IT3708

Subsymbolic Methods in AI

Assignment 1

Programming the Basics of an Evolutionary Algorithm & Evolving Colonel Blotto Strategies using a Genetic Algorithm

Kjetil Valle
kjetilva@stud.ntnu.no

Olav Bjørkøy
olavfrih@stud.ntnu.no

NTNU, Trondheim

February 8th, 2010

In this report we describe our EvoSim *framework for Evolutionary Algorithms, and its application to the* One-Max *problem. The structure of the framework is discussed, with focus on how it is designed with flexibility, modularity and task independence in mind, and how it can easily be extended to solve new problems. We then look at the* Colonel Blotto *problem and how to solve it using the framework by implementing only the domain-specific classes needed for this task.*

1 Programming the Basics of an Evolutionary Algorithm

Our application consists of two main parts: A general framework and a set of domain-specific classes tailored to each assignment. Our goal was to create a framework that could be used for any task suited to be solved using evolutionary algorithms. This will likely save us a lot of time in the remaining assignments.

We designed the framework so that it should not be necessary to change any of the general EA code, no matter the assignment. Each new problem should only have to implement domain specific classes such as `Environment`, `Developer`, `Genotype` and `Phenotype`, and specify its particular configuration of the framework in a settings file.

We have chosen to code our EA framework in Ruby. This language provides a lot of possibilities in object oriented programming, dynamic loading of classes, flexible abstract classes and inheritance, which suits this assignment well.

1.1 Framework Description

The class diagram for the framework is shown in Figure 1.

The access point when using the framework is the file `ga.rb`. This file, when run, bootstraps the framework, executes the task and reports the final result (i.e. the best solution). Which task to perform can be specified as an input argument, or set in the file itself. The settings used when executing the current task is given in the different `YAML` files in the `settings` folder. `General.yml` contains the framework wide default settings, and explains which properties can be set.

On the topmost level of the framework is the `EvoSim` class. This is the puppeteer for the `Population`, and contains the *evolutionary loop*. It directs the generations through adult and parent selection, breeding and testing of fitness, by instructing the population class.

Parent selection and *Adult selection* is handled by the `Selector` class. Parameters such as population size, elitism, generational replacement and selection mechanism are determined by the current settings.

