Natural Computing
Assignment 1 February 9, 2021
Stijn de Boer, s1003731
Ron Hommelsheim s1000522
David Leeftink, s4496612

1 Schemata (0.5 pts)

Consider the two schemata A1 = #0#101###, A2 = ##010#111. Which of the two schemata has the highest chance to survive mutation, for a mutation rate p_m = 0.01?

The probability that a gene is not mutated is $(1 - p_m)$. For a schema to survive, all genes that belong to it should survive. Using $o(S)$ as notation for the order of a schema, this leads to the following formula: $S_m(H) = (1 - p_m)^{o(H)}$ This means that schemata with a low order are more likely to survive, as they have less genes that can be mutated.

	A_1	A_2
$o(A)$	4	6
$S_m(A)$	$(1 - 0.01)^4 = 0.961$	$(1 - 0.01)^6 = 0.941$

Table 1: The effect of mutation on two schemata. $o(A)$ is the order of the schema and $S_m(A)$ is the chance of survival under mutation.

A1 is more likely to survive therefore.

2 Building Block Hypothesis (0.5 pts)

Describe a problem where the Building Block Hypothesis does not hold.

The building block hypothesis states that GA's step-wise generate solutions by combining several lower-order schemas (i.e. building blocks). This hypothesis does not hold for every problem: for example, a problem where the reward is 1 for one particular string of genes but 0 for every other gene. An example could be guessing a 4 digit pin-code, where each digits is represented with binary sequences. There is no gradient in the loss function or any other form of information available to guide the GA.

3 Selection pressure (1 pts)

Given the fitness function $f(x) = x^2$, calculate the probability of selecting the individuals $x = 2, x = 3, and x = 4$, using roulette wheel selection. Calculate the probability of selecting the same individuals when the fitness function is scaled as follows $f_1(x) = f(x) + 20$. Which fitness function yields a lower selection pressure? What can you conclude about the effect of fitness scaling on selection pressure?

x	Fitness	$p(f)$	$p(f_1)$
2	4	4/29	24/89
3	9	9/29	29/89
4	16	16/29	36/89

Table 2: Fitness and probability of selection in roulette wheel selection for the functions $f(x) = x^2$ and $f_1(x) = f(x) + 20$

In roulette wheel selection, the probability of an individual being selected is proportional to the ratio of its own fitness and the sum of fitness of the entire population. Put in a formula, it is calculated as $p(s) = \frac{f_i}{\sum_i f_i}$. In the problem at hand, we have a population of 3 individuals. Their fitness and probabilities of being selected in roulette wheel selection are:

With $f_1(x)$, the fitness scores are much closer to one another. This causes a lower selection pressure, as a small deviation in value of x causes less difference in the probability of being selected.

4 Role of selection in GA's (2 pts)

The Counting Ones problem amounts to find a bit string whose sum of its entries is maximum. Implement a simple (1 + 1)-GA for solving the Counting Ones problem.

a) *Use bit strings of length* $l = 100$ *and a mutation rate* $p = 1/l$*. For a run of 1500 iterations, plot the best fitness against the elapsed number of iterations.*

