Sub symbolic AI methods: Project 2

Evolutionary Algorithms

Bjørnar Walle Alvestad

2016

Norwegian University of Science and Technology

Implementation

Description

The algorithm features two populations (or pools) of individuals, a child pool and an adult pool. The main evolutionary loop is implemented in the *EA* class. The loop itself is designed with simplicity and readability in mind, using easy to understand method calls. First, the child pool is filled with random individuals by generating randomized genotypes. The randomization is strongly tied to the genotype design, and so a randomization method must be implemented for each class of genotype. The generated genotype is further developed into a phenotype by utilizing a development method implemented in the phenotype class. Finally, the fitness value is calculated for the phenotype, and the final individual is added to the child pool. This is repeated until the child pool is filled.

Next, the adult selection routine is invoked. This method moves individuals from the child pool to the adult pool by the appropriate adult selection strategy. This is the only chance for a child to get promoted to the adult population.

The last step in the loop is parent selection. First, the algorithm determines the number of offspring to generate. This is influenced by population size and adult selection strategy, but for full replacement the number is equal to the population size. To generate n children, n parents are selected from the adult population, by the appropriate parent selection strategy. In sexual reproduction, two parents make two offspring, whereas in asexual reproduction, a single parent make one offspring. The offspring genotype, which is a copy of the parent genotype, is subject to mutation with a specified probability. Additionally, offspring from sexual reproduction is subject to crossover. The final offspring individual is assembled and added to the child population.

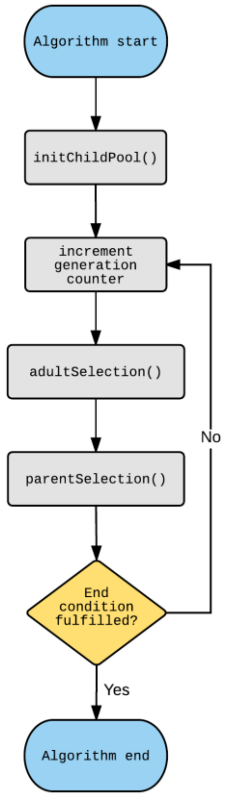


Figure 1: Flowchart diagram describing the main EA loop.

System modularity and reusability

The system incorporates binary, and integer array genotypes, with corresponding phenotypes. These can be used in a variety of problems, and can easily be specialized by subclassing. Problem specific operations such as randomization of genotypes, crossover operators and mutation operators can defined via heritance and polymorphism. These methods are defined in the abstract *Gtype* superclass, which essentially all genotypes inherits from. To change an operation, simply subclass and override the appropriate method. For example, to make a specialized version of the mutation operator and crossover operator for the binary genotype:

```
class SpecializedBinaryGtype extends BinaryGtype {
   ...
   @Override
   public void mutate(int mode, double probability) {
      // custom mutation operator code goes here
   }

   @Override
   public void crossover(BinaryGtype g) {
      // custom crossover operator code goes here
   }
}
```

The same goes for custom phenotypes; subclassing allows for customization of new phenotypes. The fitness evaluation function is defined in the phenotype superclass *Ptype*. To implement another fitness evaluation function, simply subclass and override the fitness evaluation method.

Adult and parent selection mechanisms are implemented in *AdultSelectionStrategy* and *ParentSelectionStrategy*, respectively. New selection methods must be added in the respective class, and a unique configuration id for the strategy must be specified in the *Config* class. The new selection mechanism can then be utilized by specifying the method name in the configuration file passed to the system at startup.

One-max problem

When running the 40-bit one-max problem with full generational replacement and fitness proportionate selection, the EA needed a population of 1800 in order to find a solution in under 100 generations consistently. The fitness for this run is presented in the left plot. When setting the crossover rate to 0.95, and mutation rate to 0.001, the system manages to find a solution in about 40 generations consistently (right plot). The crossover operator is one-point.

