IT3708 SUB-SYM AI-METHODS

Project I-A: Programming the Basics of an Evolutionary Algorithm(EA)

Participants: Sondre Lucas Follesø

Introduction

This report describes an implementation of an Evolutionary Algorithm with the goal to solve One-Max problems. The implementation is done in C# and the code is made with a focus on reusability for other problems. The report will focus on the degree of reusability of code and the algorithms ability to solve a 40-bit One-Max problem using different input parameters.

The Code:

Figure 1 shows a class diagram for the EA implementation. It also contains some classes from the second part of the project, the Colonel Blotto simulation. The implementation is divided up into groups of classes based on the inheritance. As can be seen, the code includes all the modules of a EA, including genotypes, phenotypes, genetic operators, fitness evaluators, development methods and selection mechanisms.

Genotypes and Phenotypes

Genotypes are translated into phenotypes using problem specific development methods. There is only one genetic operator in this implementation, because the project only uses binary genotypes. The One-Max problem only uses binary arrays as phenotypes, and the phenotypes are therefore only an object that holds its genotype.

Evaluators

The One-Max fitness evaluator holds the goal vector which each phenotype tries to reach, and calculate the fitness based on the similarities of the bit-vector genotype that the phenotype holds, and the goal vector.

Population

The binary population class is an object that is responsible of initiation a population, holding the list of adults and offspring for each generation, and perform adult selection based on the selection protocol chosen.

Genetic operator

The binary genetic operator implemented in BinaryOperatos.cs does the job of crossovers and mutation. Crossover is implemented as random one-point crossover, and mutation is random one-bit mutation.

Selection mechanisms

The selection mechanisms implemented are Fitness-propagation, Rank, Sigma Scaling and Tournament. They all inherits for AbstractSelectionMechanism.cs which holds the code for parent selection for crossovers.

The main algorithm class is OneMax.cs which inherits from AbstractEA. AbstractEA holds the main components of a Evolutionary Algorithms as objects, and has methods for generating offsprings and performing a cycle of evolution. OneMax.cs holds some more problem specific code and method for running a desired number of generations.

