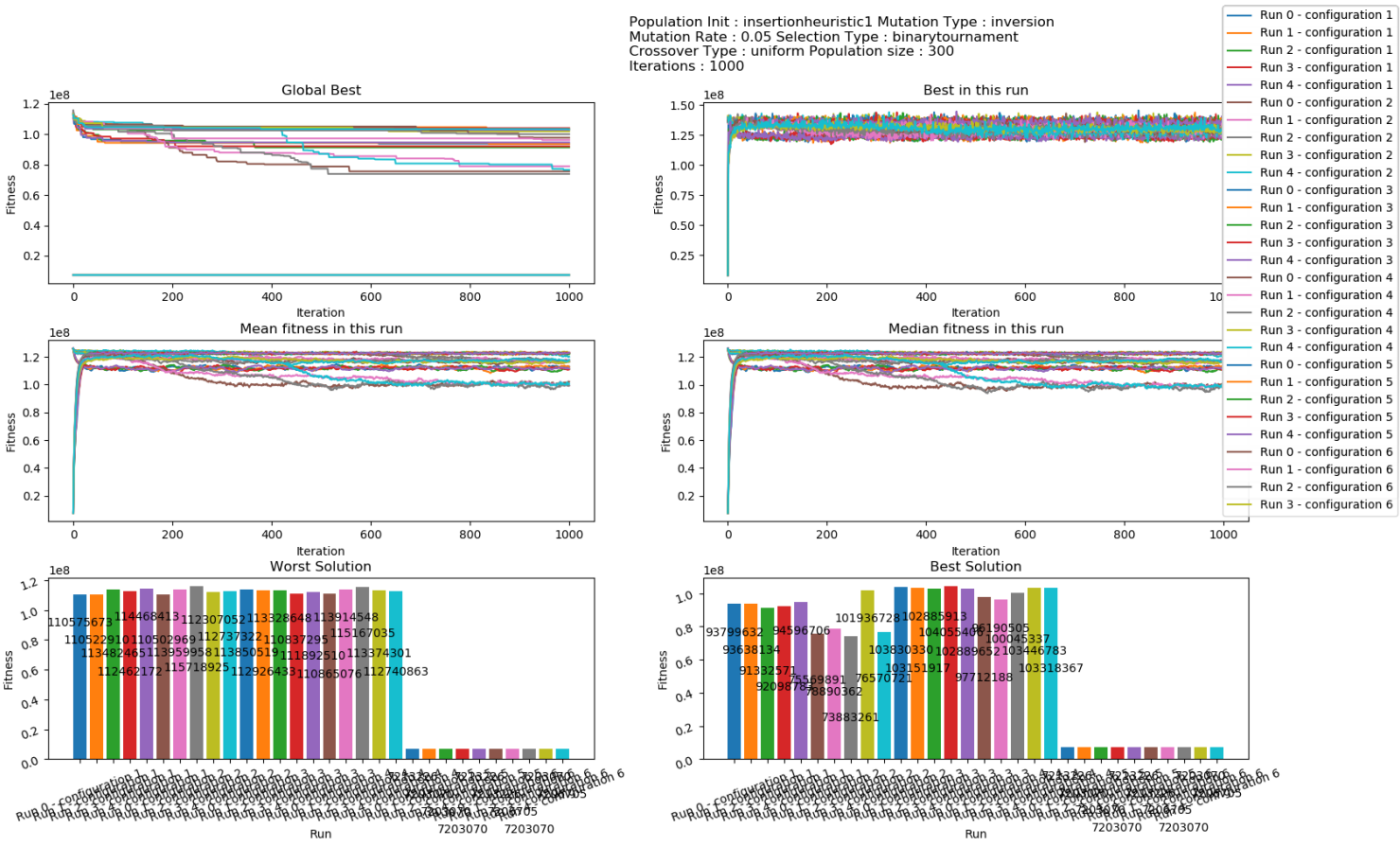
Inst-5.tsp



Inst-13.tsp



The following chart shows that no matter where the initial population started from, the GA algorithm made a bad solution better, and a good solution worse, and roughly converged to the same location.



Varying Parameters

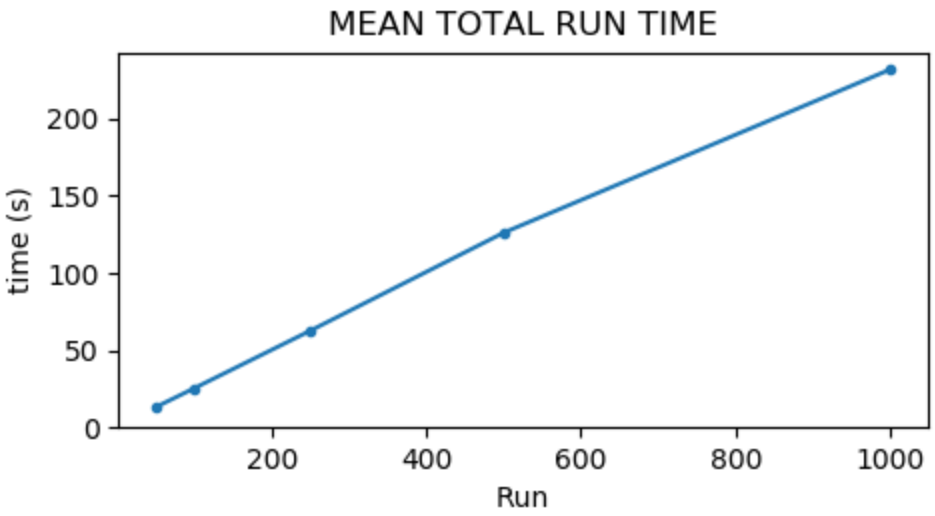
The effect of varying different parameters is discussed here.

Charts were plotted for inst-0.tsp, inst-5.tsp and inst-13.tsp, however, for the benefit of brevity, only the charts for inst-0.tsp and inst-5.tsp are presented here. The remaining charts are saved as pickle files, and can be viewed using the script plotagain.py.