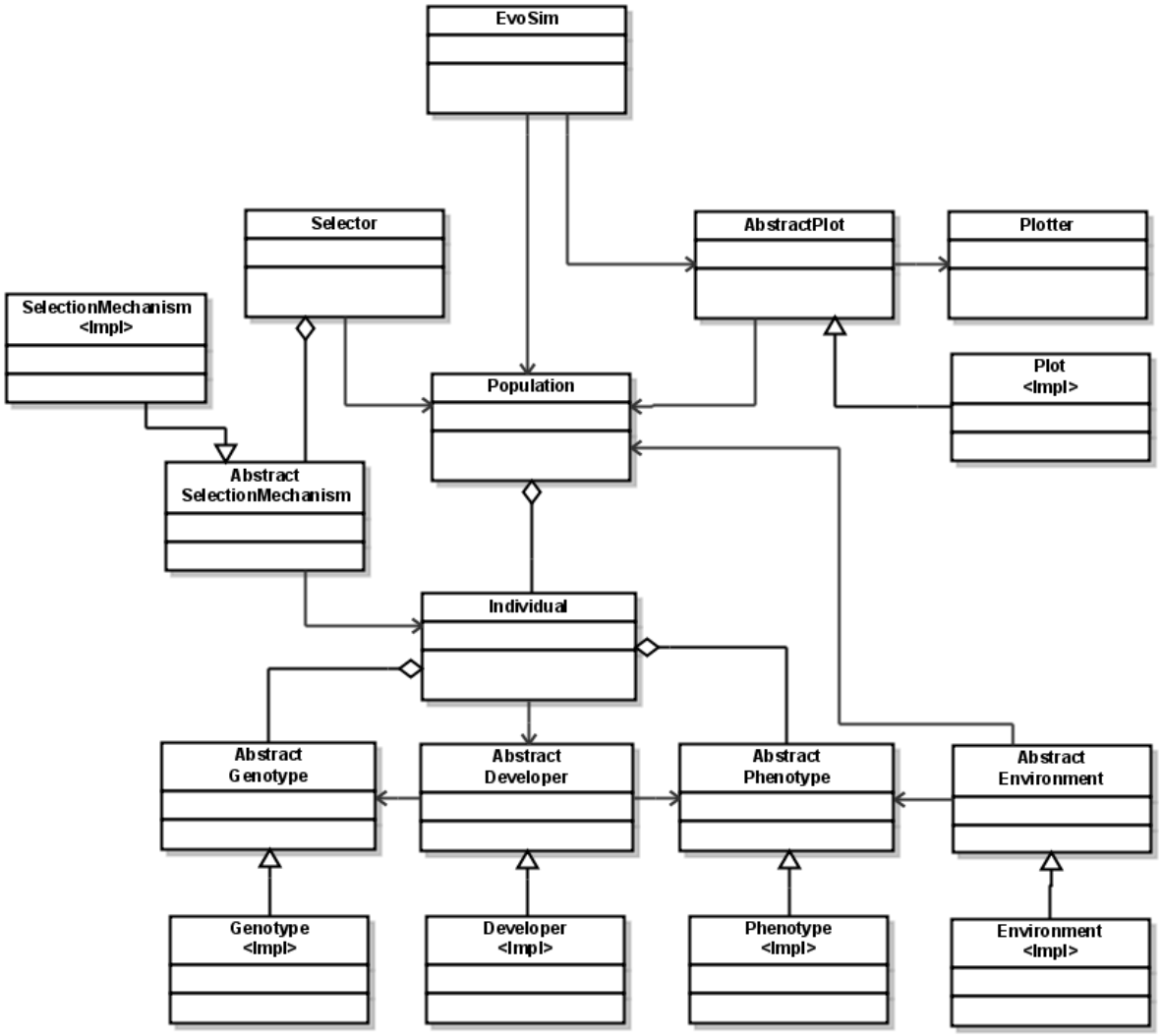

Figure 1: Class diagram of the framework and domain-specific files. The classes marked `impl` at the bottom row are domain specific implementations, tailored to each task. The abstract classes are part of the framework, and specifies which methods the domain classes must implement. They also hold methods shared across the domain classes. Each task specifies which classes it should use in its settings file.

The `Selector` has the help of an implementation of `SelectionMechanism` to do the actual selection. We have implemented all the selection mechanisms described in the compendium: `Tournament`, `MaxFitness`, `Boltzmann`, `Rank`, `FitnessProportionate` and `SigmaScaling`. The task can specify which methods it want to use in its settings.

The `Environment` a used to assess each individual's fitness is also a setting. A domain specific implementation of the `AbstractEnvironment` must be made, representing the fitness function. The environment can choose to asses each individual separately, or together with its generation in the case of coevolution.

This is also the way it works for the rest of the domain specific code. Implementations must be created (or reused) for `Genotype`, `Developer` and `Phenotype`. If a new way of doing adult or parent selection, or a new plot of the evolution, is needed it is also possible to extend `AbstractSelectionMechanism` and `AbstractPlot` for use in the task.

1.2 The One-Max problem

The framework has been successfully tested with the *One-Max* problem. A fitness-progression plot of a run can be found in Figure 2.

The run in the figure used 20 bits, elitism set to 1, selection protocol *A-II over production*, a mutation rate of 0.1 and a crossover rate of 0.9. Sigma-Scaling was used for the parent selection and and Max-Fitness for selecting adults. The genotype, phenotype and fitness function were implemented as described in the assignment. See `settings/onemax.yml` for the specific settings used while creating the plot.

