Evolutionary Computing: Convergence behavior of Genetic Algorithms

Nick Schuitemaker
6259855
n.e.schuitemaker@students.uu.nl

Ian van de Poll
4388232
i.vandepoll@students.uu.nl

Jurre Luijten
3571858
j.t.luijten@students.uu.nl

Abstract

Motivation Evolutionary Computing (INFOEA) Practical Assignment 1

Supplementary Material `https://github.com/tragamota/Evolutionary-computing`

1 Introduction

Genetic algorithms (GAs) are highly regarded optimization techniques that utilize the principles of evolution to discover optimal solutions for a variety of problems. Despite the significant achievements in machine learning and deep learning in recent years, GAs have been underestimated. To investigate the strengths and limitations of GAs, we examine a simple optimization problem concerning the convergence behavior of three fitness functions by tracking and analyzing population measures during the optimization process. We conduct five experiments using a GA that applies family competition, where two parent solutions generate two offspring solutions through either 2-point crossover or uniform crossover, and the two best solutions are selected for the next generation. The population size is a vital parameter for the GA, and we use a bisection search to determine the minimum required population size for each experiment. Multiple runs are conducted to mitigate the stochastic effects of the GA until reliable optimal solution(s) are found. This report aims to gain insights into the convergence behavior of the GA and provide a better understanding of its effectiveness in solving optimization problems.

2 Methods

2.1 Experiments

We run 5 experiments, each with a different fitness function, and apply both a GA with Uniform crossover (UX) as with 2-point crossover (2X) to these problems. In total, this results in 10 independent runs. We only consider solutions of string-length $l = 40$. Thus, the optimal fitness of all used fitness functions is 40.

We use bisection search to find the smallest population size necessary for optimizing the function. To determine this minimum population size, we start with N = 10 and double it until a solution with the global optimum has been found or the maximum population size of 1280 has been reached. In case a global solution has been found the bisection search gets applied to find the minimal population. In every stage, we perform 20 independent runs to mitigate the stochastic effect, and if 19 of these runs find the global optimum we consider the problem to be solved. The GA stops when a new offspring solution is the global optimum or when no offspring with a higher fitness has been created in the previous 10 generations.

In addition to these experiments, we dive deeper into the behavior of the population during the optimization process with the counting-ones fitness function. Different measures are tracked on a population size of $N = 200$ on a single run: the proportion of each bit (0 and 1) in the population; the number of selection decisions (more on this in the Results section); and the number of solutions belonging to a certain schema, along with their fitnesses.

The Genetic Algorithm and all experiments were written in Python 3.11, and we did not use any external libraries other than *numpy* for basic functions and *matplotlib* for plotting. The experiments were conducted on a machine with an Intel Core i7 12700k and no multi-threading or parallelization was used.

2.2 Genetic Operations

Most GAs in the literature contain both crossover and mutation. A crossover is a form of *exploitation*, recombining existing good solutions, while mutation is a form of *exploration*, providing the algorithm with new directions to search in.[1] Sometimes, crossover is also seen as a form of *exploration*.[2] To simplify the problem and only focus on the convergence behavior of GAs, we ignore the mutation and only consider the crossover operation. This has implications, as we will note in the Discussion.

We consider both Uniform Crossover (UX) and Two-point crossover (2X) for crossover operation. In short, UX randomly swaps every bit with a certain probability p between two solutions, and 2X randomly picks two crossover points in the solutions and swaps all bits in-between them. Thus, 2X assumes structure in adjacent bits by crossing over blocks of adjacent bits, while UX assumes no structure but requires a parameter to be set. For UX we use the values $p = 0.05$ and 0.15. These swap on average respectively 2 and 6 of the 40 bits

[1] "The crossover operator implements a depth search or exploitation, leaving the breadth search or exploration for the mutation operator.", from CMU School of Computer Science: https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume24/ortizboyer05a-html/node1.html

[2] See the slides of Dirk Thierens, 'Model-Based Evolutionary Algorithms Part 1: Estimation of Distribution Algorithms', slide 2

in an individual.

For every iteration t we apply the following steps to a given population P_t:

1. Combine the entire population P_t into random pairs of 2 solutions (parents) and for every such pair, create 2 offspring solutions (children) using crossover.
2. For every such family consisting of two parents and two children, let the two solutions with the best fitness survive to the next generation P_{t+1}.

In the case of ties between parents and children with the same fitness, the children are favored to go to the next generation.