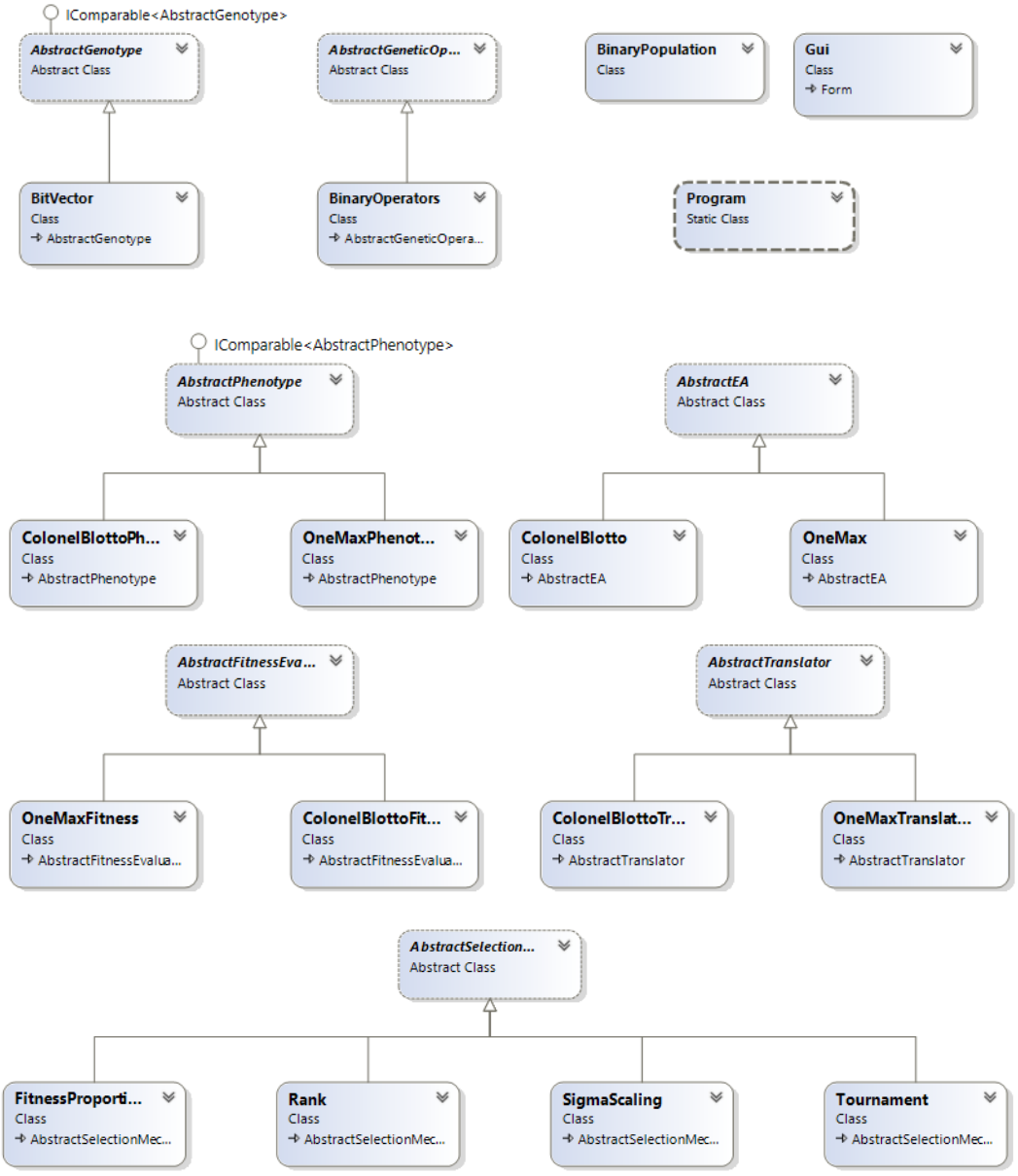

Figure 1: Class Diagram

Justification of modularity and reusability

Modularity and reusability is achieved by the use of abstract classes for all component that should be modular. Selection mechanism, phenotypes, genotypes, fitness evaluator and

development methods all have abstract classes that contains the essence of the component, and abstract methods that needs to be coded in any sub-classes.
For example, to make new phenotypes, simply make a class which extends AbstractPhenotype, this new class will then already hold a AbstractGenotype, fitness-value and a value for the probabilty to be chosen in parent selection.

```
public abstract class AbstractPhenotype:IComparable<AbstractPhenotype>
{
    public AbstractGenotype Genotype { get; set; }
    public double Fitness { get; set; }
    public double RouletteProportion { get; set; }

    public int CompareTo(AbstractPhenotype obj)
    {
        return Fitness.CompareTo(obj.Fitness);
    }
}
```

Figure 2. AbstractPhenotype.cs

Analysis

Starting with a population size of 100(figure 3.), mutation rate of 0.1 and crossover rate of 1.0, i worked my way to population size which shown to pretty much consistently be able to find the solution of a 40-bit One-Max problem. I landed on a population of 225(figure 4). The maximum fitness of runs with population size 100 was very close to 1 several times, but very rarely did it find the solution. The algorithm began to consistently solve the problem when going over 200 in population size. Fitness increase stagnates when phenotypes is are closing in on solutions. Leaving the phenotypes with small chance of mutation and only crossover with other phenotypes with very similar genotypes makes it hard to evolve the last few steps.
Increasing the mutation rate decreased the chance for finding the solution. This is probably because when the fitness stagnates do to crossover limitation, the mutation mostly mutates bits that are already 1 back to 0. Decreasing the crossover rate made the same impact.
The best settings shown to be the initial settings: population size of 225, mutation rate 0.1 and crossover rate 1.0 (58 gens)

