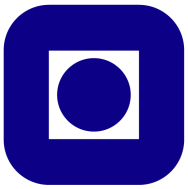

NTNU

IT3708 - Sub-symbolic AI Methods

Project 2: Programming an Evolutionary Algorithm

Mathias Ose

March 11, 2015

Implementation

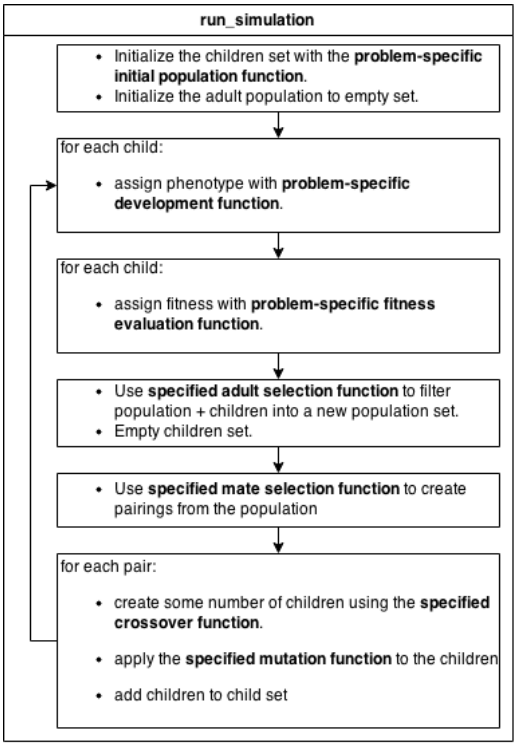

Figure 1: Implementation outline

Python 3.4 was used to implement this project. For plotting results I used the library *matplotlib*.

I tried to stay close to the EA flow as illustrated in Figure 1 in the EA appendices[?].

Problems are Python modules that contain functions for generating initial populations, development of phenotypes from genotypes, and fitness evaluation.

The problem independent functions are defined as standalone functions that get all the information they need via parameters.

Since the project emphasises modularity, I tried to use generic functional programming as much as possible.

The evolutionary loop function requires a configuration object to run, which specifies all the details of the simulation, including the problem module itself, numeric parameters, and the various other functions. If new functions are defined they can be loaded into a configuration with a simple import statement.

Listing 1: Creating a new problem type.

```
NAME = 'IS_EMPTY_STRING'

def generate_initial_population(
    population_size, genome_size)
    :
    return tuple(random_string(
        len=genome_size) for _ in
        range(population_size))

def geno_to_pheno(genotype):
    return genotype.split()

def fitness_evaluation(pheno):
    if len(pheno) == 0:
        return 1.0
    else:
        return 0.0
```

Listing 2: Plugging in a different mate selection function.

```
config.update({
    'mate_selection_method':
        some_function_reference,
    'mate_selection_args': {
        'parameter_a': 10,
        'parameter_b': "Foo"
    }
})
```

Specifics such as the formats of geno- and phenotypes or fitness values are not the concern of the loop itself, only the functions that are plugged in.

The simplest user interface for the program is to use a text editor to create a configuration in a `.py`-file and call the `run_simulation` function with it. Another option is implemented in `main.py`: a command line interface that allows the user to choose a preset, then modify the parameters before running the simulation.

One-Max problem

To find a minimal population size I created configurations that differed only by population size, ran them 1000 times each, and plotted the generation numbers where an optimal solution was found in a his-

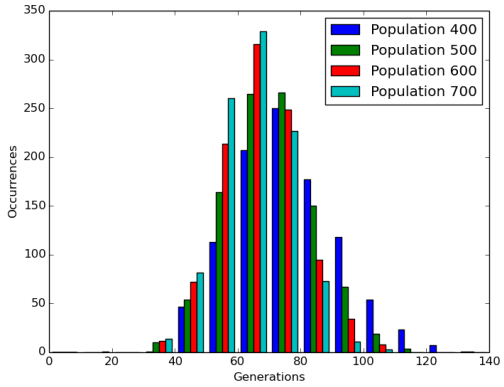

Figure 2: Generation number where an optimal solution is found. 4x1000 trials.

togram. Using this I visually identified 600 as a suitable value. At that size there were only a few (1.3%) outliers that exceeded 100 generations.

