HOMEWORK MODULE: PROGRAMMING THE BASICS OF AN EVOLUTIONARY ALGORITHM(EA)

SIGVE SEBASTIAN FARSTAD

1. Introduction

This report presents a solution to Homework Module: Programming the Basics of an Evolutionary Algorithm(EA), IT3708, spring 2014, NTNU. The assignment is to implement all of the basic components of an evolutionary algorithm, and use them to solve the One-Max problem.

2. The Evolutionary Algorithm Implementation

The evolutionary algorithm solver presented in this solution is implemented as a modular python library. A generic solver function solves arbitrary problems supplied by the user of the library.

The flow of the evolutionary algorithm is identical to the one proposed in section 1 of *EA Appendices*[1] and is illustrated in figure 1.

Each step in the algorithm can be implemented using different strategies. Different strategies have different success rates for different problems, so it is important to choose a good strategy for a given problem. Because of this, the solver library implements multiple different strategies for each step of the algorithm. The different implemented strategies are elaborated upon in the following sections.

[1]K. Downing, EA Appendices, January 19, 2014

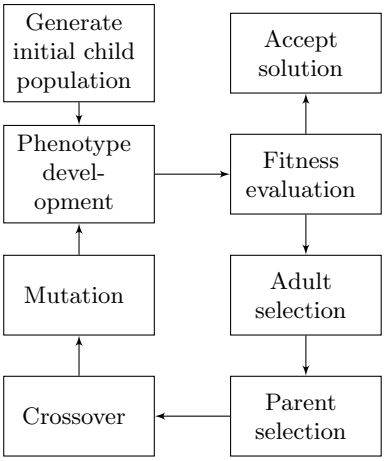


Figure 1. An abstract overview of the algorithm flow.

2.1. **Adult Selection.** Adult selection is the process of selecting individuals from the children pool and the adult pool to form the adult pool of the next generation. As suggested in *EA Appendices*, the solver implements three different strategies for adult selection: full generation replacement, over-production, and generation mixing.

2.1.1. *Full Generational Replacement.* Full generational replacement is an adult selection strategy where the adult population of generation n consists of the entire child population of generation $n-1$. It requires that the child and adult populations are equally large.

2.1.2. *Over-production.* Over-production is an adult selection strategy where the adult population of generation n consists of a true subset of the child population of generation $n-1$. The subset may be selected using many different methods. Weighting the subset selection by fitness is a popular choice. The implementation in the solver presented in this report uses unweighted random selection, as it is easier to implement.

Over-production requires that the child population is larger than the adult population.

2.1.3. *Generational Mixing.* Generational mixing is an adult selection strategy similar to over-production, except that the selected subset is a subset of both the child and adult populations from the previous generation, rather than just the child generation. The same considerations for subset selection apply for generational mixing. The implementation in the solver presented in this report uses unweighted random selection here, too.

Generational mixing does not impose any restrictions on population sizes.

2.2. **Parent Selection.** Parent selection is the process of selecting adults for combination in the crossover stage. Parent selection strategies must take care to select fit individuals while at the same time allowing for the variance that allows the algorithm to avoid getting stuck at local maxima. The presented solver implements

five different parent selection strategies: fitness-proportionate selection, sigma scaling selection, tournament selection, rank selection, and random selection.

2.2.1. *Fitness-Proportionate Selection.* Fitness-proportionate selection is a selection scheme where an individual is selected at random, with the probability of being chosen equal to $\frac{f(i)}{F}$, where $f(i)$ is the fitness of an individual, and F is the total fitness of the population.

2.2.2. *Sigma Scaling Selection.* Sigma-scaling selection is similar to fitness-proportionate scaling, but the probability of a given individual being chosen is scaled by the fitness variance of the population, as in equation 3 of *EA Appendices*.

2.2.3. *Tournament Selection.* Tournament selection is a selection scheme where a predefined $k < |population|$ number of individuals are selected for participation in a tournament. Of these competing individuals, the one with the greatest fitness score gets selected. Some tournament selection implementations use a variant of fitness proportionate scaling to select the winner individual from the tournament participants, but that is not the case in the presented solver. This is because always selecting the best individual amongst the tournament participants guarantees that the $k-1$ worst individuals in a population will not get selected.

2.2.4. *Rank Selection.* Rank selection is a selection scheme similar to fitness-proportionate selection, only the probability of selection is defined as in equation 1.

(1)
$$P_{sel} = 2 \times \frac{rank(i)}{|population| \times (|population| + 1)}$$

2.2.5. *Random Selection.* Random selection is a selection scheme that simply selects an individual at random. It is not a good selection scheme for an EA, but is included in the selection suite for benchmarking purposes.