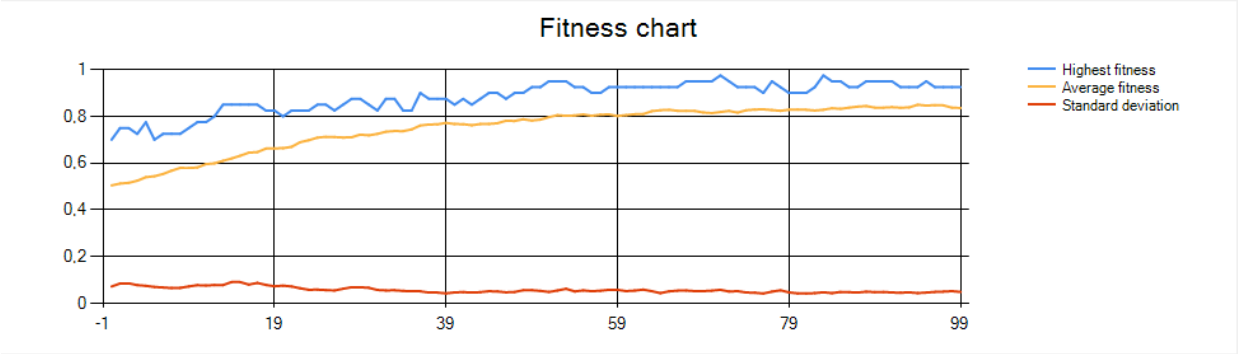

Figure 3: Population size of 100, Mutation rate 0.1 and Crossover rate 1.0

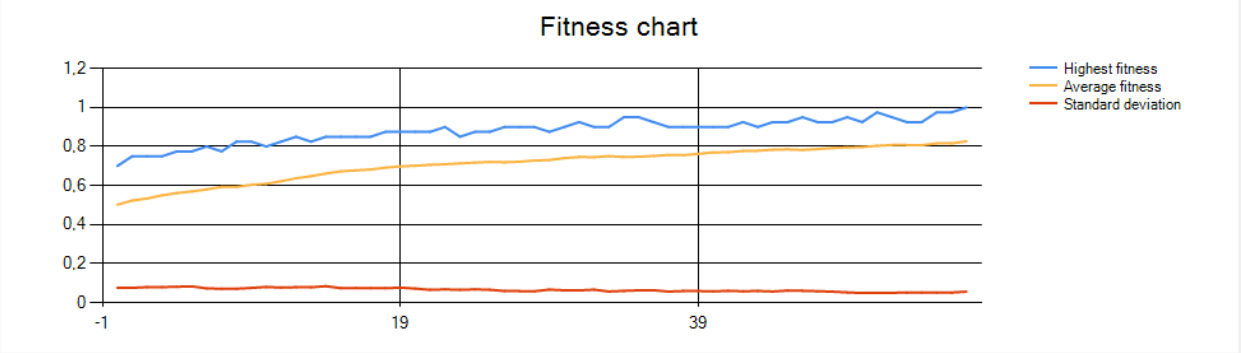

Figure 4: population size of 225, mutation rate 0.1 and crossover rate 1.0

When changing the parent selection mechanism, Sigma scaling(Figure 5.) shown to be the clear winner, reaching the solution consistently under 20 generations. Here the best phenotype quicly take over the whole population. If the population converge into one fitness, then mutation is needed to take the population further in evolution.
Tournament(Figure 6.) followed with right over 20 generation as the most usual result.
Rank(Figure 7.) usually ended at around 45 generations, leaving Fitness-proportionate the worst of the four.