For this problem, we used generic `BitVector` implementations for the Genotype and Phenotype, which can also be used for later tasks. The developer is a simple generic `Copier`, which creates a phenotype directly from the genotype.

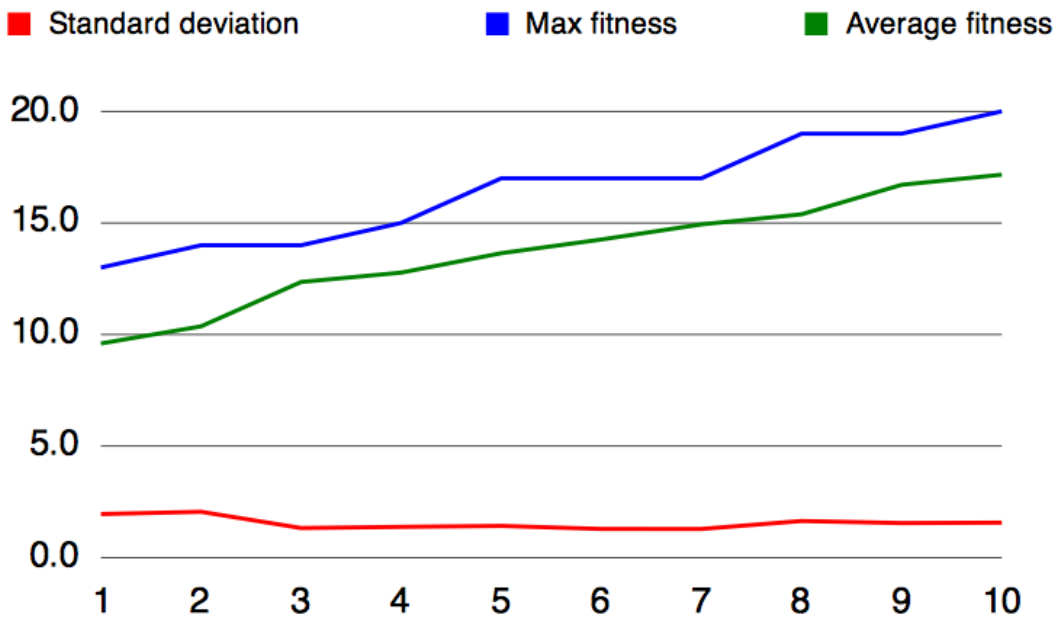

Figure 2: Fitness-progression of a run of One-Max. Elitism is set to 1, so the maximum fitness value never drops below the best solution obtained so far. The fitness cutoff threshold was set to 20, so the algorithm stopped when this goal was reached after 10 generations. If the threshold is not set, or if it is never reached, the algorithm will run until it reaches the setting for maximum number of generations.

2 Evolving Colonel Blotto Strategies using a Genetic Algorithm

We have used the framework described in part 1 to solve the *Colonel Blotto* problem. We describe our representation of the blotto strategies and how these strategies evolve and behave in different settings. We conclude by summing up our experiences using coevolution, interpreting the entropy value and with the problem itself.

2.1 Representation

To solve the Blotto problem, we implemented three domain-specific classes: `BlottoGenotype`, `BlottoEnvironment` and `FloatVectorPhenotype` (a new, generic phenotype). We were also able to reuse our `CopierDeveloper` from the One-Max problem.

The `BlottoEnvironment` is a straightforward implementation of the fitness description in the assignment. We pit the different strategies against each other using values for *reployment fraction* (R_f), *loss fraction* (L_f) and other parameters from the settings file `settings/blotto.yml`.

We have chosen a vector of floats as our representation of the strategies. The vector values are normalized, and each value represents the fraction of resources allocated to that battle.

We use the float vector to represent both the genotype and the phenotype. We found it sufficiently simple to perform the genetic operators on directly, and saw no gain in abstracting further. This, of course, also gives us perfect correlation between the genotype and phenotype spaces, as the phenotype is simply a copy of the genotype.

