Documentation Of Continuous And Combinational Optimisation Problem Genetic Algorithms

Jounaid Ruhomaun, February 2021

Background

The purpose of this report is to document the variety of genetic algorithm (GA) methods/scripts written using Python found at: https://github.com/jounaidr/py-genetic-algorithms. Wherever possible there has been an attempt to utilise Python/NumPy features to their fullest in order to reduce iteration within the algorithms resulting in more optimal code and higher scalability in terms of population size.

Continuous Optimisation

Basic Code Structure and Methods (Subtask 1.A)

The generation, selection, crossover, mutation and data visualisation methods can all be found under *ga-continuous-distrib/common.py*.

The first method, *generate_population(…)* returns a two dimensional NumPy array of size *population_size * individual_size* which is populated with random real values with the bounds specified by their respective arguments (*lower_bound, upper_bound*). Each value or 'gene' is represented by a 64 bit precision float data type. An iterator is used within this method during gene value generation, however, since initial population generation is done outside the main GA loop for each optimisation problem, this will cause no noticeable performance loss.

The basic selection method used is *selection_roulette(…)* which calculates selection probabilities for each individual in the population based on their respective fitness value in regards to the total population fitness value. The amount to be selected is calculated using the *crossover_rate* in regards to the total population size. In order to get around iteration of the population, *np.random.choice* is used to generate a 2D array of indexes using the previously calculated probabilities, of size *selection_amount * 2.* Individuals from the population with said indexes are then taken to the form the *parents* 3D array, of size *selection_amount * 2 * individual_size,* which is subsequently returned. This return format allows for easier crossover in the later methods, as the parents are already in pairs. It is also important to note the argument, *multi_selection,* which is default to true, as this will allow individuals to be selected more than once to be a parent.

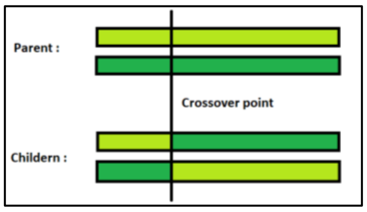


Figure 1

The basic crossover method used is *single_point_crossover_opt(…)* which takes in a *parents* array with the specified format mentioned in the selection method before creating an empty copy of said array format to store the children. The *crossover_point* (*cp*) is generated as a random integer within the range $0 \leq cp \leq individual\_size$. The children are then generated following the diagram shown in ***Figure.1***[1]. This is optimised using *np.hstack* in combination with Python array slicing at the *crossover_point*. The resultant *children* array is reshaped so that the pairs are stacked vertically before being returned.

The basic mutation method is defined in *uniform_mutation(…)*. Initially children are selected to be mutated by an index selection Boolean array generated from the *children* array argument. The probability an index will be selected (True) is specified by the *mutation_rate* value. A 2D array of gene indexes to mutate is then generated for each child to be mutated. The argument *mutations* specifies how many indexes are chosen per child, for example if *mutations = 3,* then three genes will be selected randomly in each child to be mutated. The default value of *mutations* is set to one. The mutation index array is then used to replace genes in the *children_to_mutate* array with a random value within the argument bounds (*lower_bound, upper_bound*), similar to the *generate_population(…)* method. The use of *np.arange* as well as Python broadcasting allows for the mitigation of multiple iterators during gene replacement. The mutated children are added back into the *children* array before being returned.

The final basic method in regards to the GA is *next_generation(…)* which carries out the survive functionality. An array of indexes corresponding to the lowest fitness values of size *population_size* is generated, which is used to select individuals from the current generation to 'survive' into the next. This results in a population of size *population_size* containing the fittest individuals from the previous generation, which is returned. NumPy *argpartition* is used in this method to select the fittest individuals, which again mitigates the use of iterators further increasing performance.

At the end of *common.py* there are a set of data visualisation methods. *display_population(…)* prints a populations data to console formatted using a Pandas *DataFrame*. The *display_fittest_individual(…)* prints the fittest individual from a given population, again formatted using Pandas *DataFrame*. Matplotlib *pyplot* is used to generate line graph visualisations of generations against fitness data. The full fitness range can be displayed using *plot_data_full(…)*, or a section of fitness values can be displayed using *plot_data_ylim(…)*.

Each continuous optimisation problem has its own script that can be found in the same package as common under *ga-continuous-distrib/…* for which the script name specifies its problem, for example the sum squares problem is found at *ga-continuous-distrib/sum-squares.py*. Each problem script imports the methods from *common.py* and has its own specific fitness function that returns a 1D array of fitness values that can be passed into said imported methods. Global hyperparameters can be set at the top of each script. Upon running the script, an initial population is generated and displayed before being passed into a number of threads specified by the *THREADS* global variable. Each thread submits a *main_threaded_loop(…)* task which runs the main EA loop (using the common methods). Execution time and fitness data is stored per thread and retuned as futures, which are unpacked after each thread is run and used to generate plots. The last generation of each thread is printed as well as its fittest individual and thread execution time. Before the script ends, the number of generations that ran per thread is printed, as well as if that thread contained a solution. The mean execution time of all threads is also printed.

Improved/Experimental Methods (Subtask 1.D)

There are multiple extra methods implemented within *common.py* for each of selection, crossover and mutation. These methods can be interchanged with the basic methods within each problem script to achieve different results.

The first extra selection method is *selection_rank(…),* that is essentially the same as *selection_roulette(…),* however, the probabilities used are calculated using fitness rankings rather than the actual fitness values. This method can be used when the fitness values start to have less variance between them which when using *selection_roulette(…) would* result in very similar probabilities.

The *selection_tournament(…)* method takes a different approach to the previous two, as selection probabilities are not used. The method generates a number of 'tournaments' of size *tournament_size* for the amount calculated using the *crossover_rate*, and in each tournament a winner is selected based on the highest fitness value in that tournament. The resultant *parents* array is formatted to be in pairs, as with the previous selection methods, so that the output is usable by the crossover functions. Once again great effort has been put into using NumPy features so that iteration is completely omitted within the method, which is explained further in the code comments.

The final two extra selection methods, *selection_roulette_rank(…)* and *selection_tournament_rank(…)* use a combination of selection methods based of the standard deviation of the populations fitness values. Each method will use rank selection if the fitness values standard deviation is below the specified threshold (default is one), and the other respective selection method when the values are further apart.

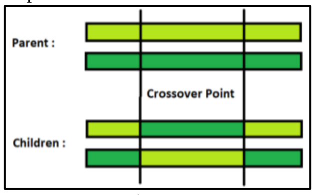


Figure 2

The first extra crossover method is *double_point_crossover_opt(…)* which is almost identical to *single_point_crossover_opt(…)*, but uses two crossover points in the *np.hstack* block so that offspring is generated as in ***Figure.2***[2].

The last extra crossover function, *single_point_crossover_multi_index(…)* is essentially the same as *single_point_crossover_opt(…)*, however, a random crossover point is used for each set of children generated. This method utilises an iterator and as such using this method will impact performance proportional to the population size. Further detail on this is given in the performance testing section of this document.

The first extra mutation function is *boundary_mutation(…)*, which is almost identical to *uniform_mutation(…)*, however, genes are replaced with either the *upper_bound* or *lower_bound* rather than a random value. The last extra mutation function is *non_uniform_mutation(…)* which works in a similar way to *uniform_mutation(…)*, however, once the average fitness has dropped below the *fitness_threshold* argument, the mutated genes are generated using a random value with boundaries close to the genes value rather than the bounds arguments. The new boundaries are calculated using: *gene_to_mutate +- avg(fitness) * gene_to_mutate*. In theory this will mean once an individual's genes are close to a solution, each subsequent generations mutations will more likely be an improvement on fitness, resulting in faster convergence and less generations before a solution is found. However, if there is a local minima below the *fitness_threshold,* that individual could optimise around said minima and wont find the global minima as the genes mutations will not be significant.