2.3 Fitness Functions

We consider 5 different fitness functions to be optimized. They are chosen to be increasingly difficult to solve for optimization algorithms. The fitness functions to optimize are:

1. Counting One's Function; the count of the number of 1s found in the entire bitstring
2. Non-deceptive Trap Function (tightly linked)
3. Deceptive Trap Function (tightly linked)
4. Non-deceptive Trap Function (unlinked)
5. Deceptive Trap Function (unlinked)

The Trap function is defined as follows:

$$B(x_1 \dots x_k) = \begin{cases} k & \text{if } CO(x_1 \dots x_k) = k \\ k - d - \frac{k-d}{k-1} CO(x) & \text{if } CO(x_1 \dots x_k) < k \end{cases} \tag{1}$$

The deceptive trap functions have $k = 4, d = 1$ and the non-deceptive trap functions have $k = 4, d = 2.5$. The tightly linked trap functions have the blocks consisting of k adjacent bits, whereas the unlinked trap functions have every block being spread out over the bitstring, thus consisting of non-adjacent bits. The unlinked versions of the trap functions are designed to mess with genetic operators that assume structure in adjacent bits, such as 2X crossover.

Trap functions are named as such because they tend to drive solutions towards local optima rather than global optima. This can be seen in the fitness function, since getting more 1's continuously decreases fitness until the global optimum of 1111 is reached. These trap functions are known to be difficult to optimize, and thus are great to view how GAs perform in a search space where the solutions are pressured away from the global optima.

The deceptive functions have the property that the fitness not only improves towards local optima rather than global optima but also that it wants to be in the area of a local optimum rather than a global optimum. This occurs because the strength (average fitness) of areas around local optima are stronger than those of global optima, as shown in this computation:

$$f(000\#) = \frac{2+3}{2} = 2.5$$

$$f(111\#) = \frac{4+0}{2} = 2$$

where f is the average schemata fitness. As $2.5 > 2$, the solutions will tend to converge more to the schemata 000# than 111#. A similar thing can be shown for the schemata:

$$f(00\#\#) = \frac{1+2+3}{3} = 2$$

$$f(11\#\#) = \frac{4+0+1}{3} \approx 1.66$$

However:

$$f(0\#\#\#) = \frac{0+1+2+3}{4} = 1.5$$

$$f(1\#\#\#) = \frac{4+0+1+2}{4} = 1.75$$

This property does not hold for the non-deceptive trap function, where $f(111\#) > f(000\#)$, yet as it is still a trap function, changing a 1 into a 0 will still likely improve the fitness.

Note that, as we merely sum over the number of 1s, the order of 1s is irrelevant in the solutions and these schemata have the same fitness for any reordering of the symbols.

3 Results

Below you can find 5 tables (one for each test function) reporting

- the number of runs that were successful for the minimal population size, or for a population size of 1280
- the minimal population size for which 19 out of the 20 runs are successful
- the average number of generations for the minimal population size with the standard deviation between parentheses
- the average number of fitness function evaluations for the minimal population size with the standard deviation between parentheses
- the average CPU time (seconds) required for running the GA (20 times) for the minimal population size with the standard deviation between parentheses

A fail is reported whenever less than 19 out of the 20 runs were successful for a population size of 1280.

Table 1 Summary of GA Performance - Counting One's Function

Metric	2X Crossover	UX Crossover ($p = 0.05$)	UX Crossover ($p = 0.15$)
Number of successful runs	20	19	19
Minimal population size	50	170	40
Average no. of generations	20.4 (3.71)	25.35 (2.032)	21.3 (2.74)
Average no. of evaluations	1020	4310	852
Average CPU time per run (s)	0.028 (0.0054)	0.134 (0.016)	0.0256 (0.0048)

Table 2 Summary of GA Performance - Non-deceptive Trap Function (tightly linked)

Metric	2X Crossover	UX Crossover ($p = 0.05$)	UX Crossover ($p = 0.15$)
Number of successful runs	20	18	19
Minimal population size	130	FAIL	390
Average no. of generations	18.45 (2.156)	FAIL	29.5 (3.956)
Average no. of evaluations	2399	FAIL	11505
Average CPU time per run (s)	0.401 (0.0405)	FAIL	1.916 (0.502)

Table 3 Summary of GA Performance - Deceptive Trap Function (tightly linked)

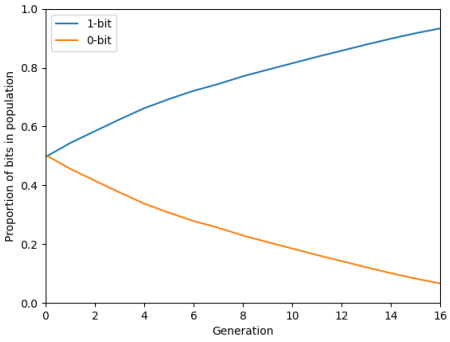
Metric	2X Crossover	UX Crossover ($p = 0.05$)	UX Crossover ($p = 0.15$)
Number of successful runs	20	0	0
Minimal population size	230	FAIL	FAIL
Average no. of generations	22.05 (2.84)	FAIL	FAIL
Average no. of evaluations	5072	FAIL	FAIL
Average CPU time per run (s)	0.86 (0.109)	FAIL	FAIL