2.2 Genetic Operators

Our mutation operator selects one of the battles in a genotype at random, selects a random portion of the forces allocated to this battle and then moves those forces to another randomly selected battle. This way one mutation only affects two battles within a strategy, a rather small change. We control this by tweaking the *mutation.rate* setting.

The reason we chose to move around resources instead of simply adding/removing them in one battle was that we in that case would have to re-normalize the force allocation values. This would let one mutation affect all battles in the strategy, which would not be beneficial.

Crossover is done in a normal, random single-point fashion, after which the vectors are normalized. This means that the traits are not inherited directly, but changed according to what other traits appear after the crossover. However, there are problems with this approach.

A genome put together from one parent with hight weights in the start of the strategy, and an another parent with high weight at the end, will get a close to uniformly distributed strategy after normalization as we see in the following example.

Example: The crossing of `[0.4 | 0.1 0.1]` and `[0.2 | 0.4 0.4]` at the indicated locations will result in `[0.4 0.4 0.4]` and `[0.2 0.1 0.1]`. The normalized children `[0.33 0.33 0.33]` and `[0.4 0.2 0.2]` looks nothing like their parents.

Because the inheritance from a crossover is this low, we have deemphasized crossover and focused primarily on mutation in this task.

2.3 Phenotype-Fitness Correlation

The phenotype-fitness correlation is a complex part of Blotto since the strategies evolve using competitive coevolution. The trouble with this type of evolution is of course that a good strategy in one generation might become a terrible one in a later generation with a different set of opponents, i.e. the fitness landscape changes with time. Without an external fitness evaluator, determining whether a solution really is as good as its ranking in the generation suggest becomes difficult.

Still, given this problem, it is reasonable to assume that the changes in the fitness landscape is not totally chaotic and that changes happen gradually over time depending highly on how much change is introduced in each new generation. The fitness landscape in a generation should therefore be similar to that of the previous one.

With our representation of the genotype and phenotype, a mutation in one battle will have no effect on the rest of the strategy, given low values for reployment fraction and strength loss fraction. In this case the consequences of winning or losing the different battles is equal. High values for these fractions, however, results in that the early battles get a much higher impact on the fitness, and thus degrades the phenotype-fitness correlation somewhat.

2.4 Entropy

The entropy of a strategy is a measure of the uniformity of the force distribution. Uniform force distributions get lower entropy values than strategies where a fraction of the battles are emphasized.

The entropy of a strategy correlates with the rules set up for the current blotto problem. For instance, if the strength reduction fraction is high, the strategies will evolve towards having a lot of forces in their first battle, as this will result in more wins. The entropy for the winning strategy in this version of the problem will then be higher than the average entropy, as the winner will have a less evenly distributed force, with most focus on the first battle. See figure 3.

In our 18 runs of different variations of the blotto problem, we see that when the number of battles is high, the difference between the entropy of the best individual, and the average entropy, gets reduced. This is as expected, since while some battles may be more important than others, the number of uniform battles is higher, resulting in a higher entropy. In runs with fewer battles, this difference is a lot more clear.

As can be seen in all of the entropy plots, Figures 3, 5, 7, and 9, good strategies invariably have higher entropy than bad ones.