Performance Testing

Figure.3 shows the mean execution time in seconds running *sum-squares.py* with the different methods mentioned before. The 'basic' test run includes the following selection, crossover and mutation methods: *selection_roulette(…), single_point_crossover_opt(…), uniform_mutation(…)*. Each test is run with data collection and with the following control variables: *THREADS = 1; INDIVIDUAL_SIZE = 10; LOWER_BOUND = -10; UPPER_BOUND = 10; CROSSOVER_RATE = 0.8; MUTATION_RATE = 0.2; MUTATIONS = 1; GENERATIONS = 10,000.* Although each problem script has its own respective fitness function, they are all calculated in the same way with NumPy arrays

POPULATION_SIZE	10	100	1000	10000
basic	2.23	2.62	5.38	55.65
double_point_crossover_opt	2.29	2.67	5.37	60.18
single_point_crossover_multi_index	2.72	5.97	38.13	382.03
selection_rank	2.79	3.21	6.06	66.26
selection_tournament(tournament_size=10)	2.47	2.97	6.10	97.43
boundary_mutation	2.35	2.77	5.51	59.83
non_uniform_mutation + selection_rank	3.62	4.06	7.43	80.42

Figure 3

and therefore have the same time complexity and do not need to be tested. Also note that only a single thread is tested, obviously a larger number of threads will affect performance however this is dependent on the hardware.

Since *non_uniform_mutation* generally finds a solution for the sum squares problem within 1000 generations, when roulette selection is attempted with a sum fitness of zero, the code will break as expected, therefore rank selection must be used to test execution time of this method with 10,000 generations. ***Appendix.1*** shows a visualisation of the execution time data for: *basic; single_point_crossover_multi_index; selection_tournament; non_uniform_mutation + selection_rank,* plot using MatLab linear interpolation. As you can see, the basic code as well as some of the more expensive methods (without iteration) have similar complexity and execution times, whereas *single_point_crossover_multi_index* with only one iterator increases execution time and complexity exponentially. This shows how much iteration affects performance within GA scripts.

Multi-dimensional Continuous Optimisation Problems (Subtasks 1.B/1.C)

Multi-dimensional continuous optimisation problems are most effectively solved with GA's (in comparison to lower dimensional problems) due to the crossover functionality. Three significantly different multi-dimensional problems have been tested with the GA methods: The Sum Squares Function; The Shwefel Function and The Michalewicz Function.

The Sum Squares Function

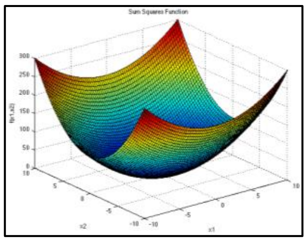


Figure 4

The implementation of this problem can be found under *ga-continuous-distrib/sum-squares.py*. ***Figure.4***[3] shows a two dimensional plot of the function which shows it's a convex 'bowl' type function with one global minima. Due to this and said function being multi-dimensional, the GA methods are likely to be effective in finding a solution. ***Figure.5***[4] shows the equation for this function, which is generally evaluated (as well as during experiments) with the bounds: *xi* ∈ *[-10, 10]*, for all of *i = (1, …, d)*. (Note that sign omission is used during experimentation to get exact solutions, however performance correlates exactly with signed function)

$$f(\mathbf{x}) = \sum_{i=1}^{d} i x_i^2$$

Figure 5

Appendix 2.1 shows fitness against generations with the basic methods run with a population of 10 for 1000 generations. Threads 3 and 5 take more than 200 generations to achieve $f(x) \leq 1$, and only thread 0 achieves $f(x) \leq 0.1$. Most of the threads make no further improvement after 600 generations, which is to be expected since with only 10 individuals and a mutation rate of 0.2 (with 1 mutation), only two mutations on average will happen per generation making it unlikely a gene will be improved. With an increase to just 100 population, a huge improvement can be seen in ***Appendix 2.2*** where most of the thread achieve a fitness of $f(x) \leq 0.1$ by 400 generations, and some threads even achieve $f(x) \leq 0.01$. When the population is increased to 1000 as shown in ***Appendix 2.3***, the fitness achieved starts to get close to a solution, with three threads at $f(x) \leq 0.001$ after 600 generations. With a population of 10,000 shown in ***Appendix 2.4*** all threads achieve $f(x) \leq 0.001$ within 600 generations, and after 800 generations 5/6 of the threads yield a fitness value very close to a solution. Since (as shown in ***Figure.3***) the execution time for 10,000 generations is approximately one minute per run (depending on other methods), this value can be thought of as being optimal for the basic methods, as any larger amount would yield diminishing results.

Appendix 3 shows the *boundary_mutation(…)* methods with a population of 10,000. It yields very similar results to the standard *uniform_mutation(…)* as to be expected, however it is slightly less effective as the fitness is much more likely to plateau due to there being less variation in the mutations. Because of this, only thread 4 reaches a fitness value close to a solution. The most effective extra method as expected is *non_uniform_mutation(…)*, since the mutation bounds get closer to the gene values as average fitness improves, vastly improving the effectiveness of mutations. Due to the method working with the average fitness across the population, it is possible that effectiveness is reduced as population size gets large.

POPULATION_SIZE	10	20	30	50	100	150	200	300	500	1000	2000	5000	10000
Average Generations	962	842	706	595	554	466	441	482	467	418	447	494	531

Figure 6

Figure.6 shows the average number of generations (with 50 threads) against different population sizes for a maximum of 1000 generations using the *non_uniform_mutation(…)* method. Despite 1000 population having the lowest number of average generations, the population size range $150 \leq n \leq 200$ generally includes lower minimum values (some threads as low as ≈ 250) albeit less consistently (resulting in a higher average). Therefore henceforth this method will be mostly tested within said range, as each thread will have higher chance of taking a minimal amount of generations, despite taking a higher amount of generations on average. Note ***Figure.6*** also includes generation values of 1000 within the averages for threads with no solution, resulting in average skewing in comparison to below figures.

Appendix 4.2 shows the *selection_tournament(…)* method (with default *tournament_size*) in replace of the basic selection method *selection_roulette(…)*, for fitness values in the range $f(x) \leq 1$. Despite also using the optimal *non_uniform_mutation(…)* mutation method, no solution is found in any threads within 1000 generations, as opposed to the roulette selection that finds a soliton in ≈ 300 generations (shown in ***Appendix 4.1)***. Even when *tournament_size* is increased to 50% and 70% of the population despite having improvements in terms of less variation, still do not find a solution within 1000 generations. Therefore it can be assumed tournament selection is much less effective than roulette wheel selection for continuous optimisation GAs. What is interesting however is that the reduction in fitness is more gradual than the other selection methods, most likely due to the fact that each tournament wont necessarily contain one of the fitter individuals, resulting in unfit individuals being selected as parents more often. ***Appendix 4.3*** shows the *selection_rank(…)* method in place of roulette selection which is much more effective as is expected. The number of generations until a solution with this method typically is with the range $100 \leq n \leq 150$, and sometimes as low as ≈ 90. The communication method *selection_roulette_rank(…)* is slightly more effective, however its improvement is negligible as the main improving factor is *selection_rank(…).*

Despite the increased variation in offspring when using the *single_point_crossover_multi_index(…)* method over the basic *single_point_crossover_opt(…)* method, there is no significant improvement in either the minimal generations until solution, or the average range seen. However, improvement is seen with the *double_point_crossover_opt(…)* method, with the generations until solution range typically being within $80 \leq n \leq 120$, and sometimes even less as seen in ***Appendix 5*** with a solution at 79 generation (thread 1). An interesting feature also observed with this method is that its fitness reduction appears to happen in jumps (as seen comparing ***Appendix 5*** and ***Appendix 4.3***).

After the above experimentation on *sum-squares.py*, it can be assumed the optimal selection, crossover and mutation methods are: *selection_roulette_rank(…), double_point_crossover_opt(…)* and *non_uniform_mutation(…)*, which will henceforth be used in other experiments as the 'optimal' methods.

INDIVIDUAL_SIZE	2	3	4	5	6	7	8	9	10	11	12	13	15	20	30	40	50	100
Average Gens	84	114	130	157	132	183	135	149	158	192	214	271	288	260	281	243	312	450

Figure 7

Figure.7 shows experimentation of the *INDIVIDUAL_SIZE* (*d*) parameter with *single_point_crossover_opt(…)* and the optimal methods (and other hyperparameters set to default), with 50 threads. Since sum squares is a fairly simple optimisation problem, it is expected that $2 \leq d \leq 4$ performs quite well despite the lack of effectiveness with crossover. For $5 \leq d \leq 10$ the average generations until solution is fairly consistent, this range can be considered optimal for the GA methods. When $d > 10$ the average generations until solution increases and becomes less consistent, although a solution is always still found within 100 generations. Therefore it can be assumed the GA is still effective with a large amount genes.