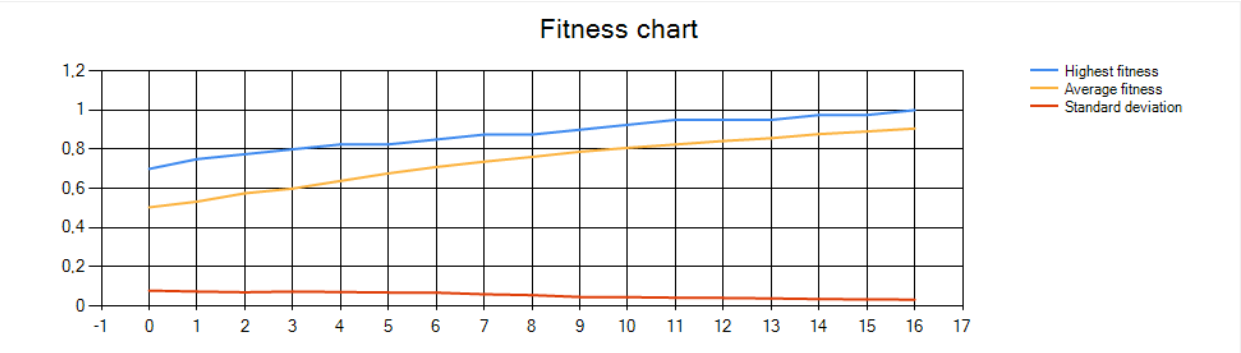

Figure 5: Sigma scaling

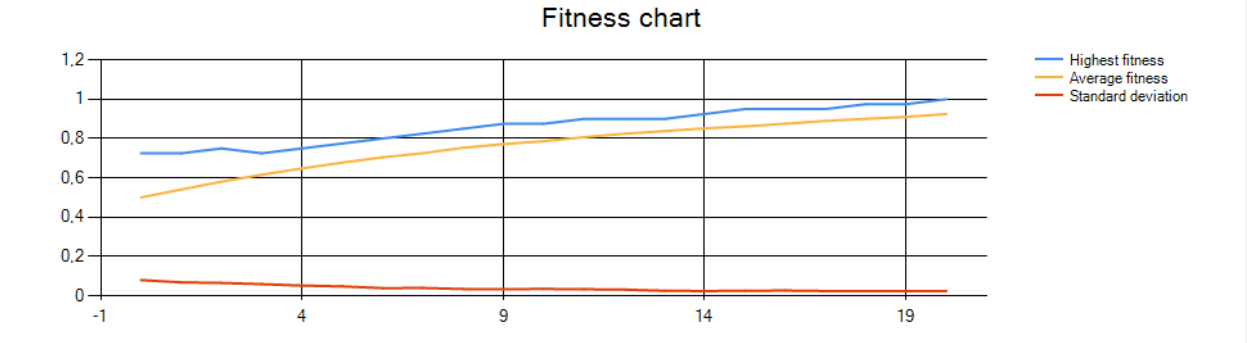

Figure 6: Tournament

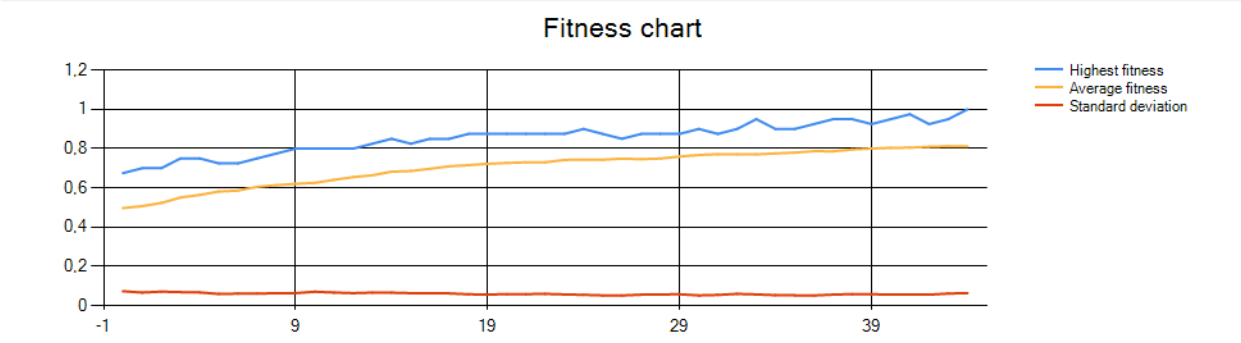

Figure 7: Rank

Running the algorithm to find random binary vectors instead of only 1's should not make any difference in performance, since the probability that a bit is correct is the same as before(50/50 chance). Four runs, using the same parameters as before, with the four different selection-mechanisms, shows each selection-mechanism performs about the same as with goal vector of only 1's(Figure 8-12.).

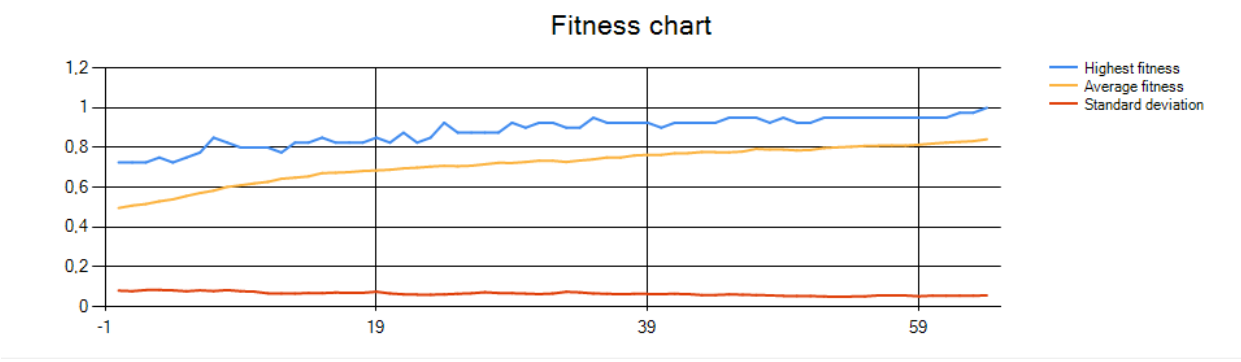

Figure 8: Fitness-propagation - random vector.