Varying Population Size

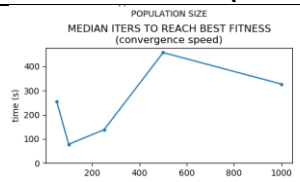
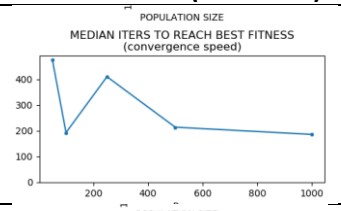
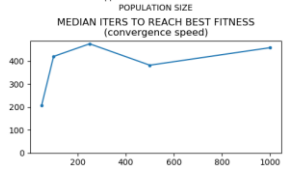
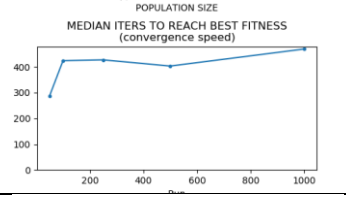
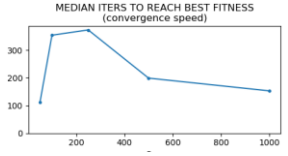
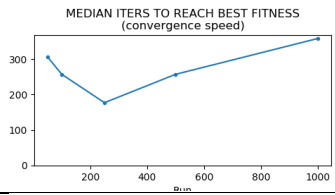
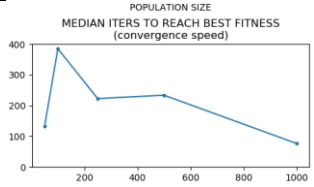
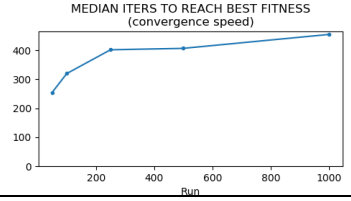
Total Run Time

Throughout this experiment, the mean total run time showed a linear relationship with the population size. This trend was observed for all configurations, but is only presented below for config-1 and inst-0.tsp.



Convergence Speed (time to reach best fitness)

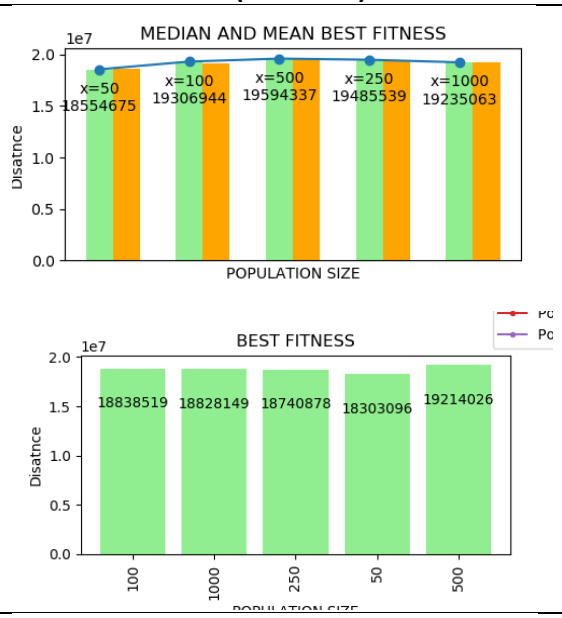
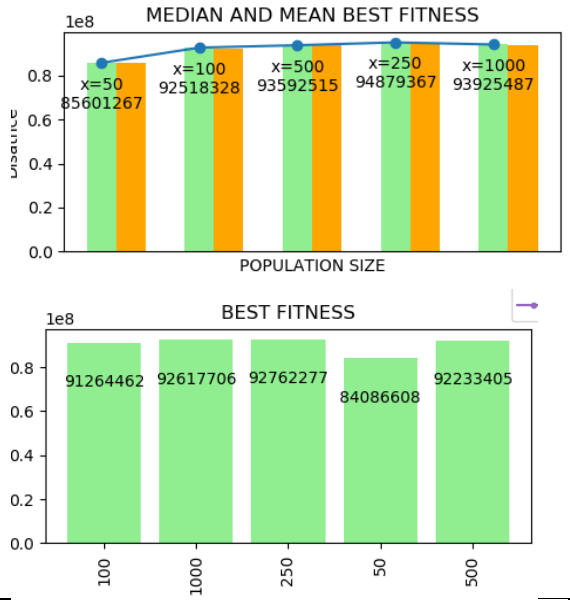
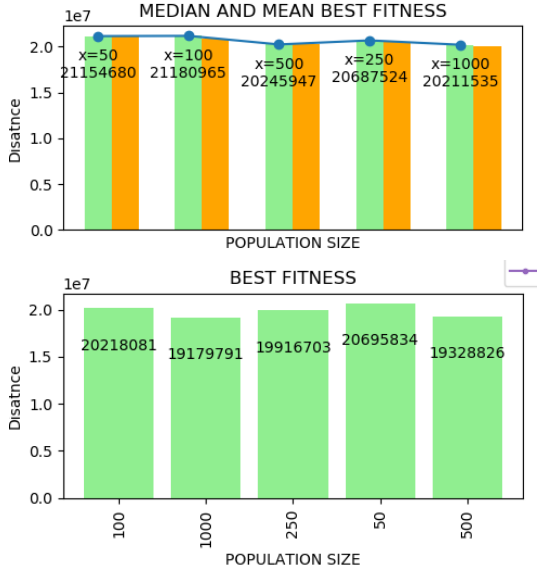
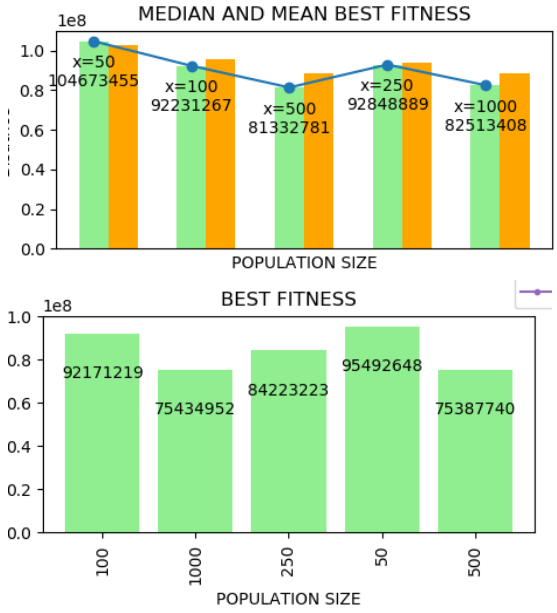
The general trend that was noted was that the higher the population, the more time it took to converge to a solution. The trend however was not strong, and there were many graphs which showed a different behavior.