Parameter Value	1	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1	0
CROSSOVER_RATE Avg Gens	169	170	158	235	269	337	328	451	625	712	1000
MUTATION_RATE Avg Gens	86	101	130	113	118	120	157	142	158	273	1000

Figure 8

Figure.8 shows experimentation around the *CROSSOVER_RATE* and *MUTATION_RATE* with the same control parameters/methods as in ***Figure.7***. The data is also visualised in ***Appendix 6*** using *smoothingspline* curve fitting in MatLab with parameter values (*Δr*) $0.1 \leq \Delta r \leq 1$. The change in *CROSSOVER_RATE* (*Δcr*) appears to affect the generations until solution as expected, with $0.8 \leq \Delta cr \leq 1$ being the optimal range, and 0.8 being the absolute optimal value. With a value of 0, a solution will never be found as no offspring will be generated, resulting in absolutely no variation between individuals in future generations. The change in *MUTATION_RATE* (*Δmr*), however has quite a surprising impact on the generations until solution is found. At *MUTATION_RATE = 1* there is a huge improvement on the best thread results, with some threads achieving as low as ≈ 40 generations until solution is found, however there is greater inconsistency with a lot of threads not finding a solution within the 1000 generations. This is likely due to non-mutated children also being evaluated during survival. However, since the experiment was carried out on the sum squares problem which is relatively simple, these results are unlikely to be achieved with more difficult problems. With a value of 0, the GA quickly finds the optimal children from the original population but no further improvement/variation can be made resulting in no solutions. As shown by ***Appendix 6.1***, the *CROSSOVER_RATE* has a much larger effect on the performance of the GA than the *MUTATION_RATE.*

MUTATIONS	0	1	2	3	4	5	6	7	8	9	10
Average Gens	1000	155	362	577	633	658	719	738	719	725	740

Figure 9

Figure.9 shows the *MUTATIONS* parameters impact on the average amount of generations until a solution is found for 50 threads with other default control variables and optimal methods (with the data visualised using MatLab *smoothingspline* curve fitting in ***Appendix 6.2***). With 0 mutations, as expected the GA cannot improve individuals beyond the original optimal children (the same behaviour as with ***Figure.8*** *MUTATION_RATE = 0*). It is also expected that 1 mutation (which is default) is optimal as with increased mutation more randomness is introduced decreasing the GA's effectiveness. This can be seen in ***Appendix 6.2,*** with the range $2 \leq MUTATIONS \leq 6$ (up to ≈ 50% of the individuals genes mutated) where the average generations until a solution quickly rises to a maximum of ≈ 750 generations at $MUTATIONS > 7$. These results suggest that a minimal amount of randomness should be introduced into GA's, just enough so that individuals do not get stuck at locally optimal genes.

The Schwefel Function

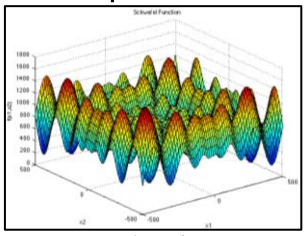


Figure 10

The implementation of this function can be found under *ga-continuous-distrib/schwefel.py*. ***Figure.10***[5] shows a two dimensional plot of the function. It is a very complex multi-dimensional function with many irregularly distributed local minima, which could cause the GA to be less effective in comparison to the sum squares function. ***Figure.11***[6] shows the equation for this function, which is generally evaluated (as well as during experiments) with the bounds: $x_i \in [-500, 500]$, for all of $i = (1, \ldots, d)$ There is a global minima at $f(x) = 0$, with $x = (420.9687, \ldots, 420.9687)$.

$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$$

Figure 11

Appendix 7 shows the optimal methods with *MUTATION_RATE = 0.8* ran on this function in the fitness range $f(x) \leq 10$. Whilst thread 3 finds a solution relatively quickly at 211 generations, most of the other threads gets stuck at local minima such as thread 5 stuck at $f(x) \approx 1.49$ and thread 4 struck at $f(x) \approx 1.2$. These results show the decrease in performance of GAs for more difficult optimisation functions in comparison to more simple functions such as sum squares, where even with the basic methods all threads achieve $f(x) \leq 1$ within 200 generations (see ***Appendix 2.2***). If further generations are run however, all threads will eventually converge at the global minima and produce solutions from what is observed during experimentation. It is also interesting to note that higher values for *MUTATION_RATE* also produce better results as with sum squares, with $MUTATION\_RATE \geq 0.7$ generally producing a generations until solution range of $200 \leq n \leq 400$ whereas $MUTATION\_RATE \leq 0.7$ produced a range of $400 \leq n \leq 800$.

The Michalewicz Function

$$f(\mathbf{x}) = -\sum_{i=1}^{d} \sin(x_i) \sin^{2m}\left(\frac{ix_i^2}{\pi}\right)$$

Figure 13

The implementation of this function can be found under *ga-continuous-distrib/michalewicz.py*. ***Figure.12***[7] shows a two dimensional plot of the function. The function has multiple local minima within 'valley' like sections, with one sharp drop at the global minima. ***Figure.13***[8] shows the equation for this function, which is generally evaluated (as well as during experiments) with the bounds: $x_i \in [0, \pi]$, for all of $i = (1, \ldots, d)$. The number of local minima is proportional to number of dimensions by $!d$, which means a solution is more difficult to find with higher amounts of dimensions. There are three known global minima found at: $d = 2$, $f(x) = -1.8013$, $x = (2.20, 1.57)$; $d = 5$, $f(x) = -4.687658$ and $d = 10$, $f(x) = -9.66015$. The parameter m can also be used to adjust the 'steepness' of the valleys which is default 10, with a higher value resulting in a more difficult search.

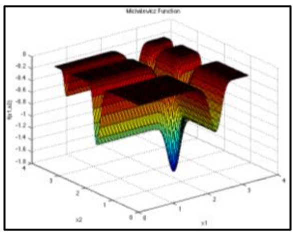


Figure 12

When experimenting with 10 dimensions and with $m = 10$, the GA seems to have a huge drop in performance in comparison to the previous multi-dimensional functions. Due to the shape and amount of local minima (for $d = 10$, local minima $= 10! = 3628800$) the *non_uniform_mutation(…)* method is much more likely to have individuals stuck at local minima which will make no further improvement in future generations at lower population sizes (can be visualised with ***Figure.12*** as individuals stuck in those local 'valleys' have no way of getting out). This is observed in ***Appendix 8.1*** where the method is run with *POPULATION_SIZE = 1000* for 4000 generations where half of the threads get stuck in the $f(x) \leq 0.1$ fitness range. Therefore contrary to the previous experiments with different optimisation functions, for this problem the basic *uniform_mutation(…)* method is possibly more effective in terms of increasing fitness in threads more frequently for lower population sizes (with an exact solution unlikely to be found). This is observed in ***Appendix 8.2*** which shows that with said basic function, all threads achieve a fitness of $f(x) \leq 0.0001$ within ≈ 1500 generations, with values very close to a solution appearing at ≈ 3500 generations.

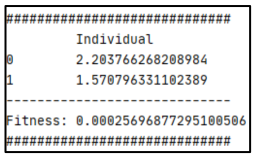
However a solution can be found using the *non_uniform_mutation(…)* method when a favourable starting population is generated. This is where scalability of the population size becomes a useful feature of this GA's implementation. With 1000 population a solution is found in ≈ *10%* of the fresh populations generated, whereas when population size is increased to 10,000 every population tested was able to find a solution. Its also important to note that of those population where a solution was achieved it was generally found in < *100* generations.