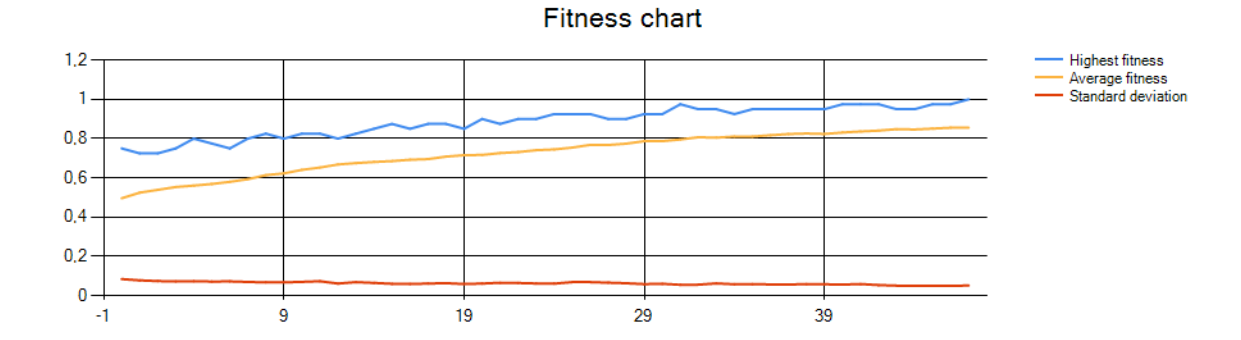

Figure 9: Rank - random vector

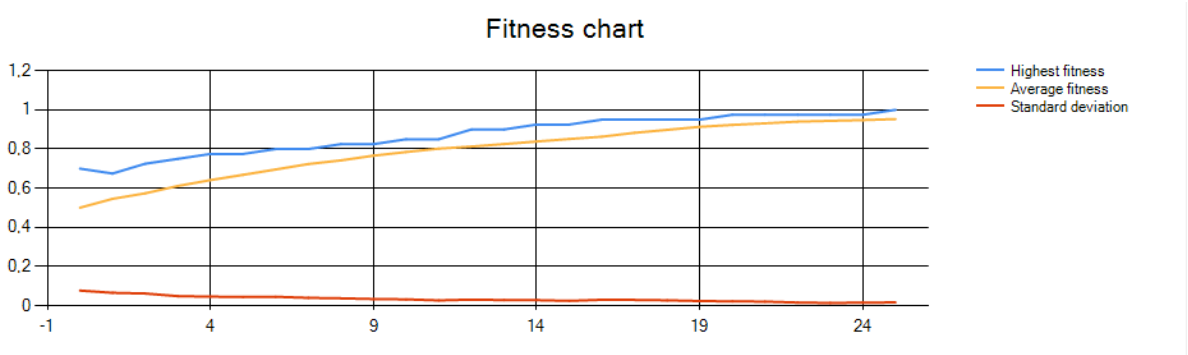

Figure 10: Tournament - random vector

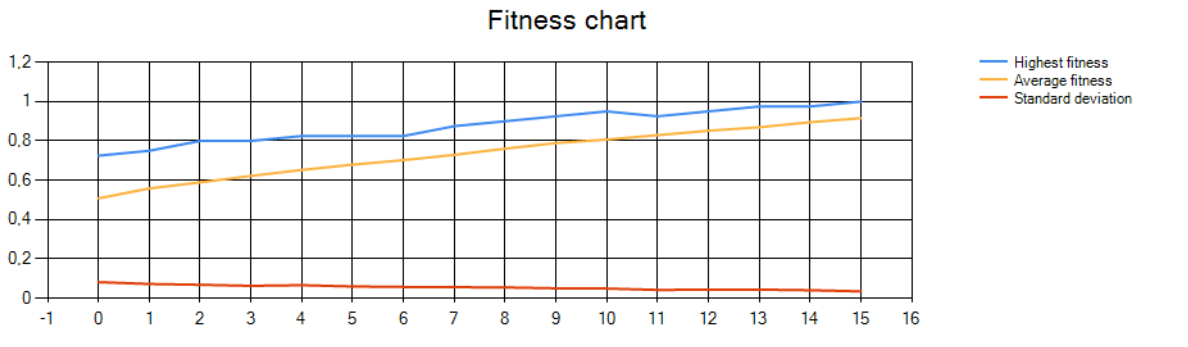

Figure 11: Sigma - random vector

Conclusion

By experimentation with different rates of mutation and crossover, as well as different selection-mechanisms for parent-selection, i found that Sigma scaling, with high crossover rate and low mutation rate, is the best at solving the OneMax problem.