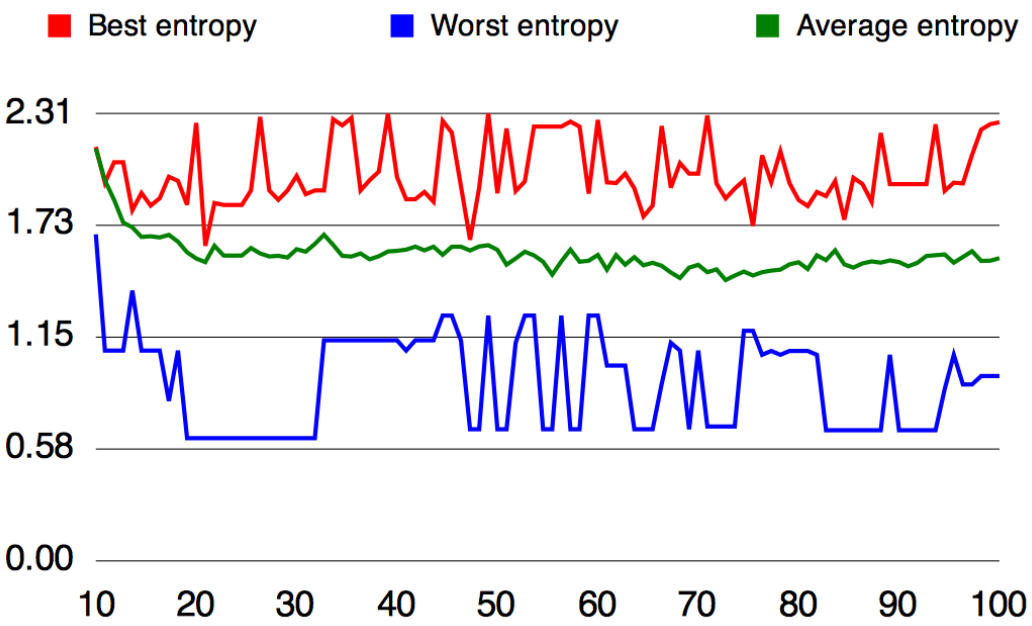

Figure 3: In this plot, we show the the entropy of the individuals with the highest and lowest fitness, and the average entropy of each generation. The strength reduction fraction and reployment fraction were both set to 0.1. We see that the entropy of the best individual is always higher than the average, and conversely, the worst individual remains below the average. Since the number of battles was low (5), the differences in entropy are quite clear, as opposed to a run with a larger number of battles.

2.5 Settings

We here list the key algorithm settings, their default values, and the reasoning supporting these values.

Max generations: As fitness is only dependent on the current generation, a fitness cutoff threshold would not be applicable to this problem. Therefore, running the algorithm long enough to explore a large part of the solution space becomes important. We chose to run the algorithm for **300** generations to make sure that the really good solutions have time to emerge.

Population size: To make sure each strategy is tried against many different competing strategies when testing fitness, we chose a default population size of **20**. We found that a lower population size would made it too easy for one strategy to dominate the others, and a higher value negatively affected running time, while not contributing to any significant change in the evolution of strategies.

Crossover rate: As discussed in Section 2.2, the crossover operator is somewhat problem-

atic in the blotto domain. We have therefore used the relatively low value of **0.2** as default here. A higher value gives a higher tendency for uniformly distributed strategies with low entropy and low fitness.

Mutation rate: With the low crossover rate, mutation becomes our main way to explore new strategies. We use a default value of **0.5**.

Elitism: As we wish to retain the best solution from the previous generation, we set the default elitism value to **1**. This is important for retaining a strategy if it has attained an optimal distribution of forces for the current number of battles. If one particular strategy is retained across multiple generations, we also get to test this currently best strategy against a larger number of differing strategies, to ensure its ability.

Parent selection mechanism: The parent selection mechanism seems to be one of the less important factors in this domain. We have tried all of them, but with only minor differences. We selected **fitness-proportionate** as our default value.

Adult selection mechanism: Same as with parent selection, the mechanism chosen for adult selection did not seem to have a huge impact on the results. Nevertheless we did seem to get an somewhat higher max-fitness with **tournament** than with any other, and selected this as our default.

Protocol: For our selection protocol we chose **A-III**, generational mixing, as the default value. This way we keep around those whom have proven themselves in the previous generation to fight another day.