For *INDIVIDUAL_SIZE* (d), $d = 5$ and $d = 2$ there are only $5! = 120$ and $2! = 2$ local minima respectively for the which the GA finds solutions very consistently in a low amount of generations, even with only 1000 population.

m	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
d = 2	37	38	38	38	38	38	35	37	38	33	29
d = 5	59	54	54	52	52	51	50	50	50	49	53
d = 10	69	63	67	67	68	70	71	75	77	79	88

Figure 14

Figure.14 shows experimentation around the *STEEPNESS* (m) parameter with different dimension sizes average generations until a solution found values. Optimal methods with *single_point_crossover_opt(…)* and *MUTATION_RATE = 0.8* are used in each experiment, as well as a population size of 10,000 with 50 threads for consistency. The data is visualised using MatLab *smoothingspline* curve fitting in ***Appendix 8.3***. The first interesting observation is the fact that for each value of d, when m is set to 0 the generations until solution found is generally greater than that of larger m values. With $d = 10$ the results are as expected (other than anomaly at $m = 0$) with the generations until solution found increasing almost linearly with $1 \leq m \leq 6$, with a sharper more exponential rise after that until $m = 10$. With $d = 5$ the results are somewhat surprising, as the generations until solution found steadily decreased with $1 \leq m \leq 9$ with a small rise at $m = 10$. The results for $d = 2$ however are the most strange and surprising, as the generations until solution found is essentially constant with $1 \leq m \leq 8$ (other than $m = 6$ with a slight decline) with a somewhat exponential decline with $9 \leq m \leq 10$. Comparing the three dimension values in ***Appendix 8.3***, $d = 2$ *and* $d = 10$ almost appear to have opposite behaviour, with $d = 5$ having somewhat of an average of the two behaviours. This could be due to the fact that with higher steepness values the global/local minima are further apart (as can be seen in ***Figure.12***) which in combination with the low amount of local minima when testing lower dimensions could result in an easier search. However, this theory is untested and much deeper experimentation would need to be carried out to understand the actual cause of this behaviour. When testing $m > 10$ the problem becomes exponentially more difficult to find a solution, and possibly even impossible to find an exact solution. Even after running the GA with $d = 2$ with a population of 10,000 for 100,000 generations, an exact solution was not found (although values somewhat close to a solution are achieved as seen in ***Figure.15***).



```
##############################
             Individual
0            2.203766268208984
1            1.570796331102389
------------------------------
Fitness: 0.00025696877295100506
##############################
```

Figure 15

Two dimensional Continuous Optimisation Problems (Subtasks 1.B/1.C)

Due to two dimensional continuous optimisation problems only working with two genes per individual, the crossover feature of GAs become less effective and offspring produced are less varied. Because of this, it can be thought that such continuous optimisation problems are generally solved less effectively than that of previously tested optimisation problems. Three significantly different two dimensional problems have been tested with the GA methods: The Matyas Function; The Bukin N.6 Function and The Six-Hump Camel Function. It is also important to note that for all of the two dimensional functions only single point crossover can be used.

The Matyas Function

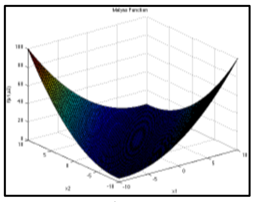


Figure 16

The implementation of this function can be found under *ga-continuous-distrib/matyas.py* with ***Figure.16***[9] showing a visualization of the function. The function is rather simple in nature, being 'plate' shaped with a wide valley containing its single global minima in the middle at 0. ***Figure.17***[10] shows the equation for the function which is generally evaluated (as well as during experiments) with the bounds: $x_i \in [-10, 10]$, for all of $i = (1, 2)$. The global minima $f(x) = 0$ can only be found at $x = (0,0)$.

$$f(\mathbf{x}) = 0.26(x_1^2 + x_2^2) - 0.48x_1x_2$$

Figure 17

Appendix 9.1 shows the function ran with optimal methods as well as with *MUTATION_RATE = 0.8* and *POPULATION_SIZE = 1000* for 10,000 generations in the fitness range $f(x) \leq 0.0001$. The *non_uniform_mutation(…)* method is shown to be ineffective against this function, as after a certain point improvement in fitness becomes very insignificant each subsequent generation. The *uniform_mutation(…)* method is somewhat more effective as seen with ***Appendix 9.2*** where with this method 6 threads achieve a fitness of $f(x) \leq 1*10^{-7}$ within 1000 generations which can be thought of as being very close to a solution. This is likely due to the fact fitness is calculated using *float64* precision meaning it would take a large amount of generations to converge on the exact solution. However in theory if the GA were to run for long enough, an exact solution would be found.

The Bukin N.6 Function

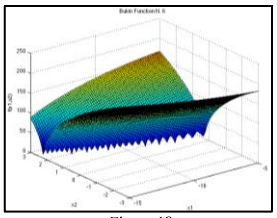


Figure 18

$$f(\mathbf{x}) = 100\sqrt{\left|x_2 - 0.01x_1^2\right|} + 0.01\left|x_1 + 10\right|$$
Figure 19

The implementation of this function can be found under *ga-continuous-distrib/bukin.py*. ***Figure.18***[11] shows a visualization of this function. It is a rather complex function with many local minima that all lie within a ridge. The local minima are somewhat regularly distributed. Due to this and the fact that the input domain only contains two dimensions this optimisation problem could prove difficult for the GA to produce solutions. ***Figure.19***[12] shows the equation for this function which is generally evaluated (as well as during experiments) with the bounds: *x1∈[-15, -5]* and *x2∈[-3, 3]*, for which the global minima can be found at *f(x) = 0, x = (-10,1)*. Due to this GA's methods implementation, *MUTATIONS* must be set to 2 if using separate bounds for each *x* value, however a single bound range of *∈[-15, 3]* can be used in combination with *MUTATIONS* set to 1.

Appendix 10 shows the function ran with single bounds, *MUTATION_RATE = 1*, *POPULATION_SIZE = 1000* and *uniform_mutation(…)* for 50,000 generations with 10 threads. No solutions or any values close are found with all threads getting struck at local minima with only 6/10 of them achieving a fitness of *f(x) ≤ 0.1*. These results show how difficult it is for the GA to find solutions when crossover loses effectiveness on problems with many local minima. It is possibly that with separate gene bounds if ran for enough generations a solution could be found, however this essentially would just be a random process meaning it would be highly unlikely (and unreproducible). Therefore it can be assumed that GA's are very ineffective in solving two dimensional continuous optimisation problems with many local minima.

The Six-Hump Camel Function

$$f(\mathbf{x}) = \left(4 - 2.1x_1^2 + \frac{x_1^4}{3}\right)x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2$$
Figure 21

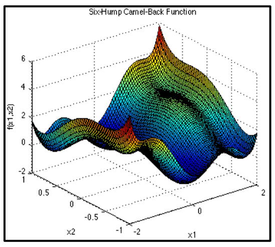
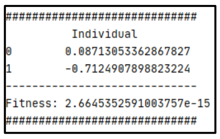


Figure 20

The implementation of this function can be found under *ga-continuous-distrib/six-hump-camel.py*. ***Figure.20***[13] shows a visualisation of a section of the function. There are 6 local minima, two of which are global. The minima are located in gradual dips unlike the previously tested function, which could result in an easier search for the GA. ***Figure.21***[14] shows the equation for this function, which is generally evaluated (as well as during experiments) with the bounds: *x1∈[-3, 3]* and *x2∈[-2, 2]*, for which the global minima can be found at *f(x) = -1.0316, x = (0.0898, -0.7126)* and *(-0.0898, 0.7126)*. The same restriction on gene bounds and *MUTATIONS* with the Bukin N.6 function also apply here.



```
##############################
          Individual
0        0.08713053362867827
1       -0.7124907898823224
------------------------------
Fitness: 2.6645352591003757e-15
##############################
```

Figure 22

Appendix 11 shows a run of this function with separate gene bounds and with a low amount of maximum generations (1000), for which some threads get stuck at local minima (for example thread 5 gets stuck at approximately generations (*n*) *200 ≤ n ≤ 800*) for a while but eventually converge to the global minima within 1000 generations. These results show the GA is much more effective at solving this function in comparison to the previously tested Bukin N.6 function despite both having local minima (although Six-Hump Camel is considerably less complex with few local minima). However, the GA suffers from the same issue with The Matyas function when attempting to find an exact solution with the *non_uniform_mutation(…)* method unable to find an exact solution within 100,000, despite being very close as shown in ***Figure.22***.