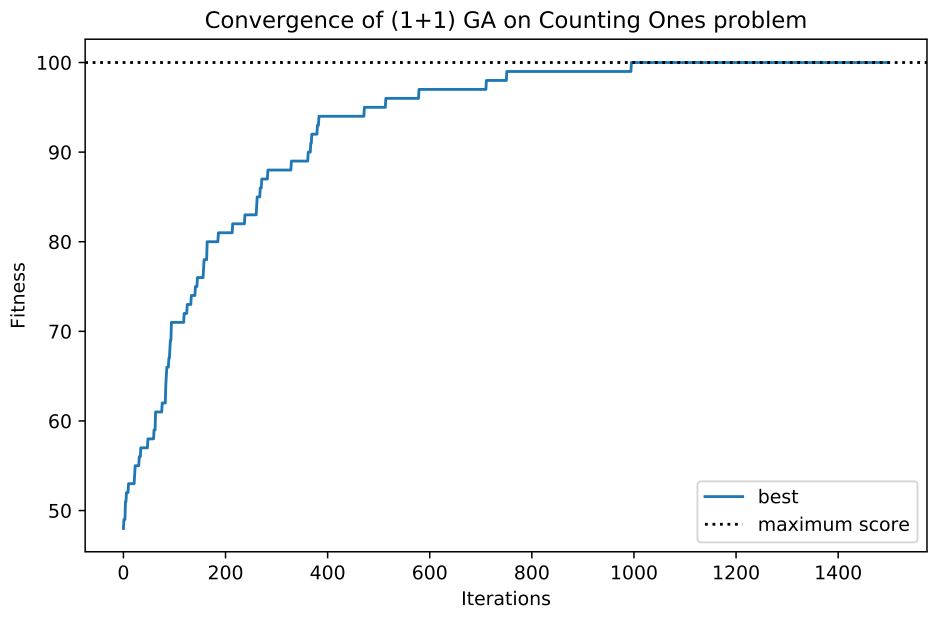


Figure 1: Convergence of (1+1) GA on Counting Ones problem

b) *Now do 10 runs. How many times the algorithm finds the optimum?* The algorithm found the optimum 9/10 runs. With 100 runs, the algorithm found the optimum 92/100 runs.

c) *Now replace (c) in the above algorithm with (c'): replace x with x_m . Is there a difference in performance when using this modification? Justify your answer*

With the alteration, the GA performs a lot worse. The fitness of the determined solution fluctuates around the initial fitness. 0/10 runs have converged to the optimum. Repeating the experiment with 100 runs shows that 0/100 runs have converged. We can conclude that rule c is an important addition in the (1+1) GA algorithm, as the algorithm forgets the best solution found thus far otherwise.

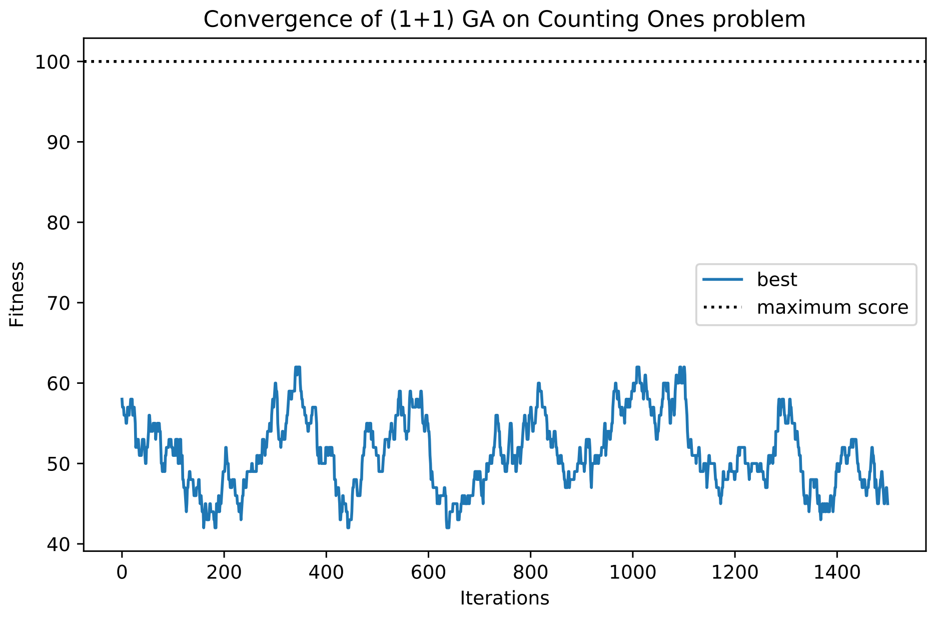


Figure 2: Convergence of (1+1) altered GA on Counting Ones problem

5 Evolutionary strategies vs local search (1 pts)

Consider a (1+5) ES. How does this differ from the (1+1) ES in how the search space is explored when optimizing a function? How does the (1+λ) ES strategy behave with respect to the value of λ when compared to greedy algorithms?

A (1+5) ES algorithm represents a population where 1 member (the best) is chosen to survive to the next generation, in which 5 new children are initialized. The (1+5) ES is different from a (1+1) ES, because in a (1+1) ES at each iteration the best solution is kept and one child is added to the population. This means that if the child outperforms the parent, it replaces the parent. If the child does not outperform the parent, it is replaced by another child. As there is one child, at each time one direction of search in the real-valued vector space is explored.

Compared to a (1+5) ES there is a large difference, since there are now 5 children that can potentially replace the parent. At each iteration, the best of the 5 children and parent is chosen to be the new parent.

With 5 children at each iteration, 5 directions of search in the real-valued vector space are explored.