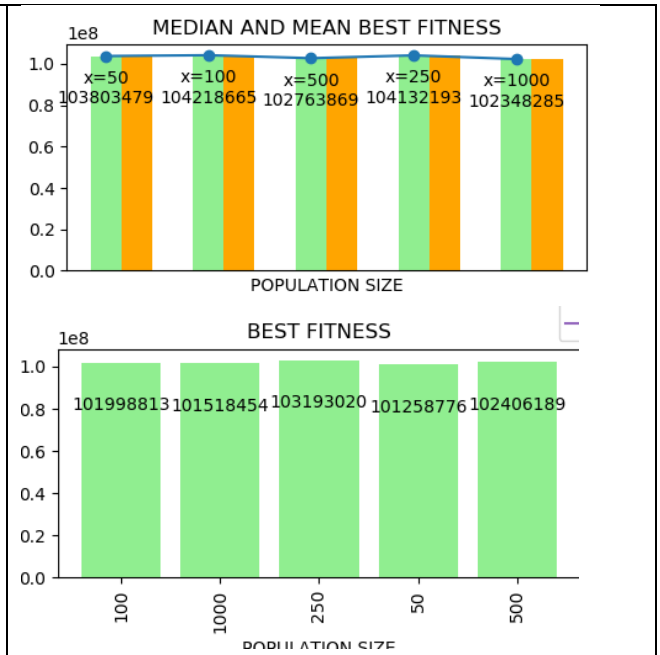
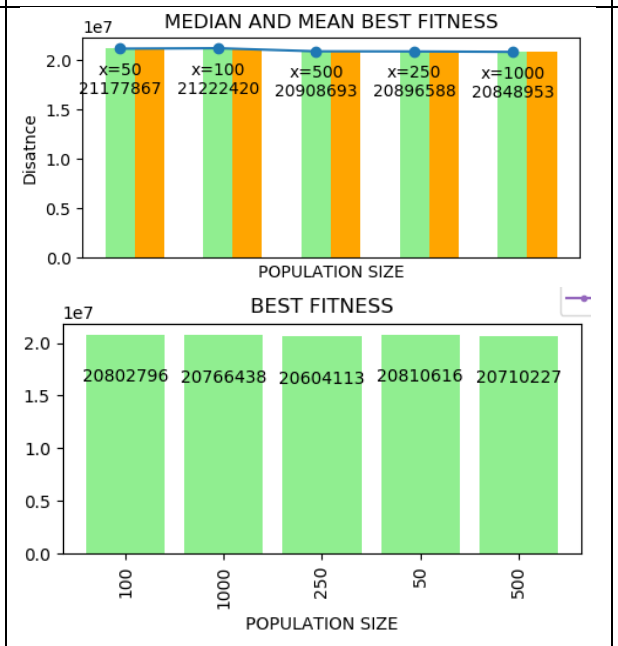
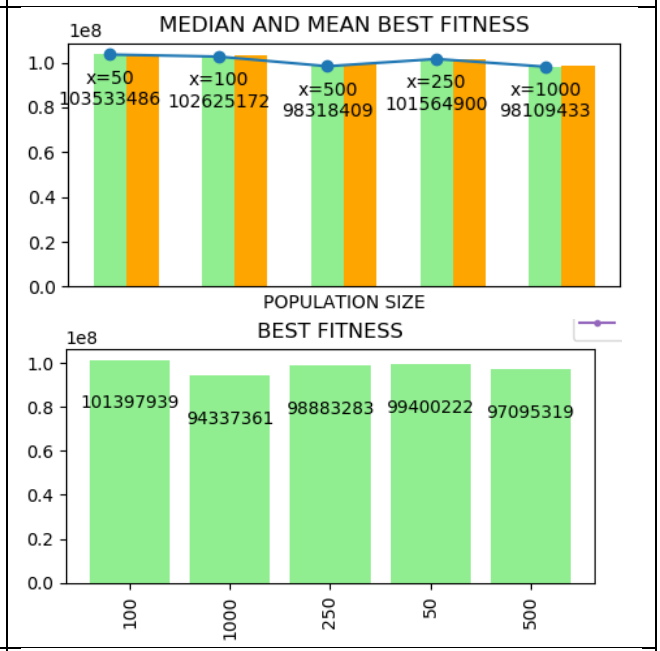
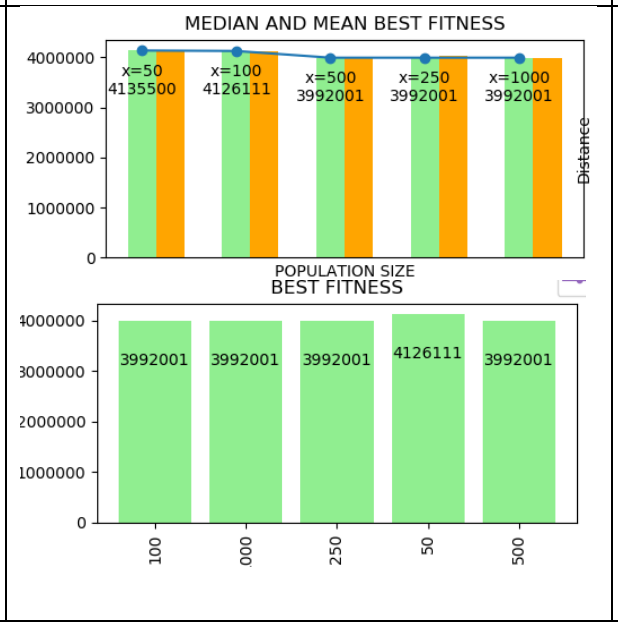
Configuration	Inst-0.tsp (184 cities)	Inst-5.tsp (819 cities)
1		
2		
3		
4		
5	n/a since the initial population contains the best	n/a since the initial population contains the best
6	n/a since the initial population contains the best	n/a since the initial population contains the best

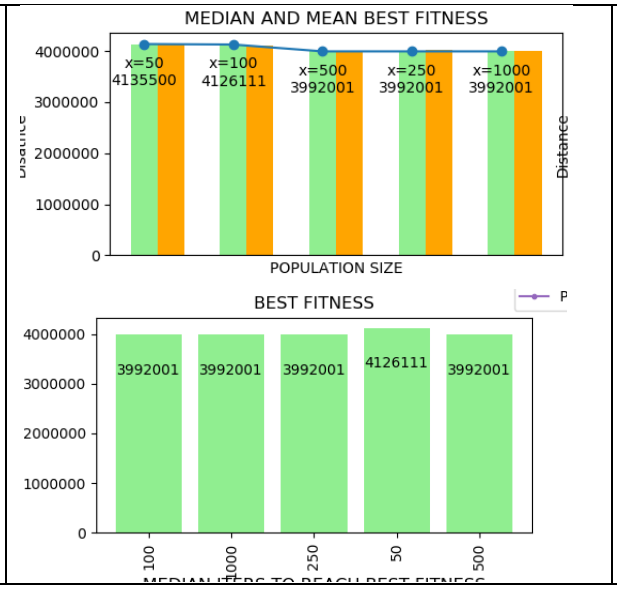
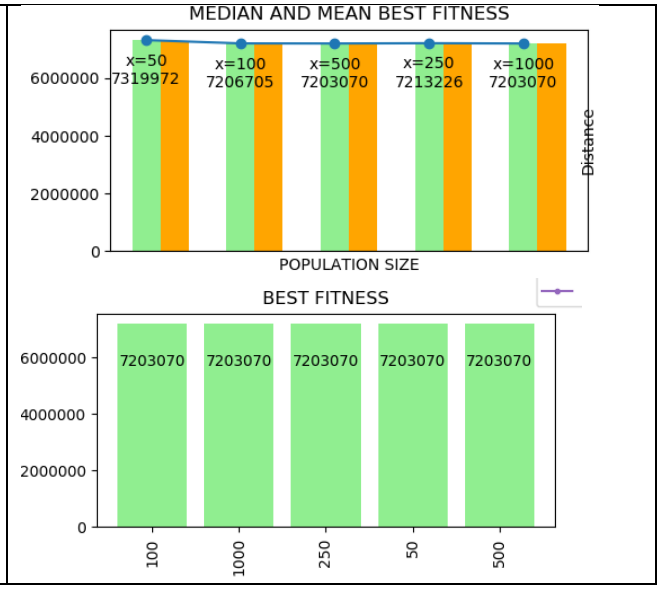
Fitness Achieved

The mean and median best fitness across 5 runs are presented, as the best fitness of all the runs.