Analysis (Subtask 1.D)

The above experimentation has shown that GAs are very effective in solving continuous optimisation problems. Problems with a large number of dimensions are most effectively solved with GAs due to the crossover functionality. Two dimensional problems can still be solved but not as effectively, depending on the amount of local minima contained within the problem. Another evolutionary algorithm technique, evolutionary programming, could potentially produce opposing results being more effective on lower dimensional problems and less effective on higher dimensional problems (in comparison to GAs) since there is no crossover but an emphasis on mutation. Evolutionary strategy could potentially be more effective solving such continuous optimisation problems, however the recombination operation could decrease its efficacy since as stated previously, the crossover search functionality is really what makes such algorithms effective in solving these problems.

Combinatorial Optimisation

Efficacy of combinatorial optimisation problems are generally dependent on the problem as they have quite a varied domain. Unlike with continuous optimisation problems where generally only the fitness function needs to be altered per problem, problem specific alterations to the GA methods generally have to be made with combinatorial optimisation problems. Five combinatorial optimisation problems have been implemented within separate discrete domains: The Sum Binary Problem; The Partition Problem; The Phrase Matching Problem; The N-Queens Problem and the Knapsack Problem.

The Sum Binary Problem (Subtask 2.A)

$$f(\mathbf{x}) = \sum_{i=1}^{n} x_i,$$
Figure 23

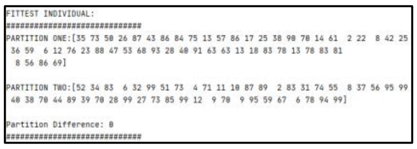
The implementation of this function can be found under *ga-combinatorial-distrib/sum-binary.py*. ***Figure.23*** shows the equation for this problem, which takes input from the discrete set xi *∈[1, 0]*. Maximization of the function happens at *f(x) = nxi* whereas minimisation happens at *f(x) = 0*. Both minimisation and maximisation fitness functions are implemented, however, experimentation will be only carried out with minimisation. Experimentation is also carried out with default *CROSSOVER_RATE, MUTATION_RATE* and *MUTATIONS.*

INDIVIDUAL_SIZE	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
Average Generations	4	8	13	21	26	35	39	50	57	65

Figure 24

The problem is solved very effectively by the GA even with the basic methods with solutions being found in *< 5* generations consistently with and *INDIVIDUAL_SIZE* of 10 and with 100 population. If the population is increased to 1000, a solution is always found in 2 generations. ***Figure.24*** shows the average generations until solution is found as *INDIVIDUAL_SIZE* is increased with 100 population for an average of 50 threads (with optimal selection and crossover methods), which shows average generations until solution found increases somewhat linearly with the number of dimensions. ***Appendix 12*** shows the function ran with both 1000 population and *INDIVIDUAL_SIZE* with 6 threads. What is interesting is that all threads follow a very similar path to the solution, with a sharp linear like decline in fitness within approximately 50 generations, with a somewhat inversely proportional decline thereafter until a solution is found. This shape is always produced with every run, which is most likely due to the fact there is little variation when genes can only take two discrete values.

The Partition Problem (Subtask 2.B)



```
FITTEST INDIVIDUAL:
##############################
PARTITION ONE:[35 73 50 26 87 43 86 84 75 13 57 86 17 25 38 90 70 14 61  2 22  8 42 25
 36 59  6 12 76 23 88 47 53 68 93 28 40 91 63 63 13 18 83 78 13 78 83 81
  8 56 86 69]

PARTITION TWO:[52 34 83  6 32 99 51 73  4 71 11 10 87 89  2 83 31 74 55  8 37 56 95 99
 40 38 70 44 89 39 70 28 99 27 73 85 99 12  9 70  9 95 59 67  6 78 94 99]

Partition Difference: 0
##############################
```

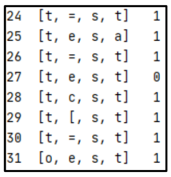
Figure 26

$$\begin{aligned} a_1x_1 + \ldots + a_nx_n &\le (a_1 + \cdots + a_n)/2 + k \\ a_1x_1 + \ldots + a_nx_n &\ge (a_1 + \cdots + a_n)/2 - k \\ x_1, \ldots, x_n &\in \{0, 1\} \end{aligned}$$
Figure 25

The implementation of this problem can be found under *ga-combinatorial-distrib/partition.py*. ***Figure.25***[15] shows the equation for this problem which takes input from the discrete set xi *∈[1, 0]*. A list of integers *a* is generated with the objective being to separate the list into two partitions so that the difference between the sums of each partition is minimal. At exact solution would be if both partitions would sum to the same value, however for some sets of *a* this isn't possible. Experimentation is done with basic methods and default hyperparameters.

Figure.26 shows an example exact solution with 100 integers in the list (with each integer being within the bounds *1 ≤ a ≤ 100*) with only population size of 10, which was found in 28 generations. With larger population sizes the GA becomes exponentially more effective, with a solution always being found within 2 generations at 1000 population. This is very similar to behaviour observed with The Sum Binary problem, which is as expected as once again, the genes can only take two discrete values (either being in partition one or partition two). What is interesting however is that the number of dimensions does not seem to have a considerable effect on the GA's performance, as is seen in ***Appendix 13*** which has an integer list size of 100,000 with a solution being found in 26 generations (again with only 10 population). In contrary to The Sum Binary problem, threads take somewhat different solution routes whilst also being somewhat more inversely proportional.

The Phrase Matching Problem (Subtask 2.B)



```
24  [t, =, s, t]   1
25  [t, e, s, a]   1
26  [t, =, s, t]   1
27  [t, e, s, t]   0
28  [t, c, s, t]   1
29  [t, [, s, t]   1
30  [t, =, s, t]   1
31  [o, e, s, t]   1
```

Figure 27

The implementation of this problem can be found under *ga-combinatorial-distrib/phrase-match.py*. The Phrase Matching Problem can be defined as a population of random string individuals being optimised in regards to matching a defined phrase. This solution randomises each character in the string using Pythons *string.printable* constant that contains every possible interpretable character for the language. ***Figure.27*** shows a sample of individuals when matching the phrase *'test'* using default parameters and basic methods with a population of 100. With only four characters a solution is found relatively fast (in the case of ***Figure.27*** in 15 generations).

Appendix 14.1 shows the respective average fitness over generations plot for ***Figure.27*** which takes an opposite solution path than the previous problems. However, when the phrase length is increased, generations until solutions found increases exponentially, as shown in ***Appendix 14.2*** with the phrase *'this is a longer test with s0me punctuation!!%**'* tested that is solved in 1276 generations. Also note that the solution takes a more standard path similar to the previous problems, but with small 'jumps' in fitness reduction. Although its important to note that due to Numpy optimisations within the GA (and phrase match fitness) functions solutions are generally found within 1 second. Even the relatively large phrase: *'this is aasdasdddddddddd longfasfasfer test asfasfaswitasddddddddh s0me punctuatasfasffffffff fffffion!!%**testtestest1(8tesasd asdasda124%%$3t'* which took 6226 generations to match was solved in only 7.1 seconds.

The N Queens Problem (Subtask 2.B)

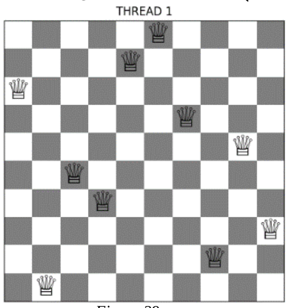
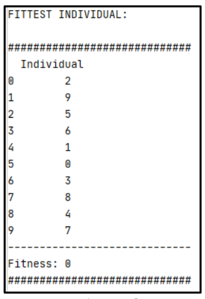


Figure 29

The implementation of this problem can be found under *ga-combinatorial-distrib/n-queen.py*. The problem is defined as placing *n* 'queens' on an *n*n* chessboard so that each queen has no other queens either horizontally or vertically within its path. This is implemented by generating individuals of size *n* with each gene being a random integer in the range of *n*. The index of each gene represents the queens *x* coordinate and the gene value represents the *y* value. This allows for a 1D representation for each individual for the actual 2D chessboard representation. The fitness function counts the number of occurrences of each gene value within each individual, and sums the number of single occurrences, which is then subtracted from *n*. This is done using *np.bincount* in order to minimize execution time. This results in the fitness value representing the number queens on unique rows, with *f(x) = 0* being a solution to the problem.