IT3708 SUB-SYM AI-METHODS

Project I-B: Evolving Colonel Blotto Strategies using a Genetic Algorithm

Participants: Sondre Lucas Follesø

Introduction

This report describes the adaptation of an previously implemented evolutionary algorithm, to evolve strategies for resource distribution in Colonel Blotto games. The population will consist of strategies that will battle each other, and reproduce to form new interesting strategies. This report will discuss the significance of strategy entropy. There will also be results and data from 18 different runs, of which three will be investigated further.

The code

The implementation is done in C#, and extends a previously developed evolutionary algorithm, using binary genotypes. All classes added to the previous system is prefixed with the string “ColBlotto”
For more information regarding the algorithm, see the report from Project-1A.

Genotype representation and translation

The genotype is represented as a bit-vector, which length depends on the highest amount of resources one battle can consist of, and the number of battles for each strategy . In this project each battle would be initialized with a randomly generated number between 0 and 10. The binary representation of “10” is “1010”, and each genotype is therefore represented by a vector where the number of bits is equal to 4 times the number for battles in a strategy.
When translating from genotype to phenotype, four and four bits are converted back to their integer value and divided by the sum of all the battles. This makes all battles a fraction, which sums to 1 and gives all strategies the same amount of resources to spend, independent of the random generated numbers.

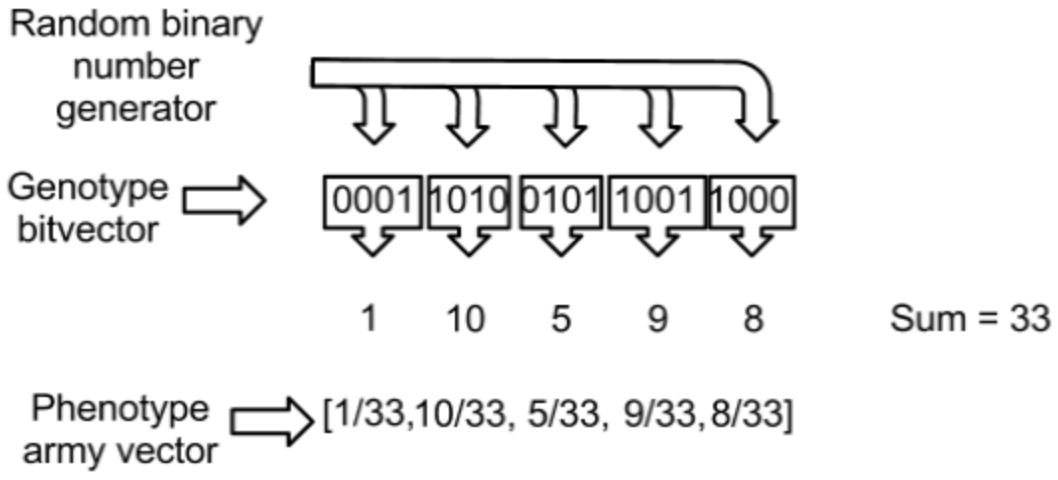

Figure 1: Genotype to phenotype translation

Strategy Entropy

The entropy of a strategy is denoted by H(pi)=-i=1Bpi*log2(pi), where B is the number of battles, pi is amount of resources devoted to the i'th battle. This means that the entropy will be highest when the number of resources is evenly distributed amongst the battles in a strategy. In turn, the entropy i low when all the resources are placed in one battle. What this means is that specialized strategies(where most of the resources are deployed to a few battles) will have less entropy than generalized strategies(where resources is evenly spread out). Strategies where the entropy is low has evolved into concentrating on few battles, and this may be a sign that it has "learned" the importance some battles over others.

EA parameters

I used the a low average strategy entropy as goal when experimenting with the EA parameters. Letting the algorithm run for 200 generations i started my search with "Full generation replacement", Fitness-proportionate as selection-mechanism, and crossover and mutation rate at the two "extremes", 1.0 and 0.0 respectively. Replacement fraction and loss fraction is set to 0, to simulate a normal Colonel Blotto game. This gave strategies that ended up only tying each other after about 50 generations. The entropy was consequently over 2.0 every generation.
Changing the selection-mechanism only made the algorithm converge even sooner.
Increasing the mutation rate to 0.1 broke the conversion, but the entropy stayed about the same.
Increasing the mutation rate further only gave more spikes in max fitness, but did not change the entropy,
By lowering the crossover rate i was able to lower the average entropy to numbers under 2.0, but was unsuccessful in making it consequently lower.
Table 1 shows the parameters of one of the best runs, where entropy was mostly around 1.8, and at some periods as low as 1.5. This shows that there was more specialised strategies in this run.