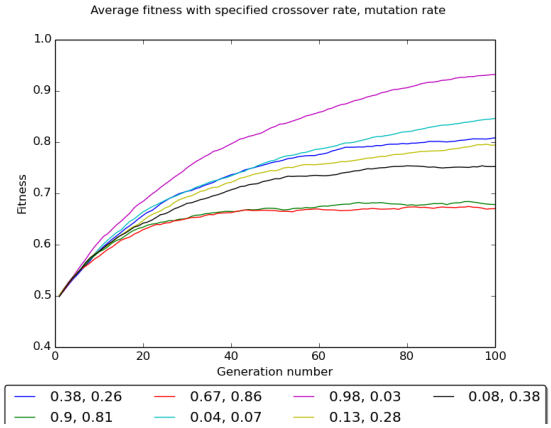

Figure 3: Averaged graph for 10 runs of each configuration

Different rates

To figure out the crossover and mutation rates I started out by generating random values, running simulations with those values 10 times in a row and averaging the lines for each configuration. The plot of this can be seen in Figue 3.

From the results it can be seen that it is not beneficial to have a high mutation rate. It is clear that it is the crossover that is supposed to steer the population in the right direction, the mutation is just there to shake things up a little bit.

Based on this I decided to proceed with a high crossover rate of 0.90 and a very low mutation rate of 0.01 per genome.

Parent selection

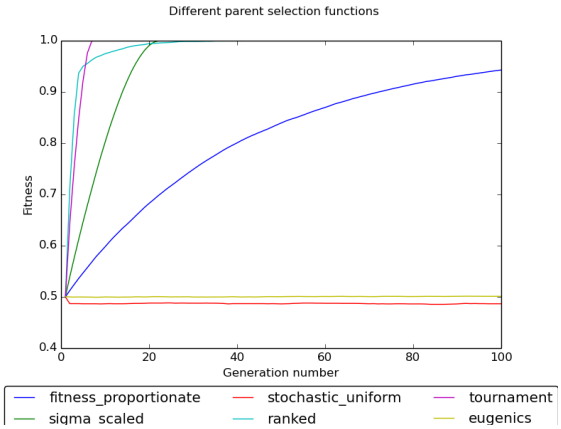

Figure 4: Testing different parent selection functions.

When testing different parent selection methods I again averaged multiple trials for each type. As Figure 4 shows, the stochastic uniform and "eugenics" methods failed to produce better individuals. The fitness proportionate method worked fine but slowly, and by 100 generations was getting close to an optimal solution. But three functions performed significantly better. The ranked roulette method got off to the best start but struggled to find the last 5% for an optimal solution. The tournament method is the best in this test, it found an optimal solution within 10 generations. The sigma scaled method worked simillar to the tournament, but at a slower rate.

Different target

Modifying the target string means modifying the fitness evaluation function to correctly evaluate the candidate vs. the target string. As long as this is done correctly, nothing about the difficulty will change. The recombination and mutation

function will not at all be affected by this change.

I tested this statement by creating two configurations that differed only by target string and averaged the plots for 100 trials of each. The results was two practically identical lines (figure 5).

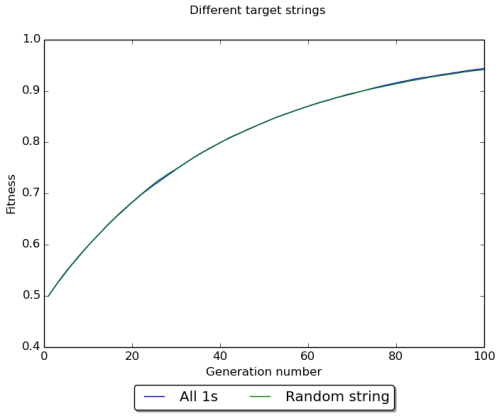

Figure 5: 100 averaged trials of searching for all 1s, and searching for a random string. The lines are practically identical.

Changing the problem

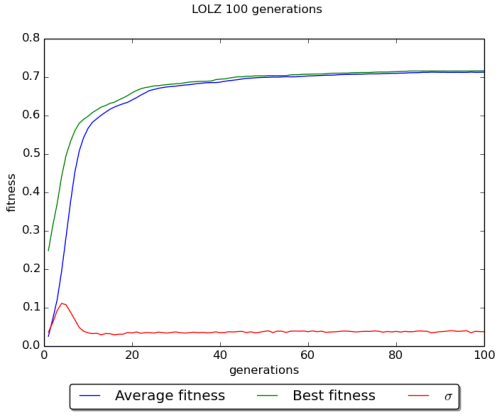

Figure 6: Average 100 trials of LOLZ z=21

When running the same configuration for the LOLZ problem, the average of multiple trials flattens out somewere between 0.5 and 1.0 (figure 6). This is because in some trials, the population moves in the direction of the local maximum of 21 zeroes. If this happens the population gets stuck at a max fitness level of $21/40 = 0.525$. In other trials the population starts heading towards the global maximum and reaches average fitness of 1.0. Since there is a 50/50 chance of the population going for leading ones or leading zeroes, the average eventually line positions itself between 1.0 and 0.525.