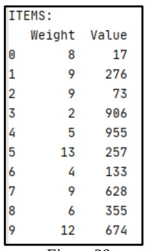
```
FITTEST INDIVIDUAL:

##############################
  Individual
0         2
1         9
2         5
3         6
4         1
5         0
6         3
7         8
8         4
9         7
------------------------------
Fitness: 0
##############################
```

Figure 28

Figure.28 shows a solution with 10 queens (with 100 population) that was achieved in 5 generations within 0.23 seconds. This is visualised using the *plot_chessboard(…)* method from *ga-combinatorial-distrib/common.py* which is seen in ***Figure.29***. In comparison to the traditional methods of solving this problem such as backtracking, the GA solution is much more efficient in terms of both iterations and execution time. The GA approach is also scalable with large amounts of queens, as shown in ***Appendix 15***, which solves the problem for 100 queens in 200 generations within 10.2 seconds. Its also interesting to note that for this problem threads take a very similar solution route despite said solution route being somewhat irregular.

The Knapsack Problem (Subtask 3.A)

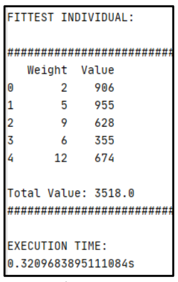


```
ITEMS:
   Weight  Value
0       8     17
1       9    276
2       9     73
3       2    906
4       5    955
5      13    257
6       4    133
7       9    628
8       6    355
9      12    674
```

Figure 30

The implementation of this problem can be found under *ga-combinatorial-distrib/knapsack.py*. The Knapsack Problem can be considered a 'real world' combinatorial optimisation problem. The problem is defined as having a 'sack' of fixed threshold weight (defined with the parameter *KNAPSACK_MAX_WEIGHT*) and a set of items each assigned a weight and value, with the solution being a subset of said items where the total value is maximised, but the total weight does not exceed the threshold weight of the sack.

In order to solve this problem with the GA, a binary population of individual size equal the total number of items is generated (done using *generate_binary_population(…)*), for which item values equal to the indexes of bit values of 1 are used to select items from the list. The fitness of each individual is calculated as the sum of the values of each item the individual selects, subject to the sum of weights of each item the individual selects being ≤ *KNAPSACK_MAX_WEIGHT* (for which the fitness will be set to 0 if said threshold is breached). The fitness value is then inverted in order to work with the minimisation functions. Since this is a maximisation problem and the since items values/weights are randomised on each run, the exact solution is unknown for each run and therefore cannot be used as a termination condition.



```
FITTEST INDIVIDUAL:

##########################
   Weight  Value
0       2    906
1       5    955
2       9    628
3       6    355
4      12    674

Total Value: 3518.0
##########################

EXECUTION TIME:
0.3209683895111084s
```

Figure 31

Figure.30 shows an example item set of 10 items, for which an assumed solution is shown in ***Figure.31***. ***Appendix 16*** shows the generations over fitness plot for this run. Within ≈ 10 generations the solution path of all threads is almost identical. The threads diverge slightly for ≈ 20 generations after before all threads converge at 3518 fitness within 40 generations.

Analysis

The above experimentation has shown the versatility of GAs in solving different combinatorial optimisation problems within different domains. Generally the GA is quite effective in solving such problems, however, there is sometimes difficultly converting these problems into a format that is workable with the GA.

Optimizing Neural Networks With GAs (Subtask 3.B)

One of the main issues surrounding neural networks (NN) is the fact that correct hyperparameters (including: number of hidden layers; learning rate; momentum; activation function; minibatch size; epochs and dropout)[16] have to be known prior to training said NN. However, with the use of GAs, optimal hyperparameters can be found prior training, vastly increasing the effectiveness of the NN.

How NN Optimisation Through GAs Is Accomplished

The first step would be to generate a population with the individuals being a set of NNs with randomly assigned hyperparameters. Fitness would be calculated through training each NN individual, for which a 'training cost' is associated to each individual. The typical GA selection methods described throughout this report can then be applied, with the lowest cost NNs having a higher chance of being selected for crossover. Again, typical crossover methods can be applied to the individuals to produce offspring, for which mutation can occur by randomising a selected hyperparameter. This will eventually result in optimal hyperparameters being generated.

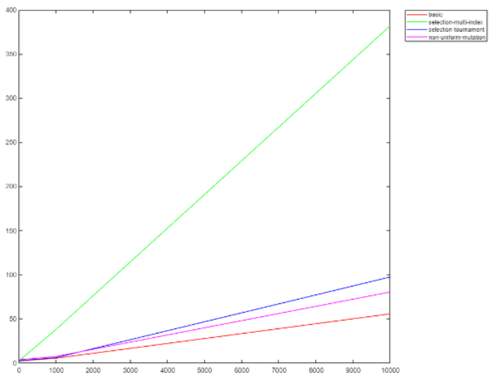
Advantages

The obvious advantage of using this method of NN hyperparameter optimisation is that manual adjustment of said hyperparameters no longer needs to be carried out which is generally tedious and time consuming. It is also likely that the hyperparameters calculated through the GA will be much more optimal than that of manual adjustment, simply due to the fact that GAs can test multiple solutions simultaneously and sequentially much faster than what can be done manually, within a reasonable time frame. Another advantage is the breadth of possible solution routes the GA can take, resulting in interesting or potentially previously unthought of hyperparameter solutions. The solutions provided by GA's are also easily analysable allowing for further insight into the NNs and in how the GA came to said hyperparameter solution.

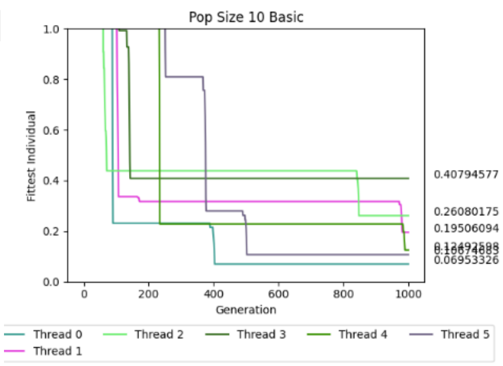
Disadvantages

The main disadvantage of GA NN hyperparameter optimisation is the execution time and computational resources required. Since each individual in the population will require a training run of the NN during fitness calculation, execution time can increase exponentially with population size. Also, parallelisation of such GAs would require significant compute in order to be effective. The typical disadvantages associated with GAs also apply here.