2.3. **Crossover.** Crossover is a method for combining the genome of two individuals to create a new individual based on the originals. The evolutionary algorithm implementation made for this assignment offers two different crossover strategies: split and per-component.

2.3.1. *Split Crossover.* Split crossover is a crossover scheme where a child's left part of its genome string comes from the first parent, and the right part of its genome string comes from the second parent. The split point defining where the left part of a genome ends and the right part of a genome begins is chosen at random for each crossover.

2.3.2. *Per-component Crossover.* Per-component crossover is a crossover scheme where each component of a child's genome has an equal chance of coming from either the first or the second parent.

2.4. **Mutation.** Mutation is a method from introducing more variance into the population. The presented solver randomly mutates individuals as they are entered into the child pool. The mutation schemes in the presented solver can be supplied by the user as a part of a problem description, as good mutation schemes are highly problem-dependent.

2.5. **Modularity of the code.** The code is quite modular and reusable. It is implemented as a generic library designed with extensibility in mind. New phenotypes, genotypes, genetic operators and selection mechanisms may be developed and dropped in as replacements for existing functionality. In regular use cases, the user needs only supply a fitness function, and configure which mechanisms shall be used. Listing 1 shows how easy configuration can be. Listing 2.5 shows how to run the solver.

```
from ea.parent_selection
    import tournament_selection
...

class OneMax():
    ...
    parent_selection = \
        tournament_selection(k=8)
    ...
```

LISTING 1. The One-Max problem being configured to be solved with tournament selection.

```
from ea.solve import solve
...

ea.solve(OneMax())
```

LISTING 2. The solver solving the One-Max problem.

3. The One-Max Problem

The One-Max problem is a simple problem popularly chosen as a demonstration problem for testing evolutionary algorithm implementations. It has no other real-world applications. The problem is: what is the binary string of length n that contains the largest number of Ones? The problem is trivial, but is good for basic evolutionary algorithm testing because of its transparency and simplicity.

3.1. **Genotype and Phenotype Representation.** A single genotype is represented as a binary vector of length n. Because of the simplicity of the program, the phenotype is identical.

3.2. **Fitness Evaluation.** The fitness function $f_{\text{OneMax}}(g)$ of a genotype g is defined as equation 2, where g_i is the i^{th} bit of the genome g.

$$f_{\text{OneMax}}(g) = \sum_{i=1}^{n} g_i \tag{2}$$

4. Performance analysis

4.1. **Finding a population size.** Part of the assignment is to find an approximate minimal population size that consistently solves One-Max in less than 100 generations while using full generational adult replacement and fitness-proportionate parent selection. The population size of 150 was empirically chosen.

4.2. **Crossover and mutation evaluation.** In order to measure the effect of crossover and mutation parameters, the solver was run with different parameters while adult selection was fixed to full generational replacement and parent selection was fixed to fitness-proportionate parent selection. For each configuration tested, the solver was run 100 times, logging the number of generations until a solution was found for each run. Results of the evaluation can be found represented as a box plot in figure 3a.

The results clearly show that it is important to keep the mutation rate low enough to allow for enough exploitation. The results also seem to favor higher crossover rates. The configuration $P_{mutation} = 0.001, P_{crossover} = 1$ is chosen as the best configuration for further analysis of other components in the system. An illustration of an example run of a One-Max problem of size 40 being solved can be seen in 2.

4.3. **Comparison of Parent Selection Mechanisms.** With the same parameters and strategies as in the previous section, the efficiency of the different available parent selection mechanisms was measured. Again, each configuration was run 100 times. The results can be seen in figure 3c. Random selection was also measured, but no test runs ever produced an acceptable solution, and is therefore not plotted in figure 3c.

The results show that, of the different selection configurations tested, tournament selection with $k = 8$ gives the best results.

4.4. **Problem variations.** The One-Max problem is a special case of the more general problem of finding a specific known bit string in a known search space. Changing the target bit string in the One-Max problem is not expected to change the difficulty of the problem, but it does require a modification to the fitness function. To test this claim, the solver was tested on Known-Bitstring-Search, a problem similar to One-Max except that it tries to find a random bitstring rather than the bitstring composed of only 1. Figure 3d shows convergence of Known-Bitstring-Search as solved by the presented solver compared to One-Max.

