Evolutionary Computation

Practical Assignment 1

Daan Westland l.d.westland@students.uu.nl
Julius Bijkerk j.j.bijkerk@students.uu.nl

1 Our Approach

For this assignment we chose to work in Python. It seems an obvious choice, but to explain, we both have extensive experience in this language (compared to Java for example), it is easily readable, and there are lots of resources online to assist us in developing a solution.

We are working in Visual Studio Code using Git and GitHub to facilitate working together. Next, we developed the solution in such a way that it is runnable locally on both of our own machines (Apple M3 chip and AMD RYZEN 7600X) to get the results.

1.1 Genetic Algorithm

For this practical assignment, we have implemented a genetic algorithm (GA) to explore its convergence behavior across different fitness functions. The family competition model forms the basis of our GA. Here, two parent solutions generate two offspring through crossover methods. Each generation, after shuffling the population, pairs of solutions are crossed to produce offspring, with the best two out of these four (parents and children) moving to the next generation. We prioritize offspring over parents when they share the highest fitness score to favor generational progress. The GA uses either uniform crossover (UX) or two-point crossover (2X) for recombination. This way, we offer a comparative view of these methods across various tests.

1.2 Fitness Functions

Our solution includes three fitness functions within the binary domain, each with a string length of: $l = 40$. Hereby, we are aiming to maximize the function values to a string of all ones:

1. Counting Ones Function: Simply counts the number of '1's in the string.
2. Trap Functions (Tightly Linked): Comprises ten contiguous sub-functions of four bits each. The function rewards complete '1's in a sub-function but penalizes any deviation based on a deception factor d:
   (a) Deceptive Trap Function: Here, $d = 1$, intensifying the penalty unless all bits are '1's.
   (b) Non-deceptive Trap Function: Here, $d = 2.5$, with a softer penalty gradient, which makes it less misleading.
3. Trap Functions (Not Tightly Linked): This variant spreads the same tightly linked sub-functions across the string to investigate the impact of gene linkage on the performance of the GA.

1.3 Experiments

The experiment is divided into five main sub-experiments. By doing this, we test the GA under different conditions with both crossover operators:

1. Experiment 1: Focuses on the Counting Ones function.
2. Experiment 2: Tests the Deceptive Trap Function (tightly linked).
3. Experiment 3: Evaluates the Non-deceptive Trap Function (tightly linked).
4. Experiment 4: Applies the GA to the Deceptive Trap Function with sub-functions not tightly linked.

5. Experiment 5: Assesses the Non-deceptive Trap Function with non-tightly linked sub-functions.

For each function, we experiment with both UX and 2X to detect which crossover method enhances the GA performance under different constraints. The population sizes are critical, starting at $N = 10$ and doubling until either the optimum is found or a maximum of $N = 1280$ is reached. If a suitable population size is found, a bisection search is conducted to refine and determine the smallest effective population size.

1.4 Stopping Criteria and Repetition

The GA stops when an offspring matches the global optimum or if no offspring surpasses its parents in fitness after 20 generations. This addresses both the convergence and a potential stagnation. We perform multiple runs for each setup to average out the randomness of the GA. Therefore, we consider a problem solved if at least 9 out of 10 runs reach the optimal solution. This approach allows us to systematically analyze the behavior of genetic algorithms across varied settings.

2 Observations

Running our GA resulted in interesting insights about how different factors interact within the evolutionary computations. We noticed that the choice of crossover operators (Uniform Crossover in particular) played an important role in the algorithm's effectiveness. This operator consistently delivered reliable results, especially when applied to the tightly linked configurations of the problem. This highlights the suitability for maintaining genetic diversity while navigating challenging fitness landscapes.

Another critical factor was the population size. It became clear that more complex or deceptive functions demanded larger populations to adequately explore the solution space and converge to satisfactory answers. This was especially visible in our experiments with simpler fitness functions like 'Counting Ones', where the algorithm quickly found the optimal solutions. This behavior is underscored by a steady increase in the proportion of ones in the population over successive generations.