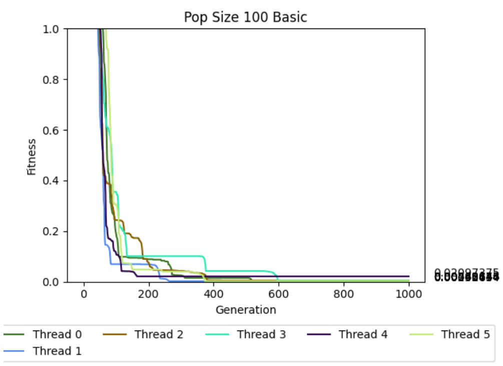
Appendix



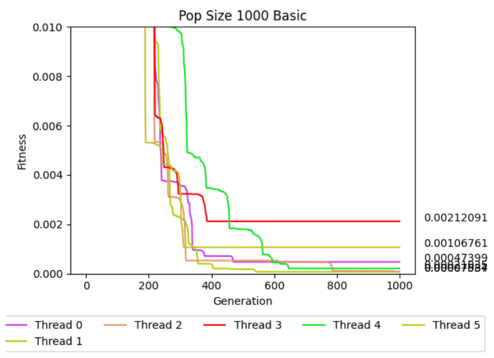
Appendix 1



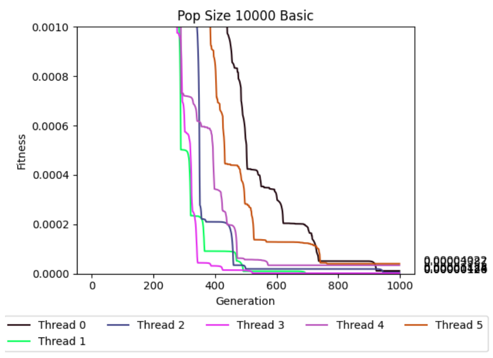
Appendix 2.1



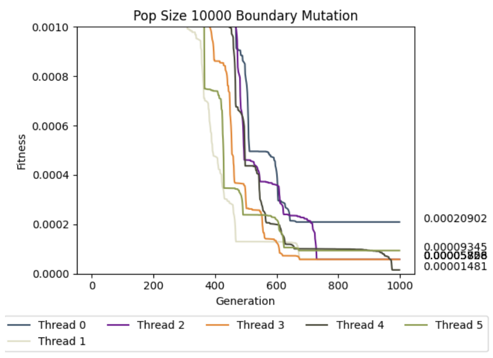
Appendix 2.2



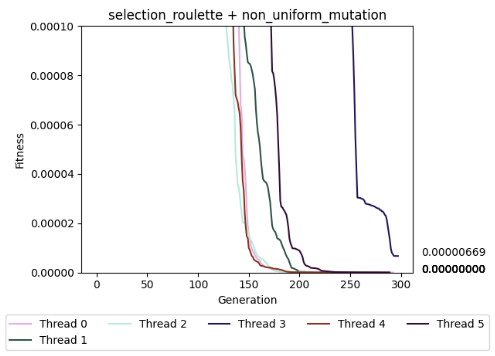
Appendix 2.3



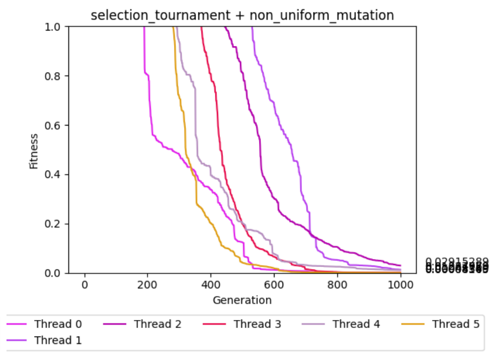
Appendix 2.4



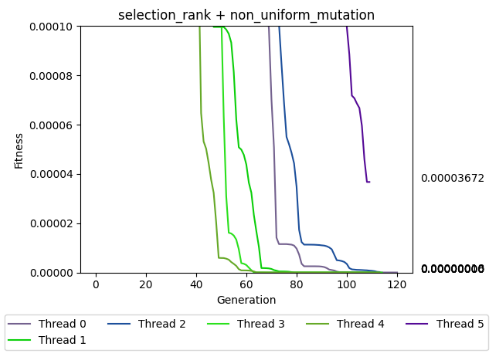
Appendix 3



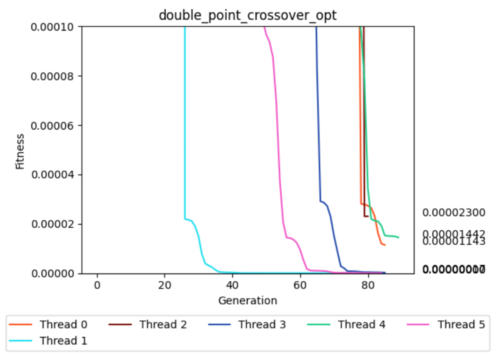
Appendix 4.1



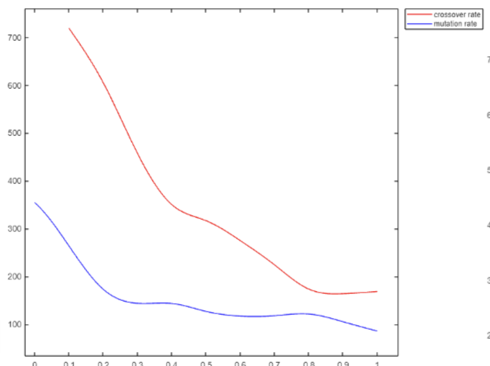
Appendix 4.2



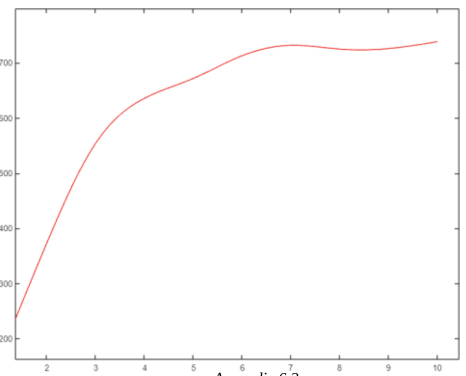
Appendix 4.3

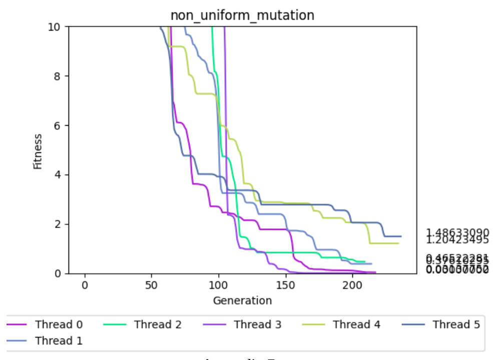


Appendix 5

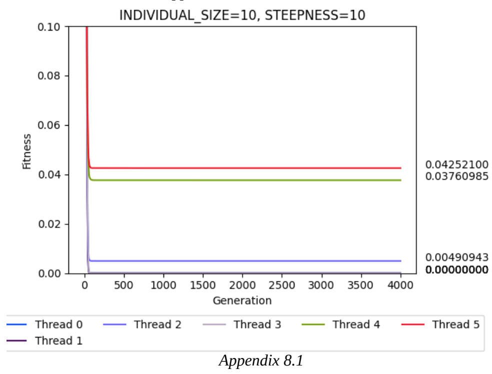


Appendix 6.1

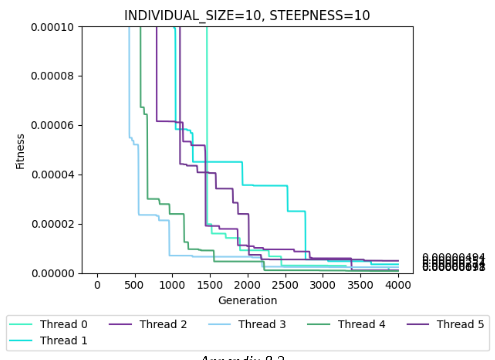

Appendix 6.2



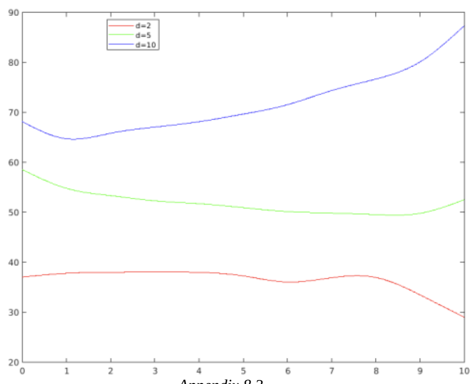
Appendix 7



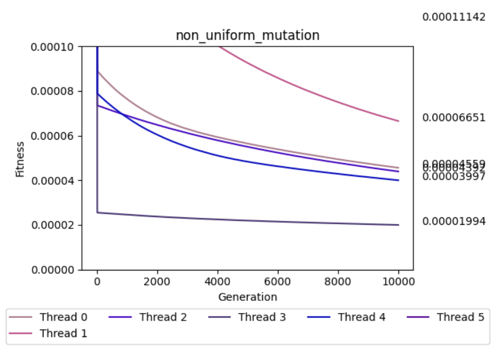
Appendix 8.1



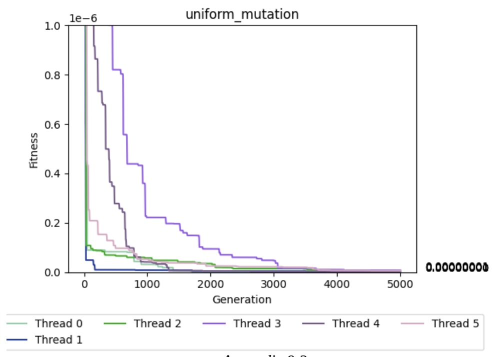
Appendix 8.2



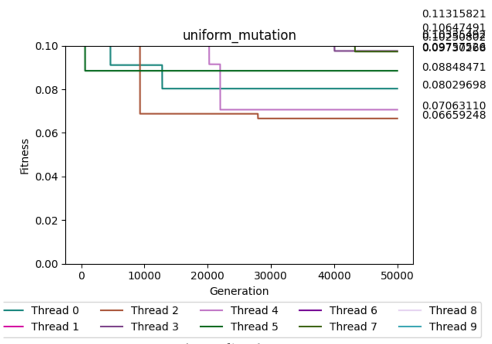
Appendix 8.3



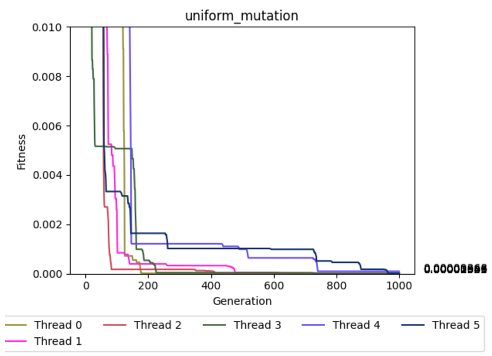
Appendix 9.1



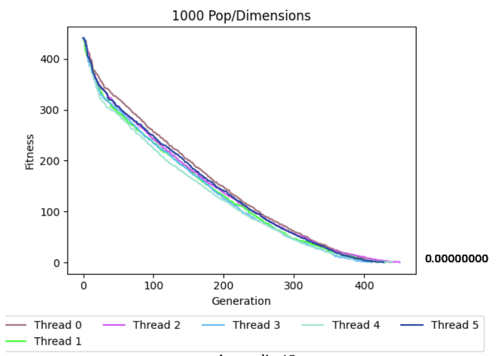
Appendix 9.2



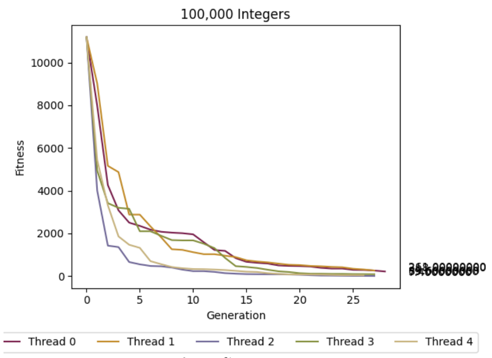
Appendix 10



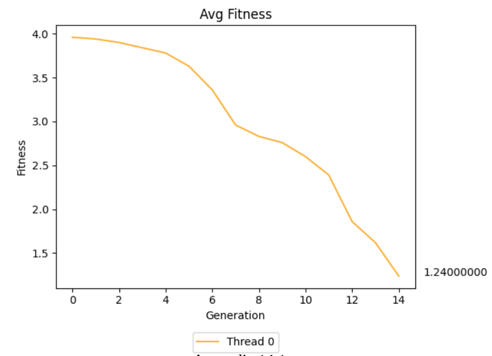
Appendix 11



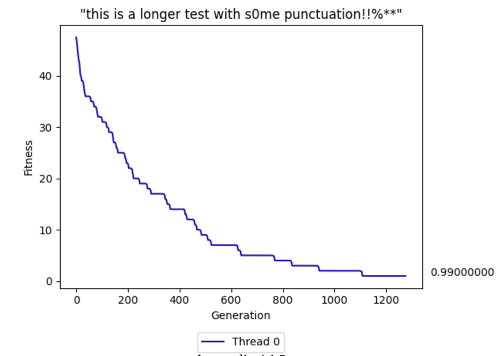
Appendix 12



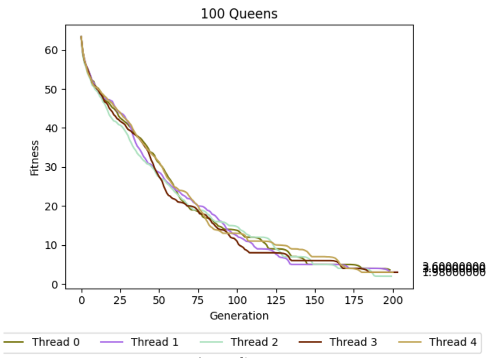
Appendix 13



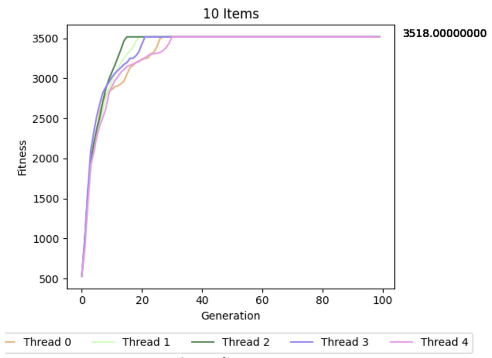
Appendix 14.1



Appendix 14.2



Appendix 15



Appendix 16

Bibliography

[1] Dutta, A. (2019, June 21). Crossover in Genetic Algorithm. Geeks For Geeks. https://www.geeksforgeeks.org/crossover-in-genetic-algorithm. Accessed (2021, Feb 28).

[2] Dutta, A. (2019, June 21). Crossover in Genetic Algorithm. Geeks For Geeks. https://www.geeksforgeeks.org/crossover-in-genetic-algorithm. Accessed (2021, Feb 28).

[3] Surjanovic, S., Bingham, D., & Fraser, S. (2013). SUM SQUARES FUNCTION. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/sumsqu.html. Accessed (2021, Feb 28).

[4] Surjanovic, S., Bingham, D., & Fraser, S. (2013). SUM SQUARES FUNCTION. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/sumsqu.html. Accessed (2021, Feb 28).