The trend that was noticed was that the higher the population, the better solutions it tended to find.

Configuration	Inst-0.tsp (184 cities)	Inst-5.tsp (819 cities)
1		
2		

3	 MEDIAN AND MEAN BEST FITNESS 1e7 x=50 21177867 x=100 21222420 x=500 20908693 x=250 20896588 x=1000 20848953 POPULATION SIZE BEST FITNESS 1e7 20802796 20766438 20604113 20810616 20710227 100 1000 250 50 500 POPULATION SIZE	 MEDIAN AND MEAN BEST FITNESS 1e8 x=50 103803479 x=100 104218665 x=500 102763869 x=250 104132193 x=1000 102348285 POPULATION SIZE BEST FITNESS 1e8 101998813 101518454 103193020 101258776 102406189 100 1000 250 50 500
4	 MEDIAN AND MEAN BEST FITNESS 1e7 Disatnce x=50 21177867 x=100 21222420 x=500 20908693 x=250 20896588 x=1000 20848953 POPULATION SIZE BEST FITNESS 1e7 20802796 20766438 20604113 20810616 20710227 100 1000 250 50 500 POPULATION SIZE	 MEDIAN AND MEAN BEST FITNESS 1e8 x=50 103533486 x=100 102625172 x=500 98318409 x=250 101564900 x=1000 98109433 POPULATION SIZE BEST FITNESS 1e8 101397939 94337361 98883283 99400222 97095319 100 1000 250 50 500
5	 MEDIAN AND MEAN BEST FITNESS x=50 4135500 x=100 4126111 x=500 3992001 x=250 3992001 x=1000 3992001 Distance POPULATION SIZE BEST FITNESS 3992001 3992001 3992001 4126111 3992001 100 000 250 50 500	 MEDIAN AND MEAN BEST FITNESS x=50 7319972 x=100 7206705 x=500 7203070 x=250 7213226 x=1000 7203070 Distance POPULATION SIZE BEST FITNESS 7203070 7203070 7203070 7203070 7203070 100 1000 250 50 500

6		

Varying Mutation Rate

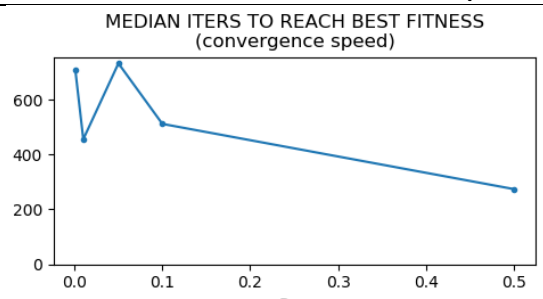
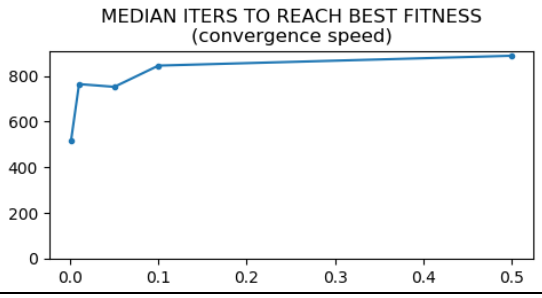
The same graphs were plotted for varying mutation rate as well. However, for the benefit of brevity, only the charts for inst-0.tsp are presented here. The remaining charts are saved as a pickle

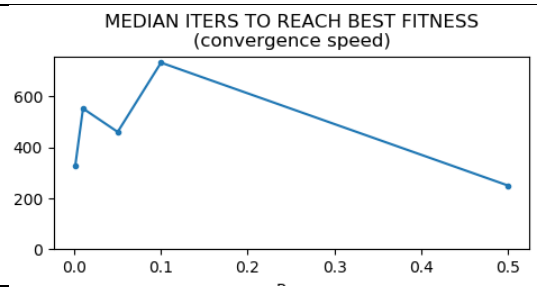
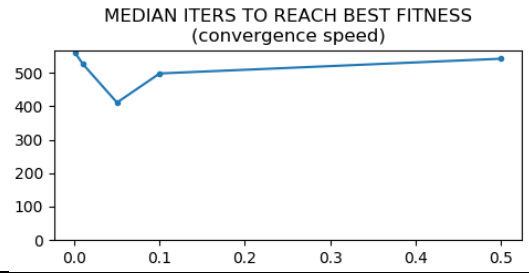
Total Run Time

The total run time remained fairly constant as the mutation rate was increased.

Convergence Speed

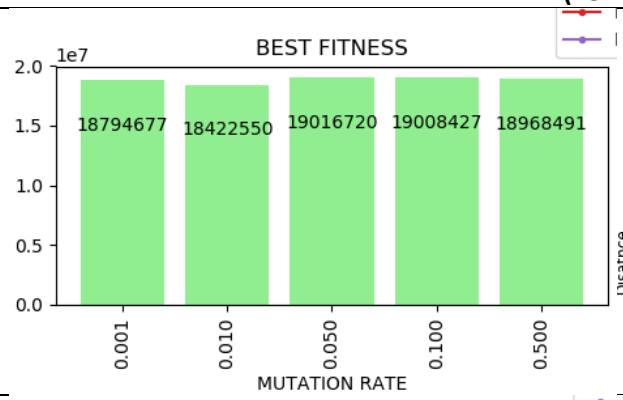
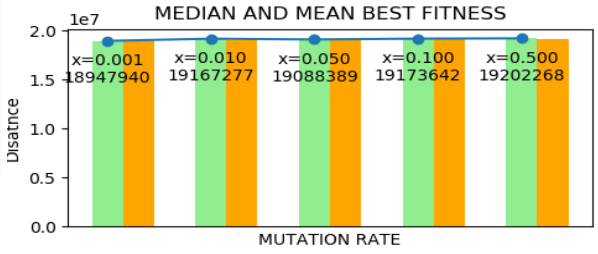
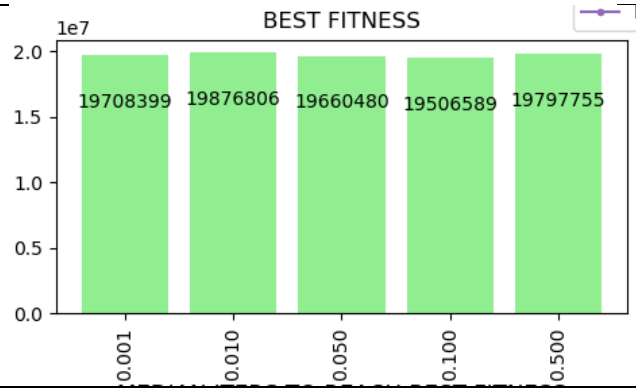
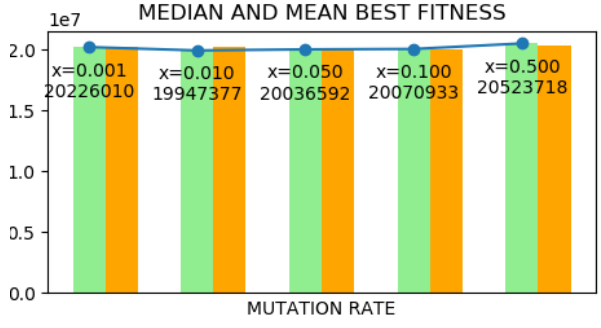
No consistent trend was noticed in convergence speed as mutation rate was changed.