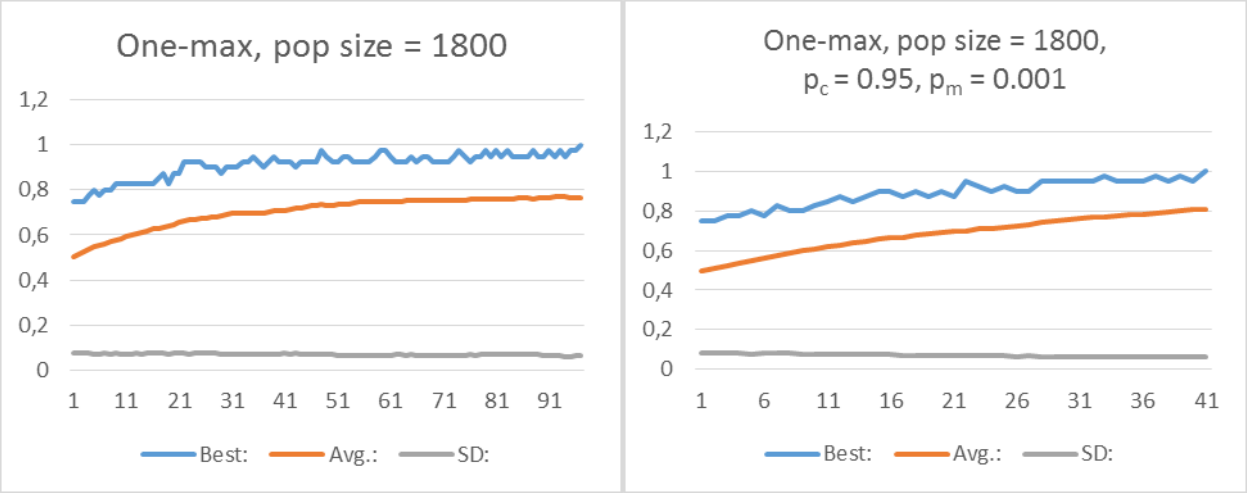


Figure 2: Plot of best fitness (Best), average fitness across population (Avg), and standard deviation (SD).

The best choice of parameters is to set mutation rate low ($p_m = 0.001$), and setting a high crossover rate ($p_c = 0.95$). This also allows shrinking of the population size down to 400 in order to solve the problem in under 50 generations.

The best parent selection strategy seems to be tournament selection, with a tournament size of 15, and $\epsilon = 0.8$. This allows the EA to solve the one-max problem in 5-8 generations. Sigma scaling yields good results as well, solving the problem in about 14 generations.