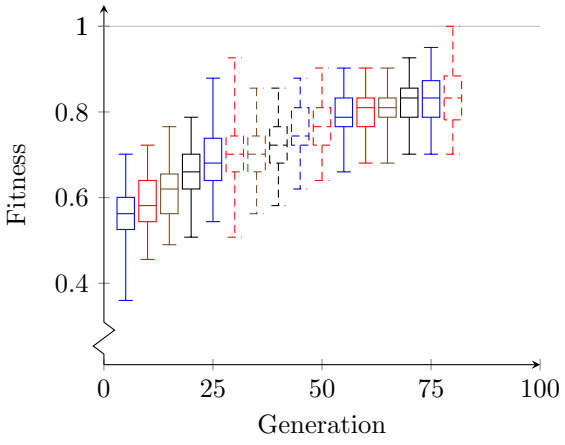
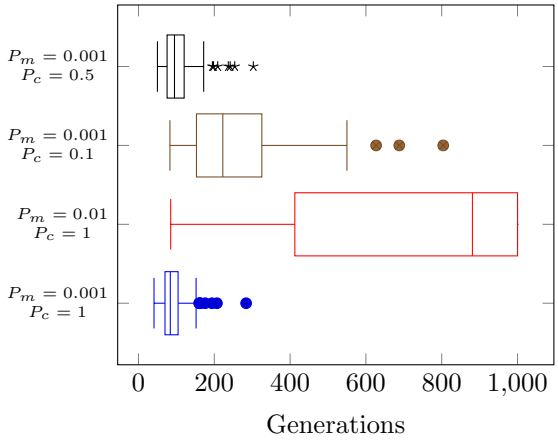
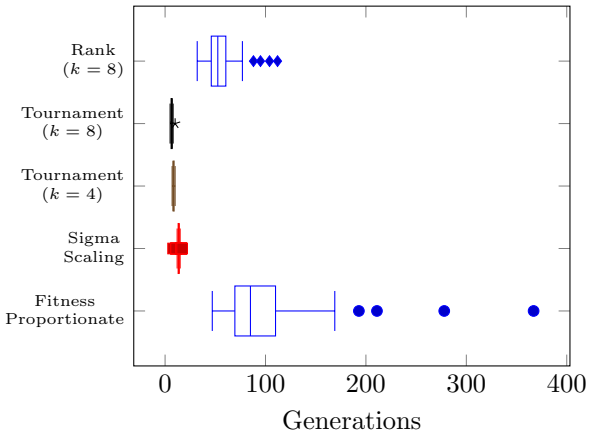


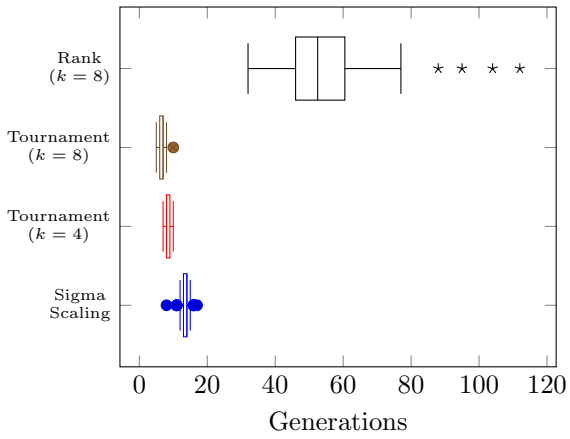
Figure 2. A single run of the solver solving a One-Max problem of size 40 using full generational adult selection, fitness-proportionate parent selection, per component crossover ($p = 1$), per genome component mutation ($p = 0.001$), and a population size of 150.



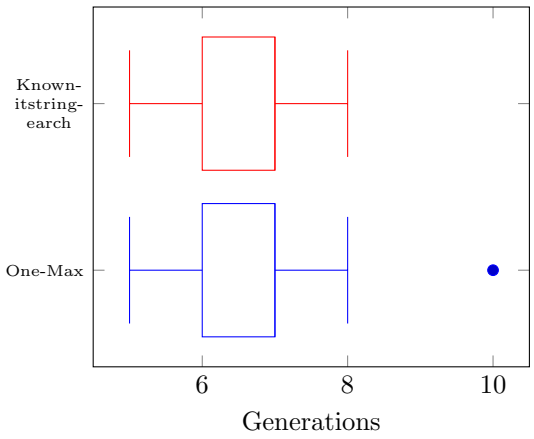
(A) Solution efficiency of the algorithm for different configurations of probability of mutation P_m, and probability of crossover P_c. The graph truncates at 1000 generations.



(B) Comparison of different selection mechanisms.



(C) Comparison of different selection mechanisms (zoom).



(D) Comparison of difficulty of Known-Bitstring-Search and One-Max. Both problems were solved with the same parameters as the Tournament Selection ($k = 8$) entry of figure 3a.

FIGURE 3. Box plot measurements of efficiency of solver runs mesured in generations until termination. Less is better. Whiskers represent the maximum and minimum measurement, the boxes contain measurements in the two center quartiles, and the center line is the median. Outliers are plotted as points. All measurements are measured over 100 runs of the solver for each case.