Since ES are concerned with optimizing real valued vectors (such as weights), we can compare the optimization behaviour with standard hill-climbing methods such as greedy or gradient-descent. Whereas these methods *always* take a step in the new direction, as (1+λ) ES only takes this step if the fitness of the step is actually improved. The higher the value of λ, the more informed the next step of the iteration becomes as more directions in the search space are explored. This can be advantageous, as this allows the algorithm to 'remember' certain solutions: it does not accidentally forget the best solution due to an unfortunate sequence of mutation, recombination and cross-over operations. However it can also cause the system to be more likely to be trapped in local optima/minima.

The procedure will pick the best of the λ children at the next time step, because of the '1' in (1+λ). As we let λ go to infinity, all *possible* children will be evaluated, and of those children the best is picked. This means that the (1+λ)-ES algorithm becomes a greedy algorithm if we let $\lambda \to \infty$, because greedy algorithms perform a sequence of locally optimal steps.

6 Memetic algorithms vs simple EAs (2.5 pts)

The implementation of the standard EA and of the memetic algorithms are made using the python library DEAP (Coelli, 1996). The implementation can be found at `https://github.com/DavidLeeftink/NaturalComputing2021/blob/main/Assignment%201/A1%20Natural%20Computing.ipynb`. Jupyter notebooks can be run by downloading the conda package; further instructions on how to install the notebook are detailed at `https://jupyter.org/install`. All other required packages are automatically installed by executing the code. The required data of the cities is given in the GitHub is as well, and can be downloaded to run the program.

a) *Implement a variant of this algorithm based on memetic algorithms.*

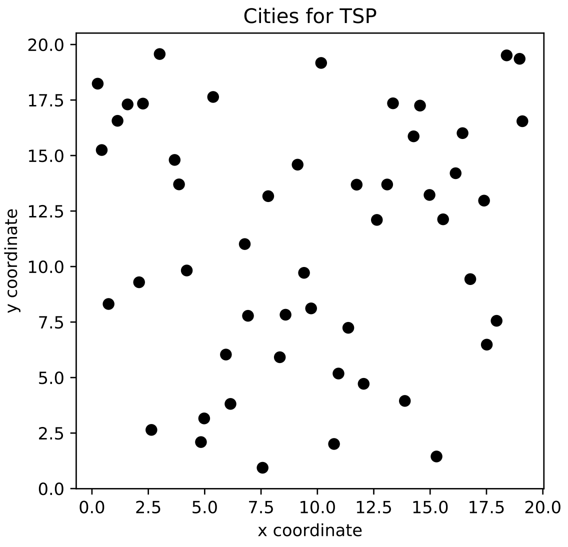


Figure 3: Given cities to be connected for the TSP problem.

b) *On the TSP problem are memetic algorithms more effective than the simple EA' s?*

As shown in figure 4 and 5, the memetic algorithm converges faster to a lower distance (higher fitness). The memetic algorithm was chosen to run for only 100 iterations, as it converges quickly and requires a much higher computation per iteration. After 100 iterations, it has reached a minimum distance near 150: we believe this might be the global optimum as well. The simple EA performs a lot worse: after 1500 iterations, the best distance is near 250.

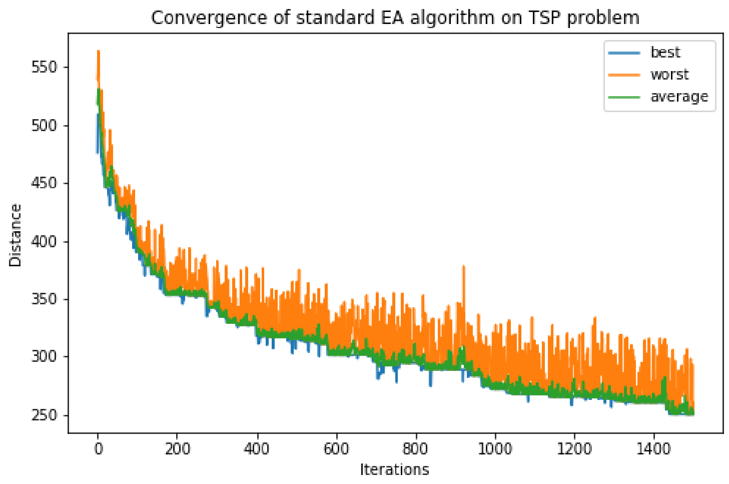


Figure 4: Convergence of standard EA algorithm on the given data in the exercise.