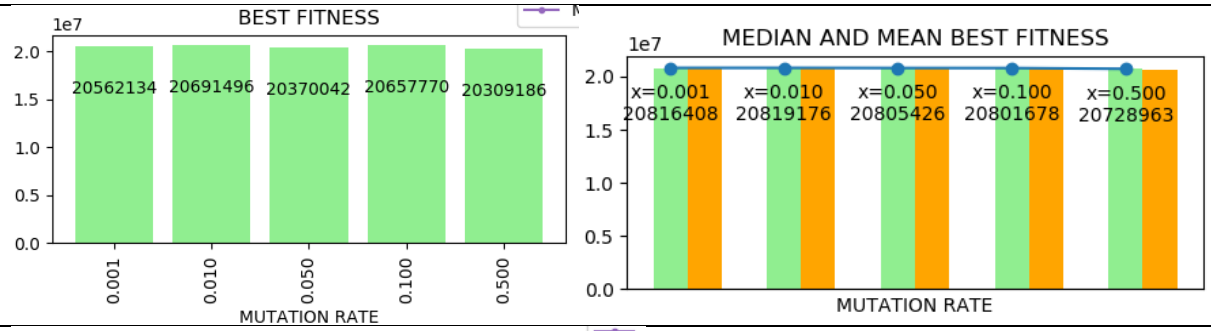
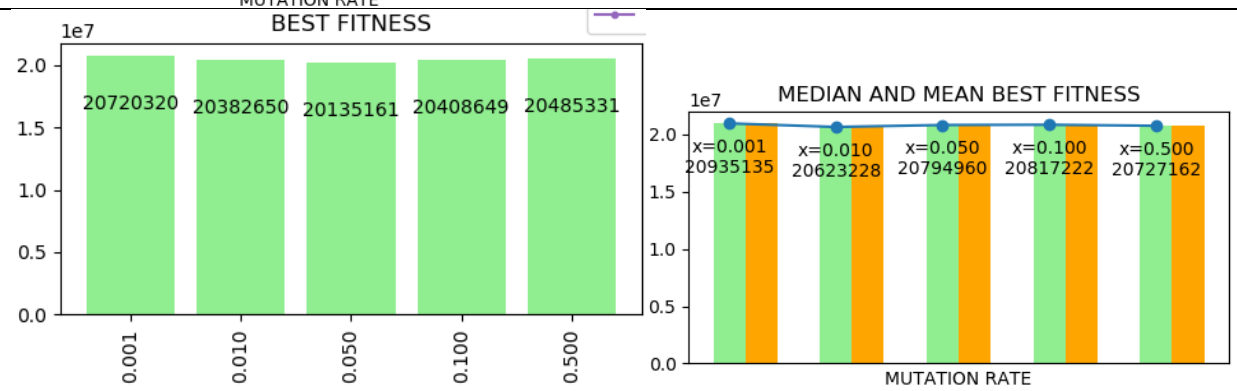
Configuration	Inst-0.tsp (184 cities)
1	
2	

3	
4	
5	**n/a since the first population generated contains the best individual**
6	**n/a since the first population generated contains the best individual**

Solution Optimality

No strong trend was noticed between the quality of solutions and the mutation rate. However, there is a weak sweet spot between the mutation rates of 0.01 and 0.05.

Configuration	**Inst-0.tsp (184 cities)**
1	 
2	 

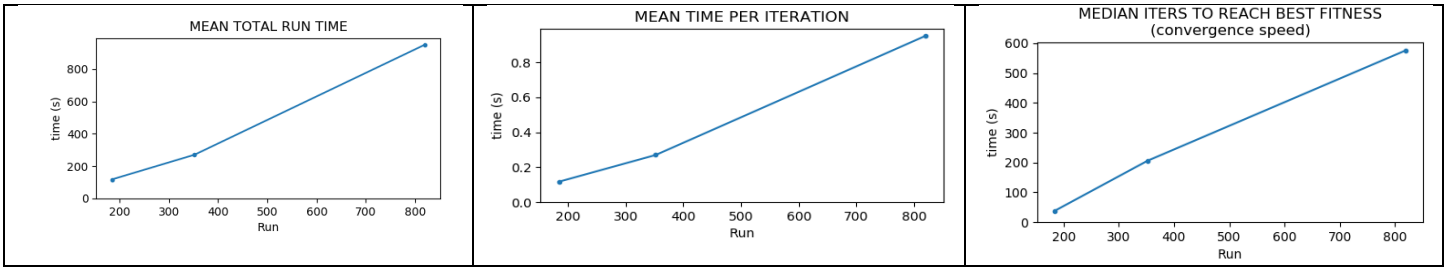
3	
4	
5	**Constant at 3992001 since the first population contains the best individual.**
6	**Constant at 3992001 since the first population contains the best individual.**

Varying Number of Genes

The number of genes in any problem and data set combination is fixed, so it is meaningless to measure optimality of solution by varying number of genes. However, number of genes have a bearing in performance, and it is useful to see the relation between number of genes and the running time.

It was observed that the running time is linearly related with the number of genes.

Linear growth was also observed at the speed at which the GA converges to a solution.



Elitism

Replacing the population after each iteration can be done in many ways. The default way is to replace the entire population with children, and retain no parents.

An alternative approach is to retain a certain number of the parents that have the best fitness, and fill the remaining slots from the children. This ensures that the best solutions amongst the parents never die out.