Population size	20
Number of battles	5
Generations	200
Mutation rate	0.1
Crossover rate	0.5
Selection protocol	A-I
Selection mechanism	Sigma Scaling

Table 1: EA parameters

18 Base runs

Table 2 shows 18 base runs with different Colonel Blotto specific settings. Each run is done with the previously found EA parameters.

Run	Number of battles	Replacement fraction	Loss fraction	Short note on entropy and “best” strategy
1	5	1	0	Winning strategy focusing on battle 1, 4 and 5. Average entropy is low
2	5	1	0,5	Very low entropy, winning over 50% on first battle
3	5	1	1	Winning strategy is focusing on the first two battles. Low entropy
4	5	0	0	High average entropy
5	5	0	0,5	Winning strategy is focusing on the first and one random battle
6	5	0	1	Very low entropy(around 1,0). same as run 2
7	5	0,5	0	High Average entropy
8	5	0,5	0,5	High Average entropy
9	5	0,5	1	Low Average entropy. Winning strategy focusing on the first two battles.
10	20	1	0	Entropy stayed around 4.0
11	20	1	0,5	Entropy at around 3.7. Focus first two battles.
12	20	1	1	Focus on battle 1 and 4
13	20	0	0	Entropy at 3.5. Focus on battle 6 and 16
14	20	0	0,5	Entropy at 3.5. Focus on several battles, and other battles are given 0 resources
15	20	0	1	Equal to run 14

16	20	0,5	0	Focus on battle 1, 16 and 17.
17	20	0,5	0,5	Equal to run 16
18	20	0,5	1	Focus on the first and latest battles.

Further discussion of run 2, 6 and 16

Out of the 18 runs I picked run 2(Figure 2), 6(Figure 3) and 16(Figure 4) for further inspection. I found run 2 and 3 most interesting. The entropy in run two quickly descended to a record low of under 1.0 in average entropy, and towards the end it got even lower. The winning strategy towards the end putting over 50% of its resources into the first battle. The same happened in run 6. But here the maximum fitness was mostly a small % above the average fitness, meaning a high degree of similarity in the population. Run 2 is a run where large resource amounts in early battle gives advantages both in replacement and in minimal strength loss, so its important to have large amount early. The same is true for run 6, where a lost battle in the first battle will make your army useless late on.