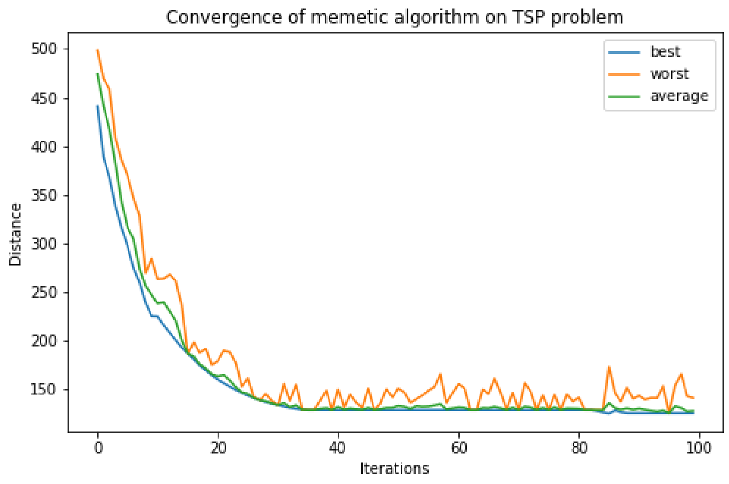


Figure 5: Convergence of memetic algorithm on the given data in the exercise.

For the second data set, we have chosen the data set `27_cities`[1], which contains (unlike the name suggests) 29 cities in Germany (Bavaria). The results of both algorithms are similar to the results found on the first data set. They are found in figure **??** and **??**.

[1]elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/bays29.tsp

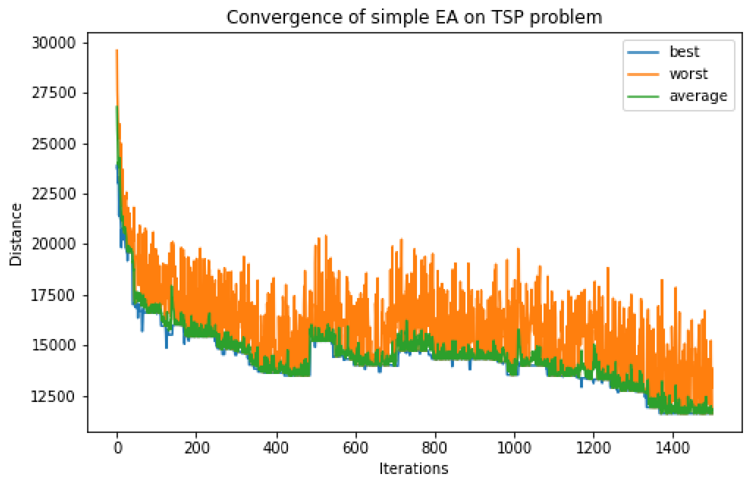


Figure 6: Convergence of standard EA algorithm on the Bavarian Cities in Germany.

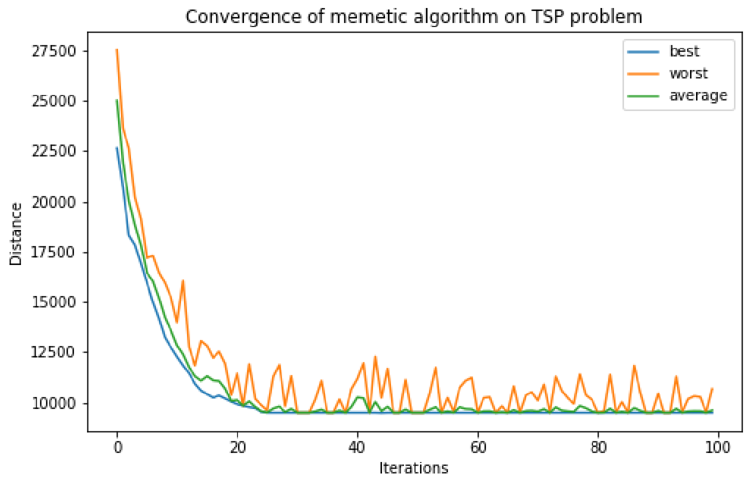


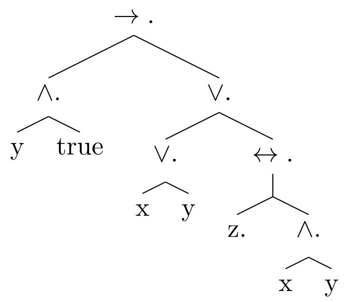
Figure 7: Convergence of memetic algorithm on the Bavarian Cities in Germany.