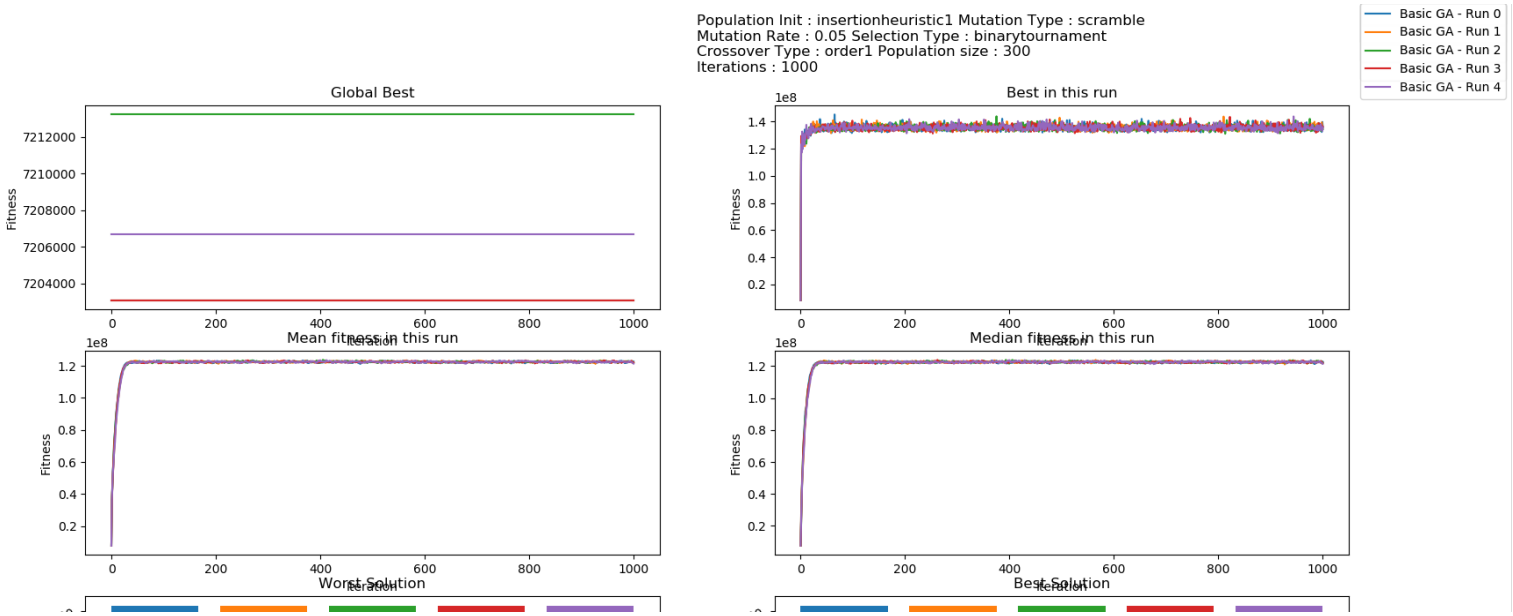
Care must be taken, however, not to overfit, ad bad solutions still might have sequences that may help optimize the problems after cross-overs, and we will be losing them.

A variant of elitism was implemented, where a fixed ratio of the best solutions from the previous generation were added to the population. This approach was able to mitigate the problems with configuration 5 and 6 where the GA

was not able to find any solution better than the initial solution, and the median solution actually became worse with every run till stabilizing at a point.

For ease of viewing, the graph from config 5 is presented below, a similar graph was seen for config 6.

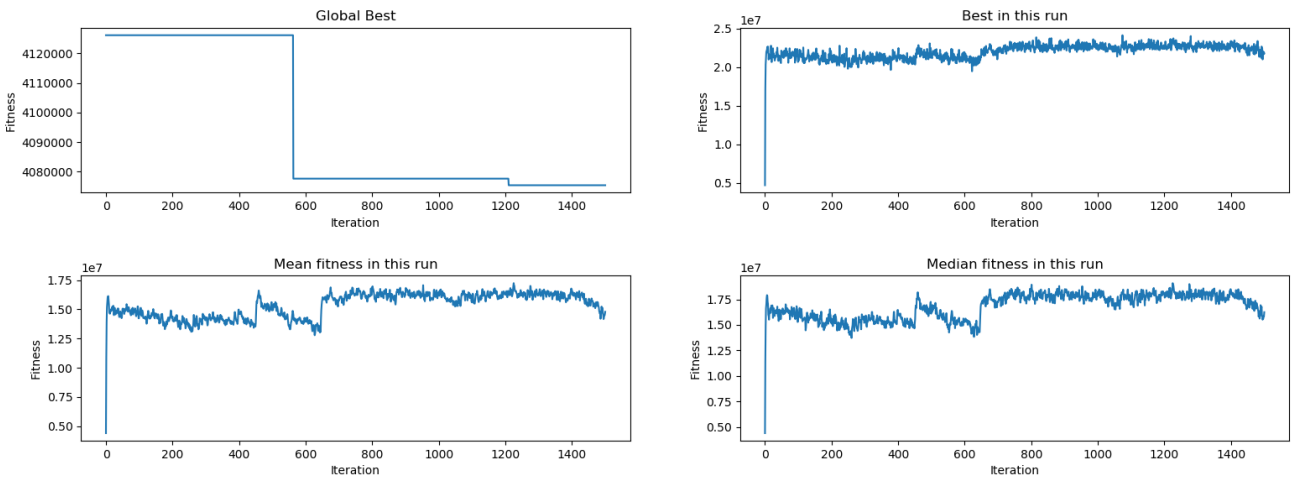
The general results was that that after implementing elitism, even with insertion heuristic, the GA was able to improve the solution to some extent. Without elitism, the GA was never able to improve a solution found with insertion heuristic.



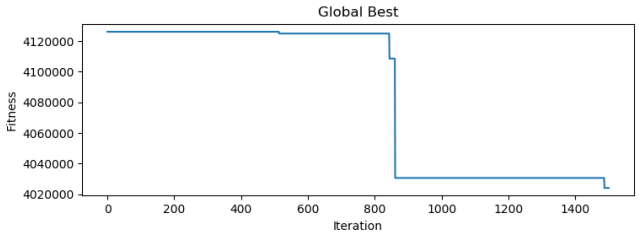
The following was tried with configuration 6 and inst-0.tsp, running each configuration once every invocation of the script.