However, it became worse with the more intricate deceptive trap functions. Here, the algorithm sometimes got stuck, struggling to find consistency and demanding frequent tweaks to population sizes while also ramping up the computational load (resulting in longer loading times).

3 Conclusions

Reflecting on our experiments, it's apparent that 'Uniform Crossover' stands out as particularly robust, streamlining the path to success in tightly linked setups by requiring smaller populations and fewer generations. This underscores a broader theme that while genetic algorithms are potent tools for tackling complex optimization challenges, their efficiency hinges heavily on how well their parameters are tuned to the problem's nuances, such as its fitness landscape.

Moreover, our exploration of these computational experiments reaffirmed that while our GA implementation can skillfully navigate simpler problems, unveiling the dynamics of genetic operations in the process, it also showed that tackling more complex or deceptive issues might require more refined strategies. These could include enhanced selection mechanisms or more adaptive crossover techniques to better handle the unpredictable terrain of difficult optimization problems.

4 Tables

4.1 Counting Ones Function

Crossover	Min Pop Size	Avg Generations	Avg Fitness Evaluations	Avg CPU Time (s)
2X	40	25.75 (8.98)	3130 (1077.91)	0.01466 (0.00435)
UX	30	16.5 (1.63)	1515 (146.51)	0.00864 (0.00125)

4.2 Deceptive Trap Function (Tightly Linked)

Crossover	Min Pop Size	Avg Generations	Avg Fitness Evaluations	Avg CPU Time (s)
2X	120	27.75 (5.54)	10110 (1994.27)	0.08225 (0.01738)
UX	610	750.89 (172.61)	1374737 (315882.77)	12.31 (2.70)

4.3 Non-deceptive Trap Function (Tightly Linked)

Crossover	Min Pop Size	Avg Generations	Avg Fitness Evaluations	Avg CPU Time (s)
2X	80	25.67 (4.67)	6240 (1120)	0.05266 (0.01176)
UX	100	58.13 (26.33)	17538 (7899.52)	0.15792 (0.06594)

4.4 Deceptive Trap Function (Not Tightly Linked)

Crossover	Min Pop Size	Avg Generations	Avg Fitness Evaluations	Avg CPU Time (s)
2X	FAIL	-	-	-
UX	440	672.67 (223.95)	888360 (295609)	8.84 (2.84)

4.5 Non-deceptive Trap Function (Not Tightly Linked)

Crossover	Min Pop Size	Avg Generations	Avg Fitness Evaluations	Avg CPU Time (s)
2X	1010	41.22 (3.39)	125913 (10278)	1.20 (0.09)
UX	120	43.11 (5.57)	15640 (2004)	0.16058 (0.01862)

5 Acknowledgments

We would like to express our thanks to Dr. ir. D. (Dirk) Thierens for the assignment and lectures. Our thanks also go to the creators of related software and packages used for this project. Lastly, we wanted to show our appreciation to the online community, of which StackOverflow is a great example.