The results are sensible, as the memetic algorithm is performing its search in a reduced search space namely only the local optimal solutions. Comparing these findings with existing literature, we find that the memetic algorithm is not preferred in every TSP problem, however. An important factor is the size of the problem (Merz & Freisleben, 2001). In addition, (Merz & Freisleben, 2001) claim that memetic algorithms are good at exploiting the correlation structure of landscapes. If fitness landscapes are smooth, i.e. a small change of ordering in cities leads to a small change in fitness, memetic are the preferred choice. Contreras-Bolton and Parada (2015) show that standard EA algorithms are competitive in terms of performance for problems where N is high.

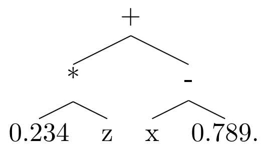
7 Genetic programming representation (0.5 pts)

Give a suitable function, terminal set and s-expression for the following logical and mathematical formulas.

a) $(y \wedge true) \rightarrow ((x \vee y) \vee (z \leftrightarrow (x \wedge y)))$
Function set: $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$
Terminal set: $\{x, y, z, true, false\}$
LISP S-expression: $(\rightarrow\ (\wedge\, y\, true)(\vee\ (\vee\, x\, y)\ (\leftrightarrow\ z\ (\wedge\, x\, y))))$
Tree representation:



b) $0.234 \cdot z + x – 0.789$
Function set: $\{+, *, -, \div\}$
Terminal set: $\{x, z\} \cup \mathbb{R}$
LISP s-expression: $(+\ (*\ 0.234\ z)\ (-\ x\ 0.789))$
Tree representation:



8 Genetic programming behaviour (2 pts)

Implement a GP program for finding a symbolic expression

The implementation of the program is made using the python library DEAP (Coelli, 1996). The implementation can be found at `https://github.com/DavidLeeftink/NaturalComputing2021/blob/main/Assignment%201/deap_experiment.ipynb`. Jupyter notebooks can be run by downloading the conda package; further instructions on how to install the notebook are detailed at `https://jupyter.org/install`. All other required packages are automatically installed by executing the code. Note that parts of code are copied from the DEAP package, to prevent additional imports. We do not claim to have written any of the functionality by the package, and merely use it to compute the optimal genetic programming solution to the given data.

The algorithm is run with the following parameters: a population size of 1000, 50 generations, $p_m = 0$, $p_c = 0.7$. In addition, we have used tournament selection with $K = 3$.

The results are shown in figure 10 and 9 for the best generation sizes vs. generation and maximum fitness vs. generations respectively. The best solution found by the program is the following expressions: `mul(add(mul(mul(x, x), x), x), sub(x, -1))`. This corresponds to the function $f(x) = x^4 + x^3 + x^2 + x$. To validate the solution, the function is visually displayed together with the observations in figure 8. This shows that the solution fits through the observations very well, and no difference between the solution and observation can be seen by eye. The good performance is validated by the small mean absolute error. Note that the optimal solution is already found at generation 6.

Whereas the mean absolute error is converging to 0, the max size of the solutions as a function of generations is increasing. This means that the longer the algorithm is run, the more complex the solutions become, and is known as the 'bloat' phenomenon (also referred to as survival of the fattest). This is likely to cause the system to overfit on the observations, due to a lack of parsimony of the model. In the end, we are interested in the simplest model that can explain the same data as well as any more complex model. In real-life applications, it would therefore be preferred to use cross-validation or use complexity penalties in model selection, to ensure that the model is able to generalize well to other values of x that are not seen in the observed data.