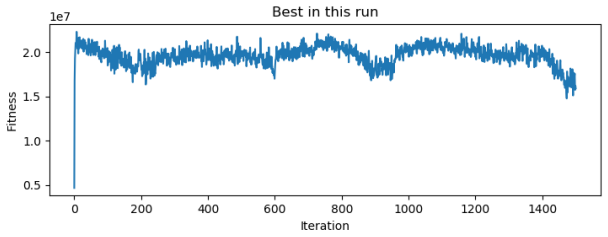
The observations were as follows:

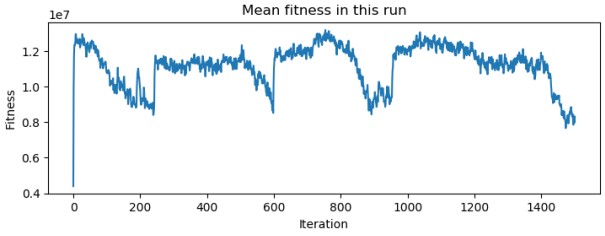
1. **As the percentage of parents was increased, the value of the distance first improved till a point, and then started becoming worse. The optimal value was around 20% of replacement.**
2. **The mean and median fitness in every run became less variable as the percentage was increased, as expected, as there is no change in the parents, and all change comes from the children. So a higher number of parents means less variation. This, however, is not of any benefit in finding an optimal solution.**

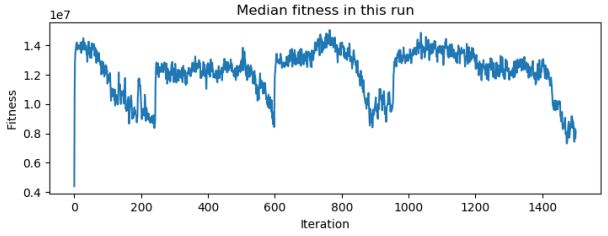
Percentage Parents Retained	
10% Best = 4075397	