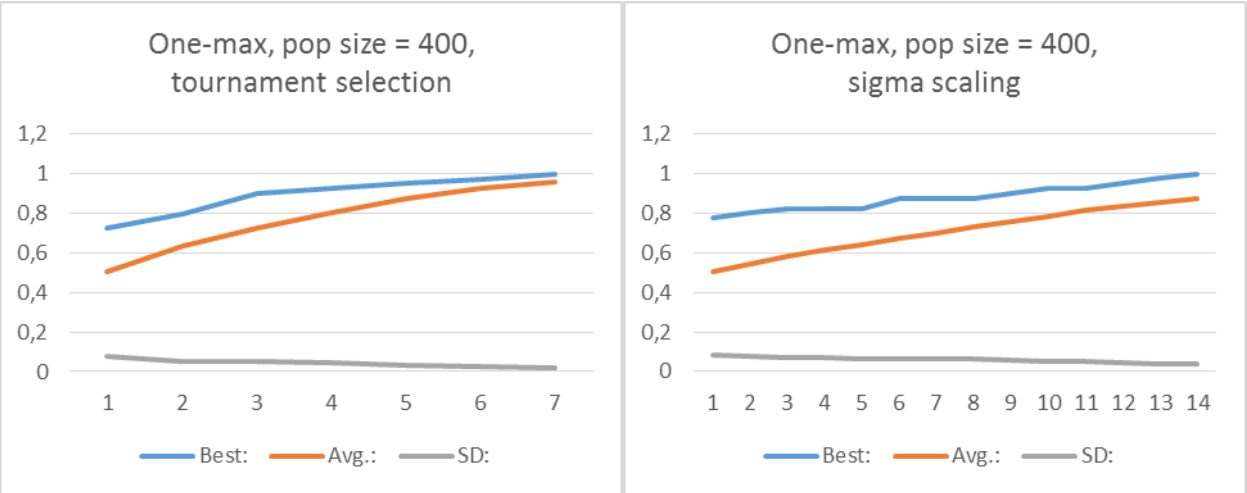


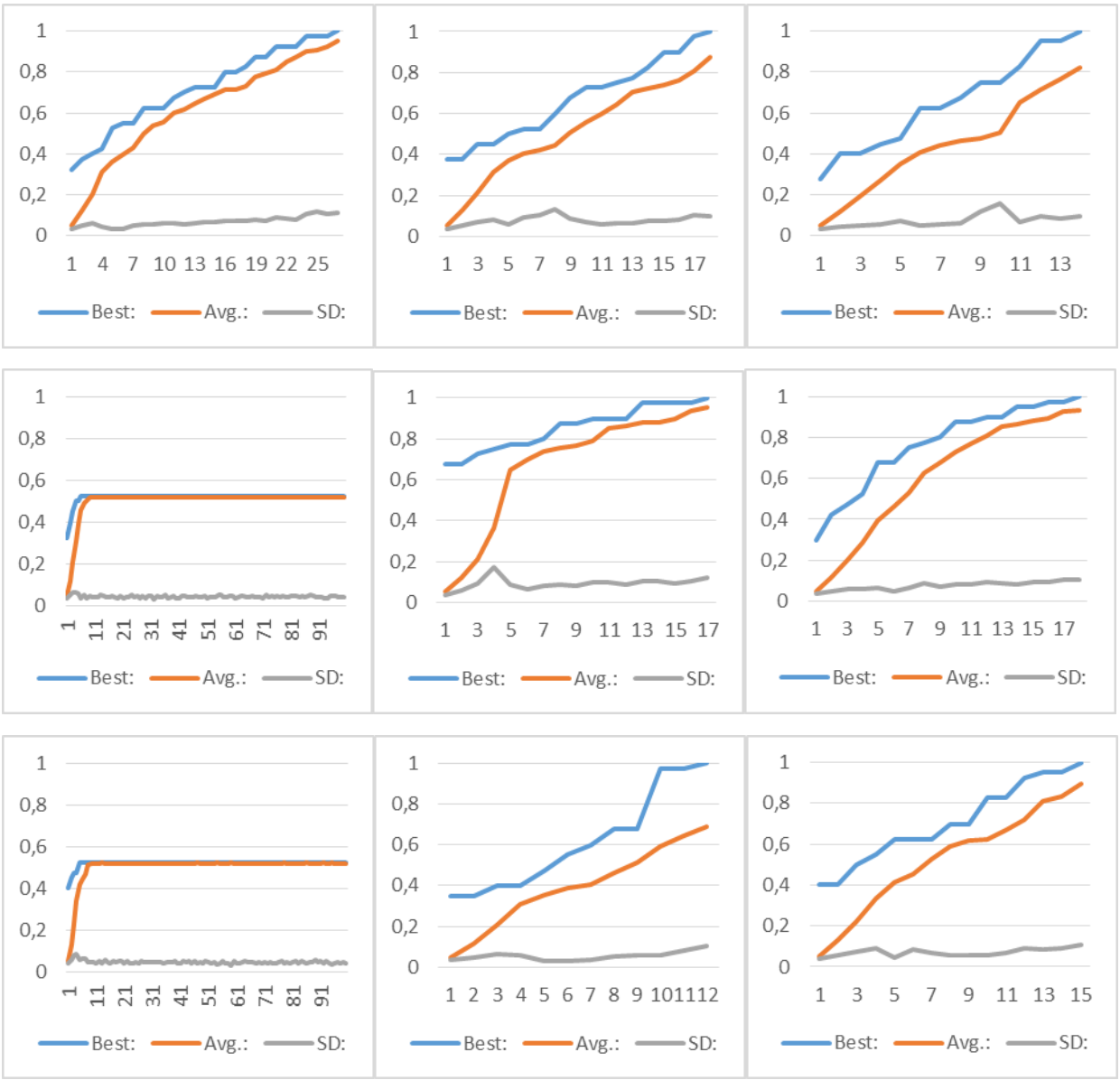
Figure 3: Plot of fitness values when using tournament selection and sigma scaling, respectively.

Using a randomly generated bit-string as target should not increase the difficulty of the problem, because after all, 111…111 is also just another permutation of n bits, just as any other random generated n-bit string.

Config	*Target = 111…111*	*Target = Random bit string*
Pop = 1800	94	96
Pop = 1800, $p_c = 0.95$, $p_m = 0.001$	41	40
Pop = 400, tournament selection	7	8
Pop = 400, sigma scaling	14	15

The table shows the number of generations needed to solve the problem with the specified configurations. The system needs an equal number of generations top solve the problem when setting the target to a random bit string. This strongly suggests that the problem does not get harder.

LOLZ prefix problem



The 40-bit LOLZ prefix problem is configured with the best parameters found in the one-max section: population size = 1800, mutation rate = 0.001, crossover rate = 0.95, tournament parent selection and tournament size = 15. Results from the 9 runs is presented above. A plot showing the best, average and standard deviation for each run.

The system found the optimal solution 7 out of 9 times. In the remaining two cases, the evolution converged to a local maximum, and was not able to solve the problem optimally. In a sub-optimal solution, leading zeros dominate the bit string, and the algorithm is not able to escape from this local maximum. By increasing diversity and reducing selection pressure, the system will be able to find the optimal solution of only 1's more often. Reducing the population size degrades the performance of the EA for this problem. That is why the population size is set to 1800 in this problem.