Table 4 Summary of GA Performance - Non-deceptive Trap Function (unlinked)

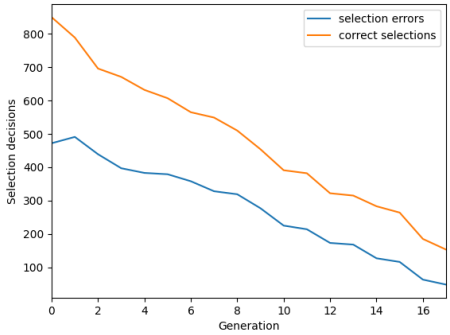
Metric	2X Crossover	UX Crossover ($p = 0.05$)	UX Crossover ($p = 0.15$)
Number of successful runs	18	16	20
Minimal population size	FAIL	FAIL	570
Average no. of generations	FAIL	FAIL	28.8 (3.57)
Average no. of evaluations	FAIL	FAIL	16416
Average CPU time per run (s)	FAIL	FAIL	2.351 (1.095)

Table 5 Summary of GA Performance - Deceptive Trap Function (unlinked)

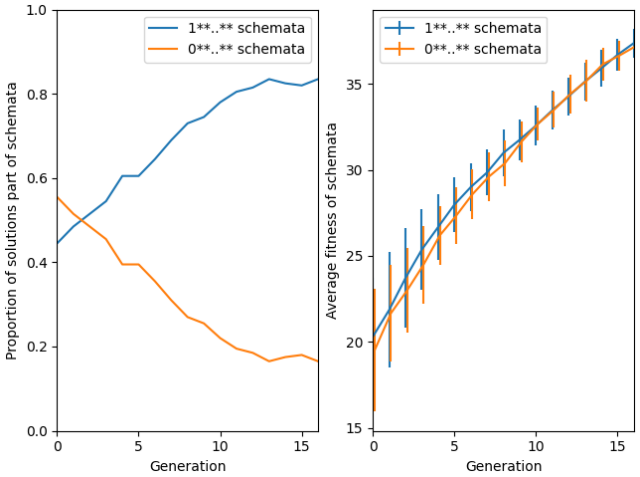
Metric	2X Crossover	UX Crossover ($p = 0.05$)	UX Crossover ($p = 0.15$)
Number of successful runs	0	0	0
Minimal population size	FAIL	FAIL	FAIL
Average no. of generations	FAIL	FAIL	FAIL
Average no. of evaluations	FAIL	FAIL	FAIL
Average CPU time per run (s)	FAIL	FAIL	FAIL



(a) The proportions of bits in the population against the generation. The 1-bit and 0-bit are inversely proportional.



(b) The number of selection errors and correct selections against the generation.



(c) The proportion of solutions part of schemata, and the average fitness of those schemata, plotted against the generation. Note that in the left image, the two schemata are inversely proportional. In the right image, the vertical bars represent standard deviations.

Figure 1 Results of a GA being applied to the Counting Ones problem with population size $N = 200$.

Moreover, Figure 1 shows the result of an additional experiment, measuring the population when optimizing the Counting Ones function with a population size of $N = 200$. The figure shows:

(a) The proportion of bits in the population, plotted against the generation. We would expect the number of 1s to be increasing over time, and thus the number of 0s to be decreasing.
(b) The number of selection errors and correct selection decisions, plotted against the generation. A selection error appears when two parents cross a 0- and 1-bit over at a position i, and the winners both have a 0-bit in that position i. Similarly, a correct selection appears when both winners have a 1-bit in the respective position i.
(c) The proportion of solutions part of schemata 1##..## and 0##..##, plotted against the generation (noted $1**..**$ and $0**..**$ in the Figure). We would expect the number of solutions part of 1##..## to be increasing over time, and thus 0##..## to be decreasing. Additionally, the figure shows the average fitness of the respective

schemata, plotted against the generation, along with the standard deviations (vertical bars).

4 Discussion

From Table 2 and 3, it becomes clear that 2X crossover performs better than UX crossover for tightly linked trap functions. This is shown in the non-deceptive variant through the number of successful runs (20 compares to 18 and 19) and the minimal population size (130 vs 390), and in the deceptive variant simply through the number of successful runs (20 vs 0 and 0). This result is due to the fitness function, in which adjacent blocks depend on each other for the score. In 2X crossover, many blocks will be preserved after crossover (all except a maximum of 2 as only 2 crossover cuts are made), thus preserving important building blocks. On the other hand, UX crossover swaps bits at random, not considering any structures.