20%
Best =
4024156

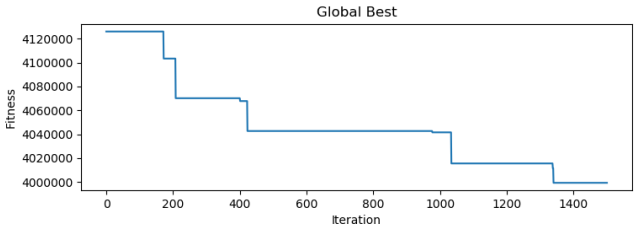


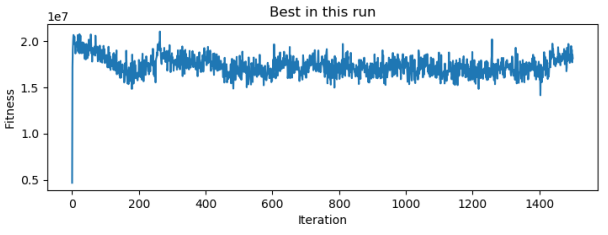


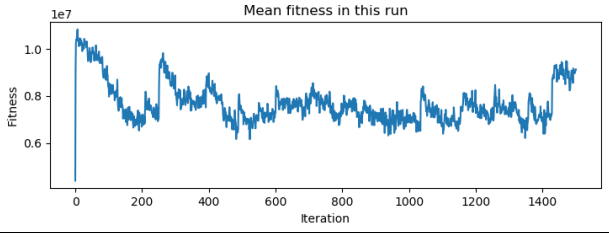


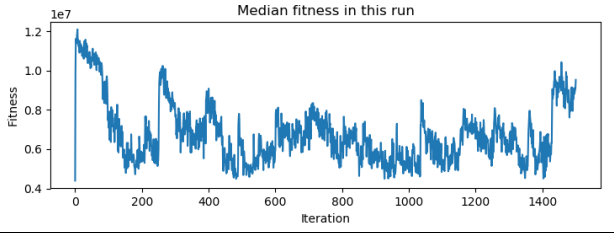


30%
Best =
39992259

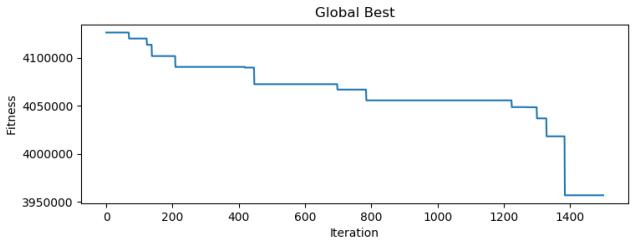


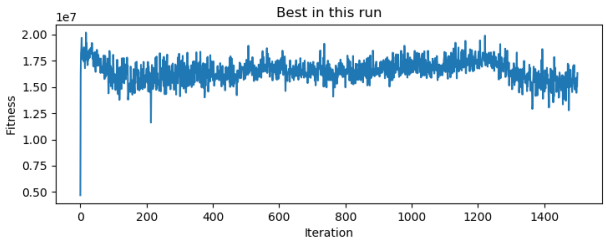


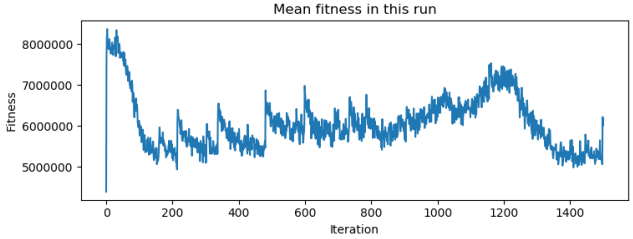


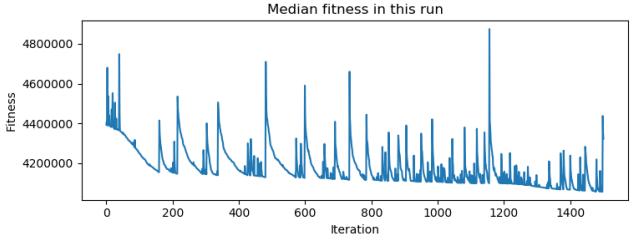


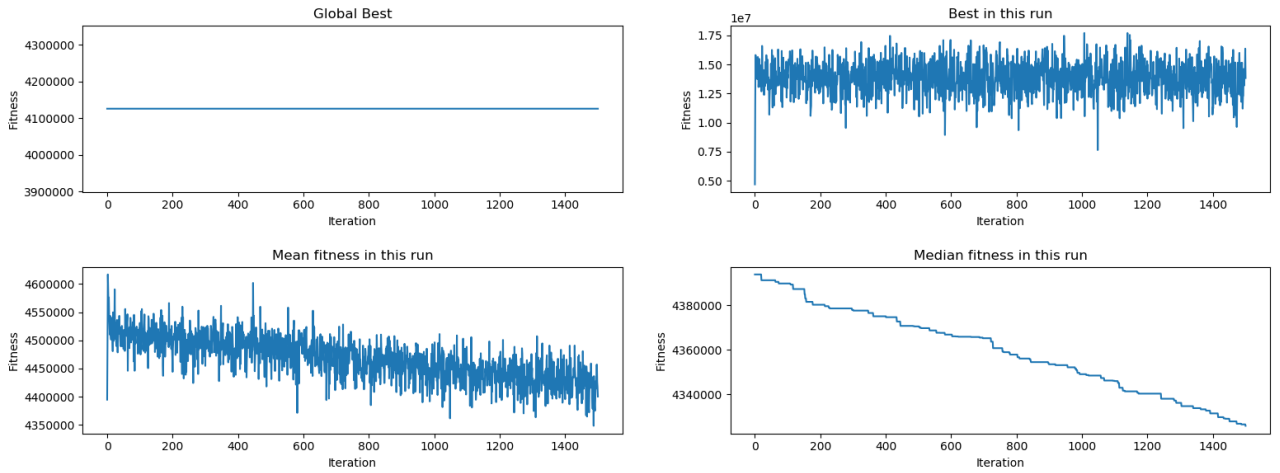
50%
Best =
3956834



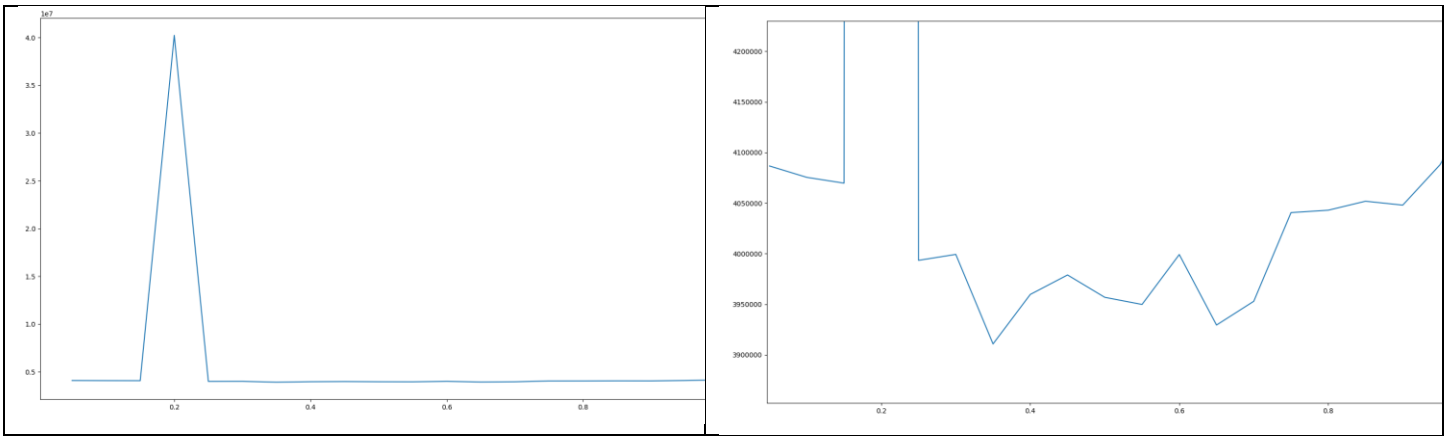




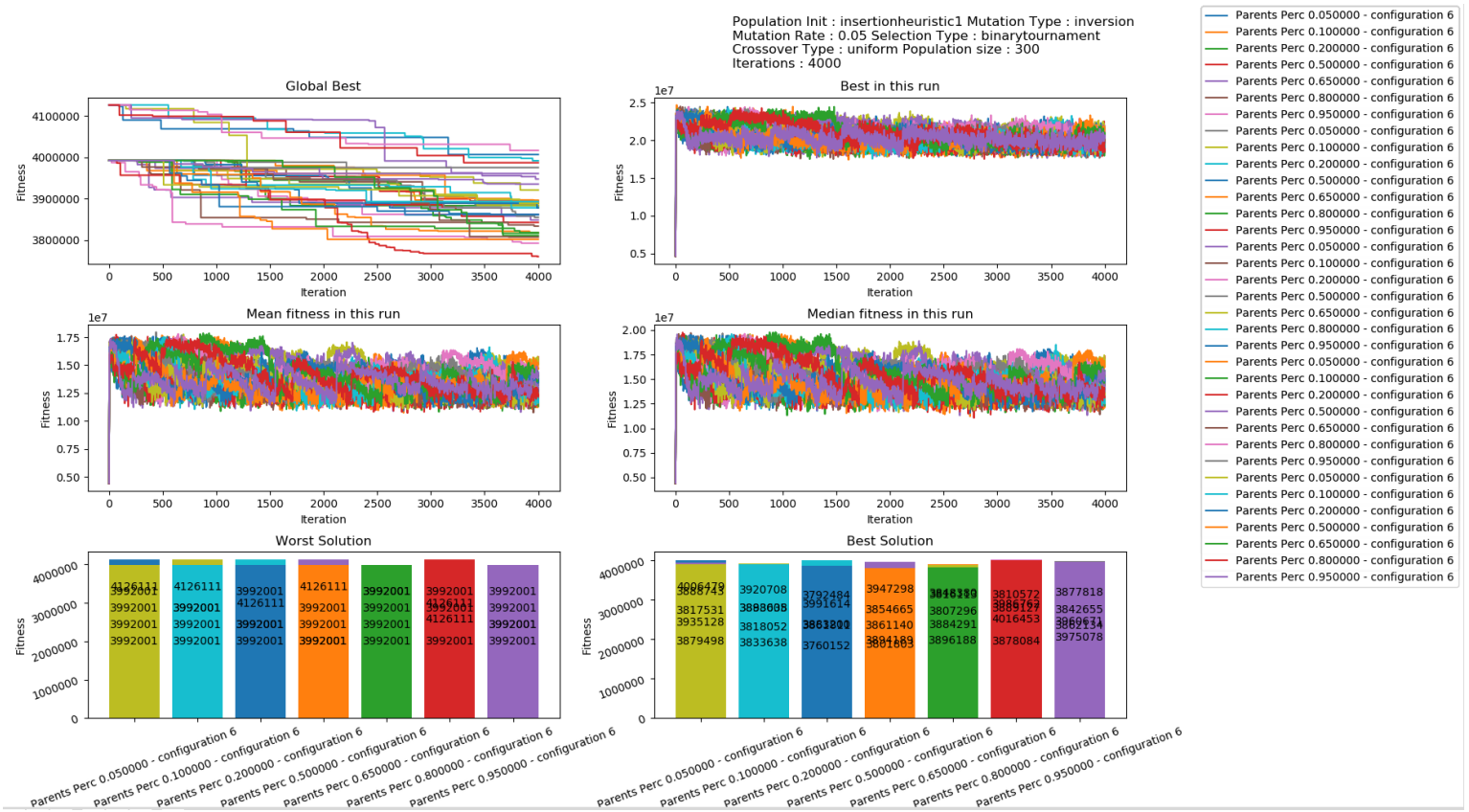


98% Best = 4126111 (no change)	

The above data plotted as the best fitness against the percentage of parents retained shows the trend. The chart on the right is a zoomed in view of the chart on the left:



In the following graph, 5 runs were run in parallel (using -vemr option), with config 6 and inst-0.tsp.
-f TSPdata/inst-0.tsp -vepr -mt -c 6 -i 4000 -nr 5 -ucl -ocl



The graph below plots mean and median fitness, best fitness, and the convergence speed w.r.t changing percentage of parents in the new population.

From the graph below, it is clear that a low value of the percentage of parents in the new population produced better results, in our case, the best solution was reached at 20% of parents.