Surprising sequences

The genotype used in the surprising sequence problem is internally represented as an integer array with length L, where each cell contains an integer between 0 and S. The phenotype is also represented as an integer array, containing exactly the same values as the genotype. Because of this, there is no need for a development method. The values are simply copied from the genotype to the phenotype.

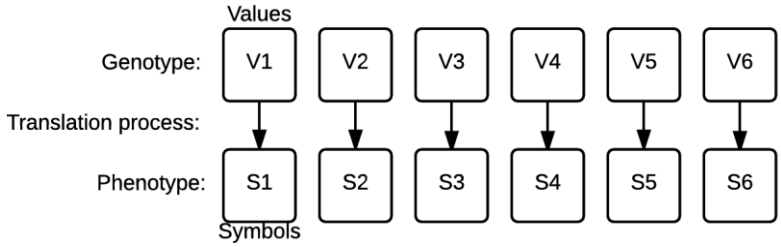


Figure 4: Illustration of the genotype and phenotype used in the surprising sequences problem.

Fitness evaluation is calculated by first generating all possible patterns (AX_nB, for all A, B and n) of the given sequence. The size of this collection (s_{tot}) is then recorded and stored. Duplicates are removed from the collection, leaving a set of unique patterns. The size of this set (s_{set}) is recorded and stored. s_{tot} can be calculated without needing to generate all possible patterns, using the sequence length l. Fitness can also be calculated by:

$$fitness = \frac{s_{set}}{s_{tot}} \qquad s_{tot}(l) = \sum_{n=1}^{l-1} n$$

Local runs:

S	L	Gen.	Pop.	Sequence
3	10	1	400	ACBAABBCCA
5	26	10	1000	EEABEDDECCEBCDCAADBBDACBAE
10	100	293	1200	CAJEIFFIJHHCJFHJAEBFBEDDFJJBGIGDBBIHIBAIEHBHDCIIDJGGBDICEGAAGCCGJDHACBJIADGHEAHFABCFCDAFDEFGEECHGFEJ
15	220	836	1400	CLFDKCAJJHBMGGMCKACEIFHLOKOGEALBGCBHHJIODMBAKIBKFAADBBCNNEJGJBFMMOBDLIDJNGONKMEFKKLKNDDGBJCGIKJECOFGFOAOINIILDEEGLNOHIEKHOCHGKDOJKBOOLHKEMDNBEOEBNLJDFBIGDIANCIJLELGHMNMHEDCDAFFLLABLMJFCJMIHFIMKGAHNHDHCMLCFEHAENFJAMFNAGNJ
20	385	5744	120	FLDCCBLTOMMHMGQJRNPQANESTGNAACNFQGEMCKNDQSDDHCEKQNNLJSROTBNMBHTIQEFTLPSJMITAMTMDGGHABSCOAONJLLCGIHJFDIBJPLOPATPDFBOKBEAICTDNHFPEOQHRQFIPNSLASNTSHHDSAFKPBBDEPGTKJCJJTHGDOHIMOIGALBIFMKFJQPRTNOBFRSBRJNBAJENCLNRRDBTJAKREQDTEGLGOOCRBQOECPHQKKDMFHOJKSIRARGRPFOFNQMNINKMAQCILMSSKTRKOLEIORIKCSMRFGPCHPPJBGFCDJIIDAPKGKHBKAHKLIETQIJGSQTFAGBPISOGJDRMELQQBCFSEERHNGCAEDKEBMPOSGMQRLHSFFEHEJODPTCQLS

Global runs:

S	L	Gen.	Pop.	Sequence
3	7	1	120	CBACABB
5	12	7	120	ACDEEABEBDCA
10	25	31	200	EBIECFHAGDFJJHCDIBABEGACI
15	39	1939	200	ILOGHMKGDAFJMNEBIBAKCCJLIDEONKFMOHNGALG
20	51	10026	120	CODIFFNGSQPKLERJTGJHMHQAIDBAMLTONCPBTQOMKRGFEKPDSCN

EASY	One-Max. There are no local maxima to for the EA to get trapped in. Very straight forward evolving.
MEDIUM	LOLZ. One obvious local maximum. Not very hard to avoid, just run the EA multiple times.
HARD	Local surprising sequences. Many local maxima. EA needs to explore to find optimum solution.
HARD	Global surprising sequences. Even more local maxima, but L is lower bounded than local.