2.6 Results

Test Runs

We have tested our implementation of Blotto with different settings for the number of battles and the reployment and strength reduction factors. Table 1 lists the results from 18 runs with variations on these parameters. The convergence numbers 1-3 signify the following:

1. Convergence to one stable strategy.
2. Alternating between similar strategies.
3. Continuous shifting between a host of different strategies.

Run	Battles	Reployment	Strength	Average Entropy	Best Entropy	Convergence
1	5	0	0	1.52	2.12	3
2	5	0	0.5	1.62	1.53	1
3	5	0	1	1.68	1.3	1
4	5	0.5	0	1.59	1.63	2
5	5	0.5	0.5	1.54	1.65	1
6	5	0.5	1	1.64	1.16	1
7	5	1	0	1.56	2.10	2
8	5	1	0.5	1.59	1.72	1
9	5	1	1	1.62	0.86	1
10	20	0	0	4.02	4.11	3
11	20	0	0.5	3.98	4.11	1
12	20	0	1	4.02	3.91	1
13	20	0.5	0	3.99	4.07	2
14	20	0.5	0.5	4.02	3.91	1
15	20	0.5	1	3.98	3.86	1
16	20	1	0	4.03	3.98	2
17	20	1	0.5	4.02	4.03	1
18	20	1	1	4.01	4.01	1

Table 1: Runs of variations on the Blotto problem. Each run has a setting for the number of battles, the reployment fraction and the strength reduction fraction. Also given is the average entropy in the final generation, and the entropy of the individual with the highest fitness in the last generation. See the text for description of the convergence numbers and notes on each run.

Notes on each of the runs in table 1:

1. Winning strategy gets continuously adapted to the other individuals, no single winning strategies emerge.
2. First battle gets the most resources, and each following battle is less and less prioritized.
3. Emphasizing first battle, and dividing remaining resources between two later battles.
4. Deemphasizing one battle and reinforcing other battles, often two strong and two medium forces.
5. As #4, but keeps the first and second battles strong, switches ignored battle between remaining slots.
6. As #5, but with even more focus on the first battle.
7. Always forces in first battle, deemphasizing one or two battles, distributing resources between remaining battles. First battle often the strongest.
8. Same as #7, but converges to one strategy. Even more focus on forces in first battle.

9. Emphasizing first battle, few forces in remaining battles. One (or two) of the remaining battles is often stronger than the rest, expect the first battle.
10. Strategy gets adapted to the other individuals, no single winning strategy.
11. First battle emphasized, remaining resources evenly distributed.
12. Emphasizing first battle, while semi-uniformly distributing remaining forces between the other battles.
13. Mostly uniform distribution of forces across battles.
14. Keeps first battle strong, while distributing remaining forces across other battles.
15. As #14, but with even more focus on the first battle.
16. Always forces in first battle, even distribution of forces in remaining battles.
17. Always forces in first battle, even distribution of forces in remaining battles.
18. Emphasizing first battle, while semi-uniformly distributing remaining forces between the other battles.

The settings in these 18 runs had their default values, as described in the previous section.

Runs 10-18 mostly exhibit the same characteristics as their counterparts with 5 runs, although the characteristics is far less distinct. The strategies then seem to evolve to similar goals, regardless of number of battles. In other words, the number of battles only change the rate of evolution and specificity of the prevailing strategy, not the overall convergence (or lack thereof). This can be seen in the ratio between best entropy and average entropy. In the runs with 20 battles the best solution has a distribution much more similar to the average distribution than in runs with 5 battles.

We see that without either *strength reduction* or *force reployment* the strategies seem to adapt to each other in a chaotic fashion. With only troop reployment a small set of similar strategies compete with the winner shifting among them. These strategies typically emphasize the first battles somewhat (although not always) and ignores one or two of the last battles almost completely. The introduction of strength reduction creates a convergence to strategies shifting their troops to the first battles, ignoring the rest.

Signature Runs

The following are descriptions of one run from each of the three convergence categories. They all use 5 battles, and the default configuration as given in the previous section. Each signature run is only different in their values for the reployment and strength reduction fractions.