6 Appendix

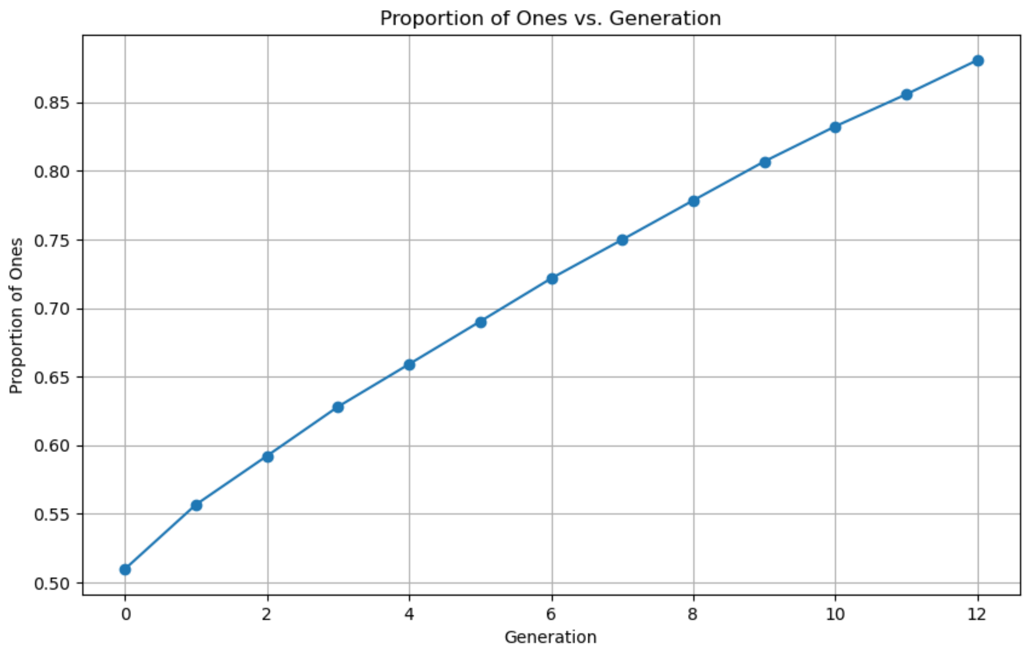

Figure 1: Experiment 1 Proportion Of Ones

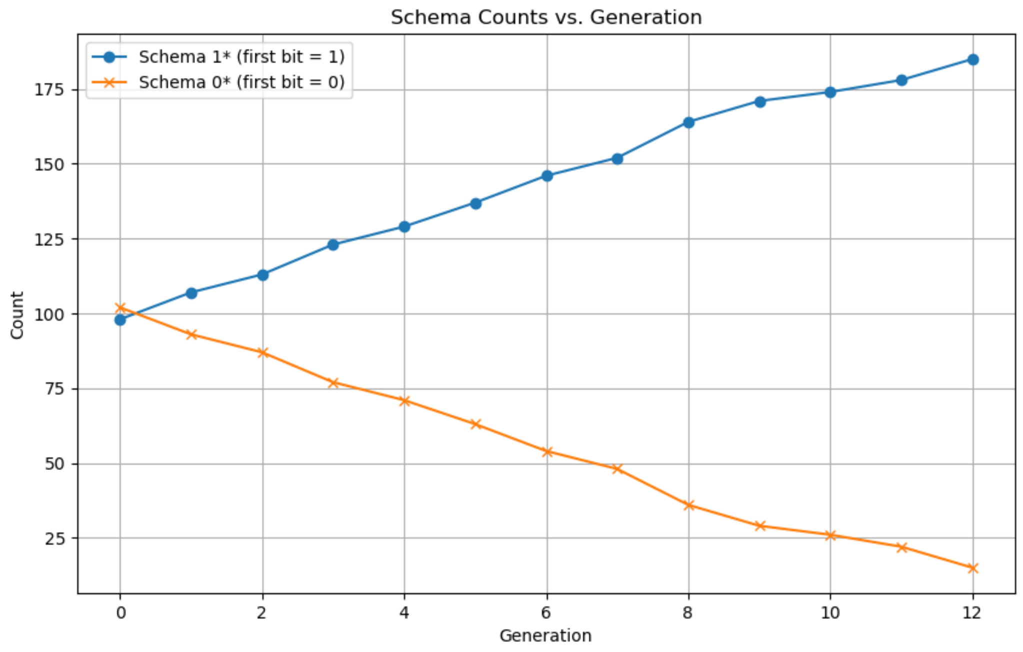

Figure 2: Experiment 1 Schema Counts