From Table 4 and 5, it does not seem that 2X crossover outperforms UX crossover for unlinked trap functions. This is due to the fitness functions, where the bits in a block are spread out over the solution. The deceptive unlinked trap function notably gets 0 successful runs for all crossover variants, due to the bad convergence behavior of GAs on trap functions, and due to the lack of structure in unlinked blocks. In non-deceptive unlinked trap functions however, 2X performs reasonably well, completing 18 out of 20 runs successfully, but UX crossover with $p = 0.15$ outperforms it significantly, finishing 20 out of 20 runs successfully with a low population size of 570.

The reason 2X crossover performs worse in unlinked functions compared to the tightly linked versions is as follows: because in 2X crossover two splitting points are set, all tightly linked blocks would remain intact except for a maximum of two (if the splitting points are both within rather than between blocks). However, that same crossover is likely to split a majority (if not all) of the unlinked blocks due to their bits being spread over the solution. Thus, 2X crossover does not preserve nearly as much structure in unlinked trap functions as in tightly linked trap functions.

The reason why UX crossover performs better is that it is less biased to cross adjacent bits over and can instead cross over bits uniformly. Though 2X is likely to split many unlinked blocks by crossing over only parts of their bits, UX is less likely to do so. It can be shown that the chance of a single block to be kept intact (by crossing over all or none) by UX is $p^4 + (1 - p)^4$ for a crossover chance p. This is 0.81 for $p = 0.05$ and 0.52 for $p = 0.15$. Similarly, by case enumeration, it can be shown that the chance of a single block to be kept intact by 2X is 0.25 for a string-length $l = 40$. Thus, UX is more likely to preserve blocks than 2X. Now, the version of UX with $p = 0.15$ does perform significantly better than the version with $p = 0.05$, likely because $p = 0.05$ is simply too low for the solutions to converge in a reasonable amount of runs and with reasonable population size.

An additional thing to note is what the effect is of not using mutation. Considering that we only cross bits over and have no way to introduce new bits in the population, bits at certain positions can die out. More precisely, if all individuals of a population contain the same bit at a position i, this position i will only contain that bit in all individuals of all future generations. If this happens with a 0-bit, this will prevent the global optimum from ever being found. We have not measured how often this occurred in the experiments, but it might be a partial explanation as to why deceptive problems are typically not solved. This is also a reason why crossover is typically referred to as an exploitation operator rather than

an exploration operator[3], as crossover does not introduce new bits in the population.

Figure 1 mostly follows expectations. Figure 1a agrees with our intuition: given that the fitness function favors more 1s in the solutions, we expect the proportion of 1s in each solution to converge to 1 over time. As we start with random solutions, the proportion starts around 0.5, and as time goes on, more and more 1s find their way into the population. In this run, the proportion of 1-bits never reaches 1, since only a single solution needs to find the global optimum for the stop criterion to be reached. However, if all solutions would need to find the global optimum for the simulation to be halted, the proportion of 1-bits in the population would likely reach 1 in the final generation.

Figure 1b shows that the total number of selection decisions is dropping. This makes sense given that, over time, all solutions will converge to similar values, which results in there being fewer disagreements (the sum of both lines at any x-value). Also, the correct decisions always seem to dominate the errors. This also makes sense, given that correct decisions result in more 1s while decision errors result in fewer 1s. As the fitness is based on the sum of all 1s, a winning solution will likely have more correct selection decisions than selection errors.

Similar to Figure 1a, Figure 1c shows that the number of solutions containing a 1 as the first bit will outgrow the number of solutions containing a 0 as the first bit, which is expected as the former has a higher average fitness given the extra 1 in the solutions. On the right is a bit more surprising image. The average fitness of both schemata at any generation is almost the same. This raises the question of where the selection pressure of choosing the 1##..## schemata over the 0##..## schemata comes from. The answer to that, firstly, is that the 1##..## schemata nearly always dominate the 0##..## schemata, though only slightly. As the algorithm simply picks solutions with the highest fitness, it doesn't matter that this difference is only small or large, which results in the 1##..## schemata being preferred on average. Secondly, this small change is logical, as the two schemata differ only by one bit (thus, the fitness also differs by 1 on average). Also note that the largest differences in fitness, from around 0-10 (right image), correspond to the largest increases in the proportion of the 1##..## schemata (left image). This shows that increases in the fitness of 1##..## do increase the proportion of 1##..## in the population, suggesting that selection pressure does occur.

[3] though the slides of Dirk Thierens disagree, see 'Model-Based Evolutionary Algorithms Part 1: Estimation of Distribution Algorithms', slide 2