The fitness and entropy plots for each signature case can be found in the ***appendix****.*

1) Convergence to a stable strategy: In this run $R_f = 0$ and $L_f = 0.5$. In Table 1, this is run #2. Loss fraction dictates that the first battles are especially important, in descending order from the very first battle. Because of this, one strategy will often beat all other strategies when it has the highest force allocation in the first battle. This can be seen in the fitness plot, where there is often one very good strategy, which then gets replaced by another, slightly better strategy, with a bit more forces in the first battles. See figures 4 and 5 in the appendix.
Example of winning strategy: `[0.55, 0.28, 0.03, 0.03, 0.11]`

2) Alternating between similar strategies: In this run $R_f = 0.5$ and $L_f = 0$. In Table 1, this is run #4. In this run, the reployment factor dictates that the important factor for winning is to have a few strong battles, while deemphasizing one or more other battles. The resulting strategies alternate between different versions of this basic scheme. As seen in the entropy graph for this case, an uneven distribution of forces is desirable to a uniform distribution, as a strategy with entropy value higher than the average entropy often prevails. See figures 6 and 7 in the appendix.
Example of winning strategy: `[0.25, 0.37, 0.11, 0.0, 0.27]`

3) Continuous shifting between many different strategies: In this run $R_f = 0$ and $L_f = 0$. In Table 1, this is run #1. Since both fractions are 0, there is not specific value to one or more specific battles - each one is as valuable. Because of this, the strategies continuously shift in order to prevail against the rest of the population. As soon as one strategy takes over the lead, other strategies evolve towards beating the new leader, resulting in a saw-tooth graph of new supreme strategies. See figures 8 and 9 in the appendix.
Example of winning strategy: `[0.15, 0.0, 0.19, 0.36, 0.3]`

2.7 Discussion

The Colonel Blotto problem is quite different from the One-Max problem. For instance, Blotto uses competitive coevolution to determine fitness values. This means that each fitness value is only dependent on the other individuals in the current generation. There is no external fitness metric against which each individual is measured. Because of this, the progression of the fitness values across generations will tell us quite different things than the progression exhibited by the One-Max problem.

Because fitness is only dependent on the current generation, the plot of maximum fitness will not be strictly increasing, even if elitism is applied. A non-increasing maximum fitness value across generations might in this problem mean that all the strategies in the population are getting better at beating each other. Since this blotto is a zero-sum game, the average fitness will always be 0.5.

In many of the fitness plots from this problem, we see a saw-tooth shaped pattern in the fitness values. This is caused by that one strategy is winning for a while, while more and

more other strategies are learning how to beat it, decreasing the fitness value of the best strategy. At some point, a new strategy will take over as the leading solution, starting another climb and descent in the fitness plot.

Our results were especially interesting when the reployment and strength reduction fractions were active. The visible convergence we observed could also be seen in the entropy plots, where it was clear that the best strategy would be one that dared follow the implications of fraction values to their logical conclusion by aggressively redistributing forces in a non-uniform manner.

Depending on the settings for strength reduction and troop reployment in the strategies, we observed different behaviour. Different settings gave either chaotic shifting between a host of strategies, shifting between a small set of similar strategies, or convergence to a single type of winner.