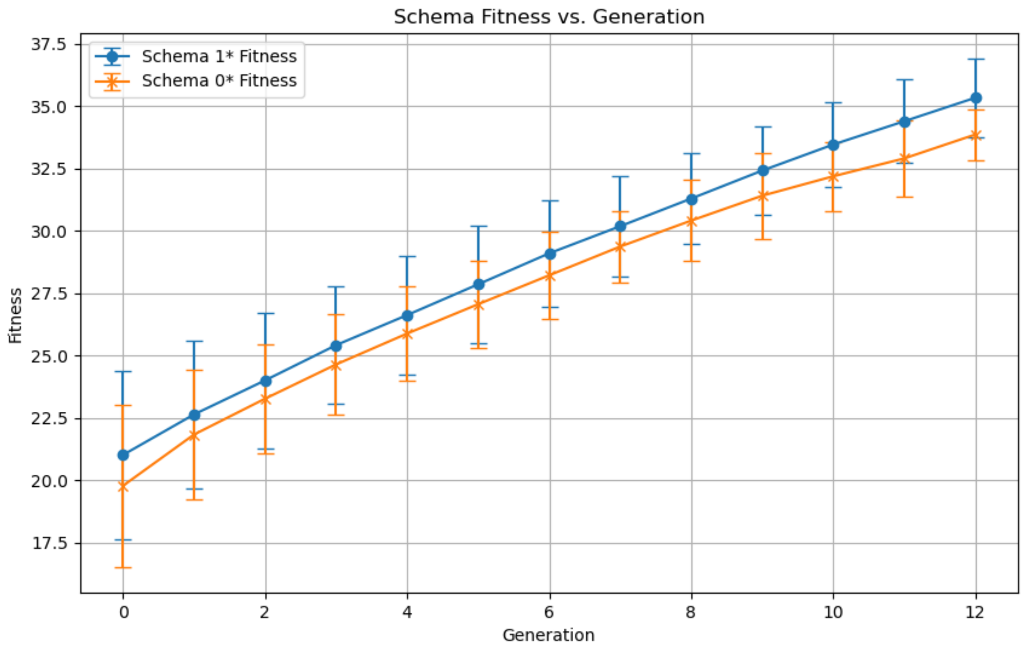

Figure 3: Experiment 1 Schema Fitness

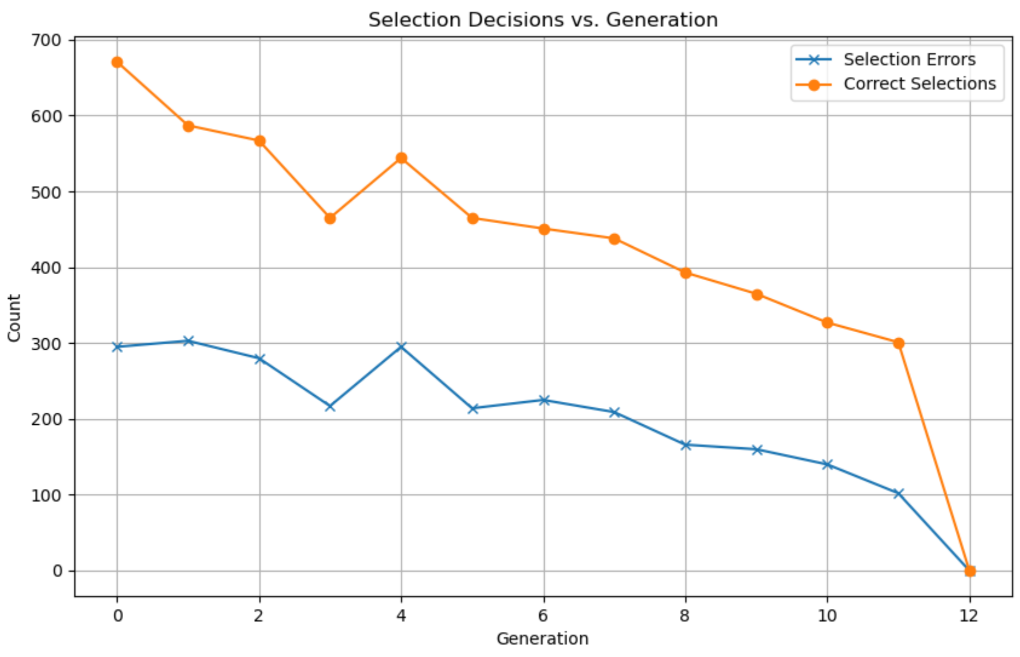

Figure 4: Experiment 1 Selection Decisions