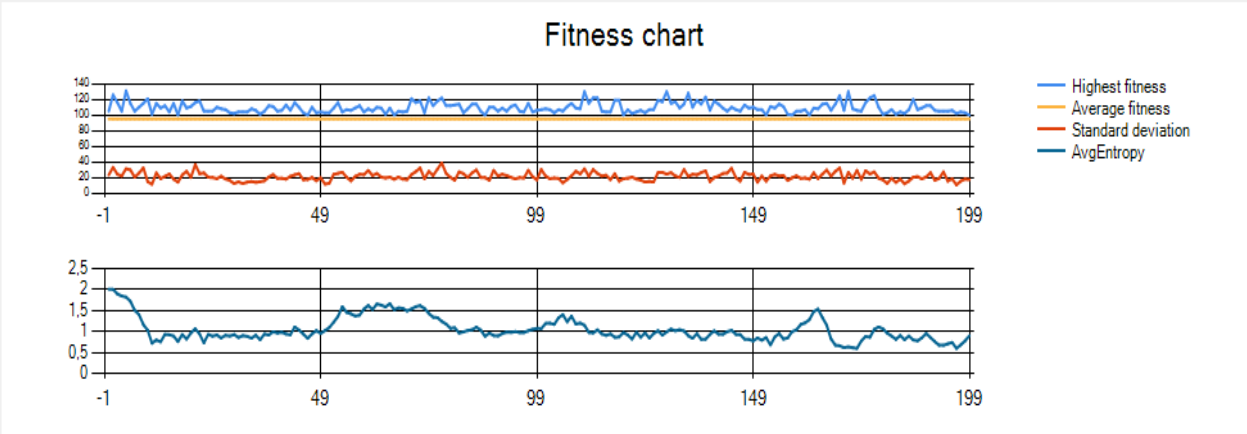

Figure 2: Run 2

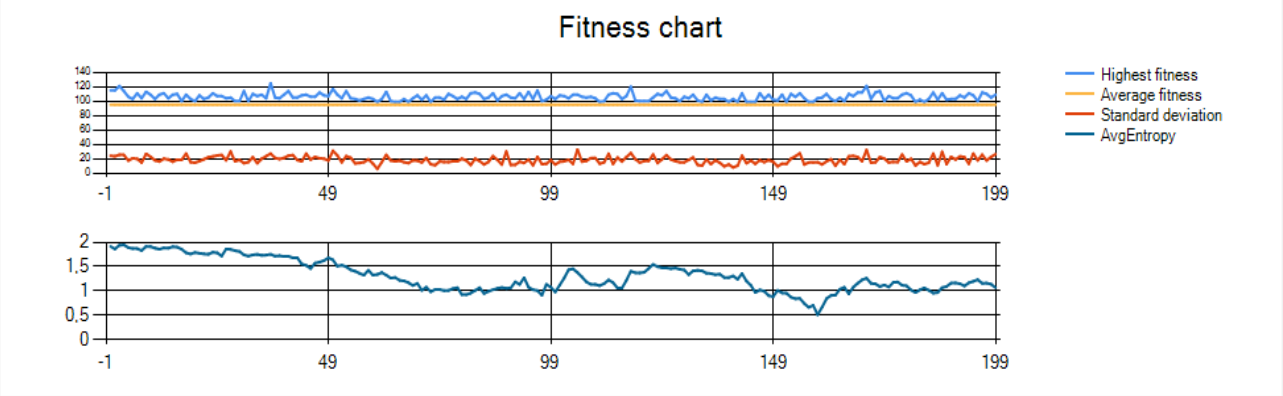

Figure 3: Run 6

It was more difficult to find very interesting trends in the runs with 20 battles. The high number of battles made the genotypes so large that the one-point random crossover operator and small chance for mutation does little work. Figure 4 shows how the population is close to converging, probably kept away by some genetic operations.
Since there are so many battles to distribute the resources it's important to keep the army strength up through the whole war. The focus on the first battles does this. It also gives the opportunity to relocate soldiers to the late battle, but since there are so many battles, the amount given to each battle is pretty low. A good strategy is therefore as the winning strategy does here in run 16, where the focus is on early and late battles rather than the ones in between.

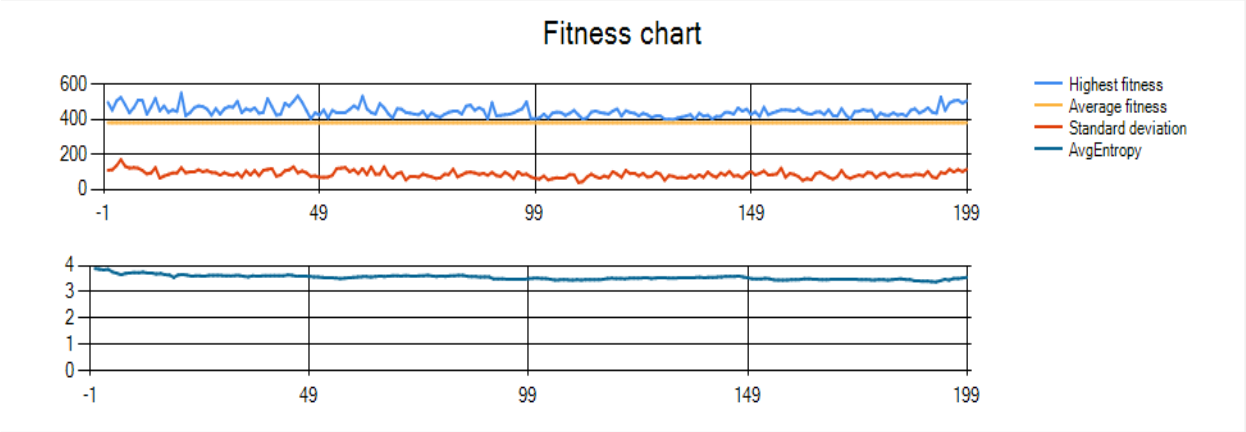

Figure 4: Run 16

Conclusion and Coevolution discussion

Its clear that the algorithm works almost as expected, given that some strategies evolve into very specific strategies when confronted with the choice of placing large amount of resources early to gain replacement resources or simply to keep moral high in the army.
I've learned that in coevolution, the population will stagnate if left with no chance of crossover or mutation. The fitness is not as important a factor towards a good solution, and entropy tells us if a strategy or population of strategies is specialised or general.