[5] Surjanovic, S., Bingham, D., & Fraser, S. (2013). SCHWEFEL FUNCTION. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/schwef.html. Accessed (2021, Feb 28).

[6] Surjanovic, S., Bingham, D., & Fraser, S. (2013). SCHWEFEL FUNCTION. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/schwef.html. Accessed (2021, Feb 28).

[7] Surjanovic, S., Bingham, D., & Fraser, S. (2013). MICHALEWICZ FUNCTION. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/michal.html. Accessed (2021, Feb 28).

[8] Surjanovic, S., Bingham, D., & Fraser, S. (2013). MICHALEWICZ FUNCTION. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/michal.html. Accessed (2021, Feb 28).

[9] Surjanovic, S., Bingham, D., & Fraser, S. (2013). MATYAS FUNCTION. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/matya.html. Accessed (2021, Feb 28).

[10] Surjanovic, S., Bingham, D., & Fraser, S. (2013). MATYAS FUNCTION. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/matya.html. Accessed (2021, Feb 28).

[11] Surjanovic, S., Bingham, D., & Fraser, S. (2013). BUKIN FUNCTION N. 6. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/bukin6.html. Accessed (2021, Feb 28).

[12] Surjanovic, S., Bingham, D., & Fraser, S. (2013). BUKIN FUNCTION N. 6. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/bukin6.html. Accessed (2021, Feb 28).

[13] Surjanovic, S., Bingham, D., & Fraser, S. (2013). SIX-HUMP CAMEL FUNCTION. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/camel6.html. Accessed (2021, Feb 28).

[14] Surjanovic, S., Bingham, D., & Fraser, S. (2013). SIX-HUMP CAMEL FUNCTION. Virtual Library of Simulation Experiments: Test Functions and Datasets. https://www.sfu.ca/~ssurjano/camel6.html. Accessed (2021, Feb 28).

[15] Camarena, O. A. (n.d.). Some combinatorial optimization problems. Matam. https://www.matem.unam.mx/%7Eomar/math340/comb-opt.html. Accessed (2021, Feb 28).

[16] Alto, V. (2019, July 6). Neural Networks: parameters, hyperparameters and optimization strategies. Medium. https://towardsdatascience.com/neural-networks-parameters-hyperparameters-and-optimization-strategies-3f0842fac0a5. Accessed (2021, Feb 28).