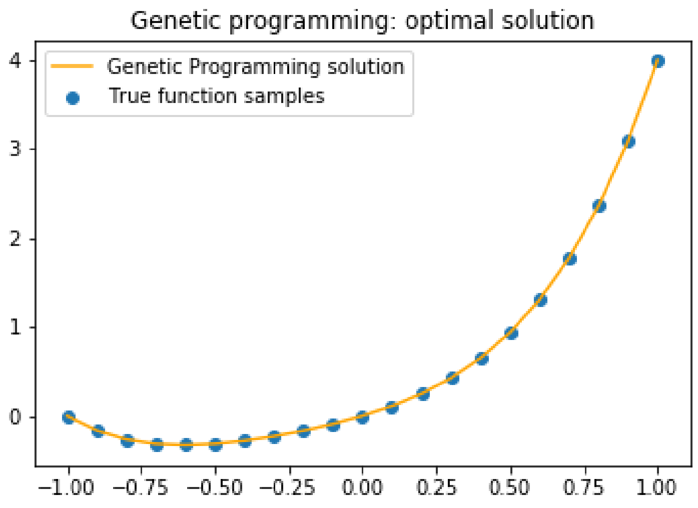


Figure 8: Genetic programming: visualization of the solution found. The final function found by the program: `mul(add(mul(mul(x, x), x), x), sub(x, -1))`. This corresponds to: $f(x) = x^4 + x^3 + x^2 + x$.

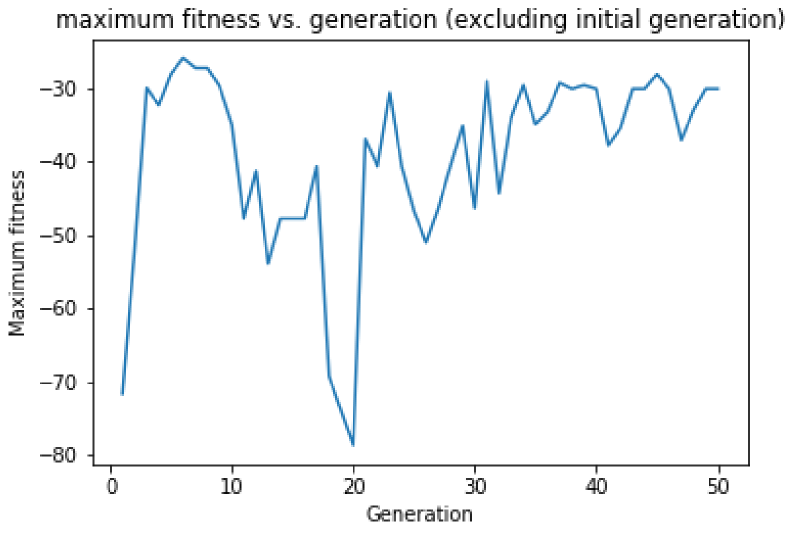


Figure 9: Maximum fitness as a function of generations (the initial generation is excluded for clearness of the plot)

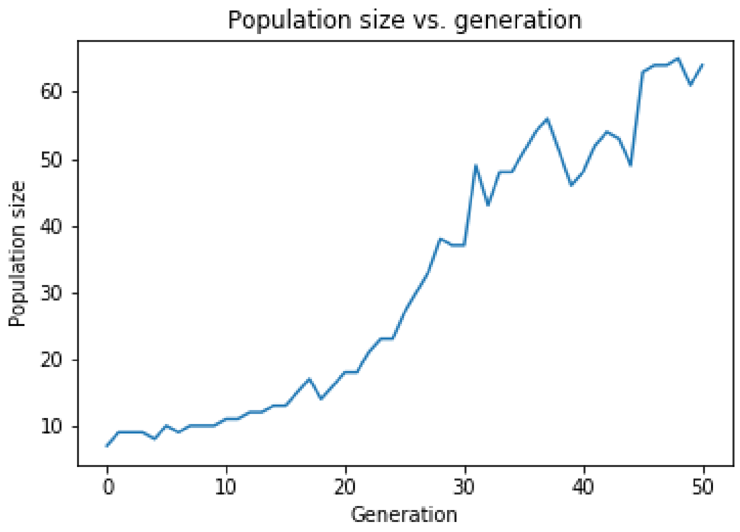


Figure 10: Best of generation size as function of generations for the Genetic Programming algorithm.

References

Coelli, T. (1996). A guide to deap version 2.1: a data envelopment analysis (computer) program. *Centre for Efficiency and Productivity Analysis, University of New England, Australia*, *96*(08), 1–49.

Contreras-Bolton, C., & Parada, V. (2015). Automatic combination of operators in a genetic algorithm to solve the traveling salesman problem. *PloS one*, *10*(9), e0137724.

Merz, P., & Freisleben, B. (2001). Memetic algorithms for the traveling salesman problem. *complex Systems*, *13*(4), 297–346.