Surprising Sequences

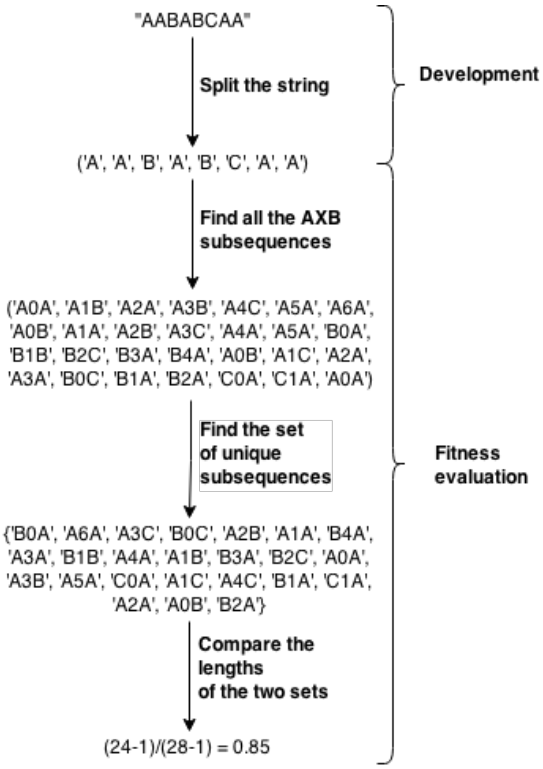

Figure 7: Example of how a genotype is processed.

Working with the type system in Python shaped the formats of geno- and phenotypes in my implementation greatly. I went with more high level representation of the types than perhaps intended. In the case of the Surprising Sequences problem the genotypes are regular Python strings composed of the characters in the character set of size S. Phenotypes are just the list version of the same string.

I considered doing more operations during the development function, specifically finding the AX_nB substrings at that

point, but ultimately felt it was more appropriate to let that be the job of the fitness function.

The fitness function scans through each phenotype to find all the subsequences of the format AX_nB. For finding globally surprising sequences it will find the subsequences for any n, for locally surprising sequences it will stop after $n = 0$. It then compares the number of *unique* AX_nB subsequences to the total number of subsequences.

$$f = \frac{|\text{unique } AX_nB\ | - 1}{|\text{total } AX_nB| - 1} \quad \text{for } L > 2$$

This will give a score of 1.0 if the sequence is completely surprising. For locally surprising sequences it will score 0.0 if the sequence is just the same character repeated, while that sequence will get a low but non-zero score when evaluated for globally surprisingness.

S	Population	Generations	L
3	50	1	10
5	50	383	26
10	50	876	94
15	50	620	196
20	200	548	343

Table 1: Longest L found in less than 1000 generations, local surprising

Difficulty

One-Max is the easiest. In the solution space there are no local maximas, every increment leads towards the single global maximum. Combining two fit parents is likely to produce a child that is also very fit.

LOLZ is more difficult than One-Max because the z parameter means there exists a local maximum. If the population starts trending towards that maximum it is unlikely that they will ever find the global maximum, as the sub-population with leading zeroes will outcompete and eradicate the subpopulation with leading ones.

Another issue is that combining two parents with simillar fitnesses is not guaranteed to produce a fit individual at all, since the two parents might be fit in different ways. There is also the fact that an individual like "0111111" has the potential to be the parent of a very fit individual, but is itself very unfit and therefore might not get to mate at all. These are not problems that occur in One-Max.

The two Suprising Sequences problems are the most difficult, with globally suprising being even more difficult than locally surprising, since a globally surprising sequence must satisfy the criteria for local surprisingness and then some.

Unlike the previous problems where the fitness topology has only one global maxima (and one local maxima in LOLZ) there are many global and local maxima for the surprising sequences. Finding the correct path to a global maximum is not easy, and most trials become "needle in a haystack"-situations. It is also more difficult to tell if a random mutation is beneficial or not, since each genome component factors into the fitness in so many ways.

S	Population	Generations	L
3	200	1	7
5	200	6	12
10	200	748	24
15	200	176	36
20	200	81	46

Table 2: Longest L found in less than 1000 generations, global surprising

References

[1] Anonymous. EA Appendices, January 2014