Appendix

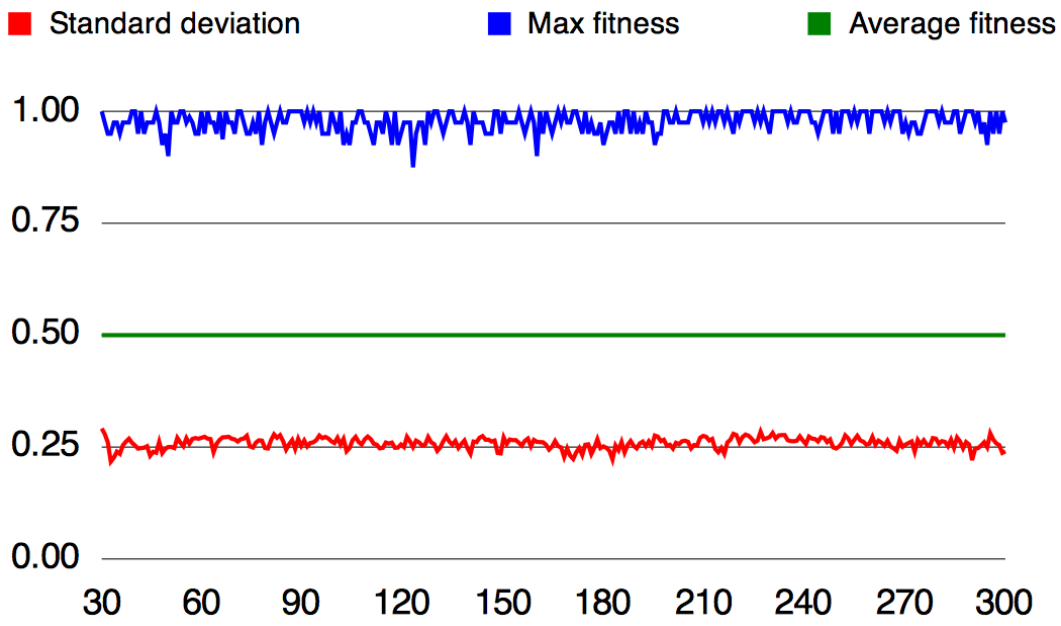

Figure 4: Signature Run 1: Fitness plot

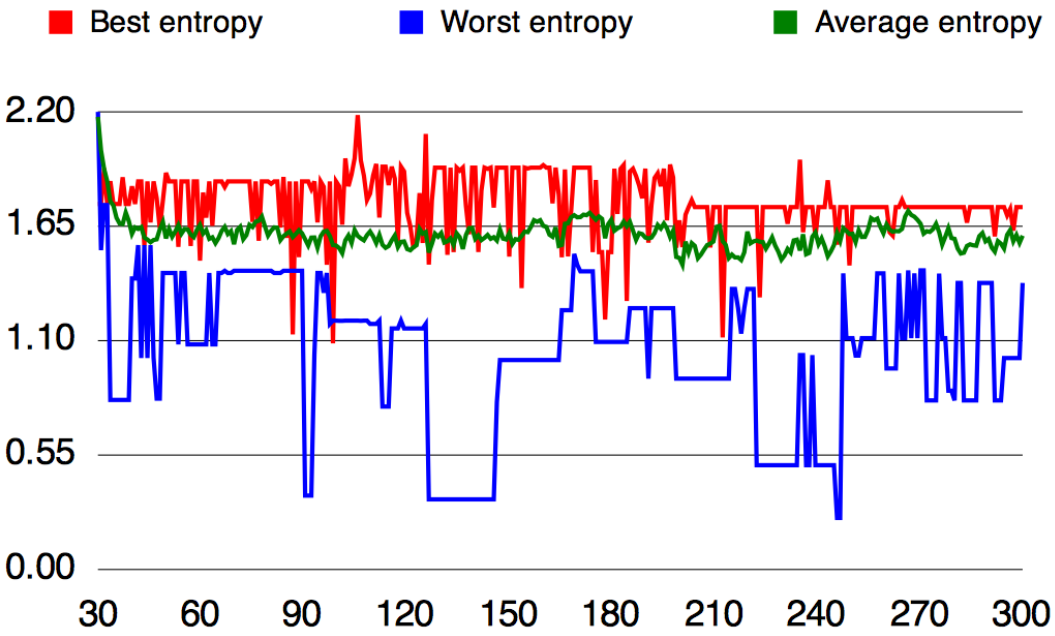

Figure 5: Signature Run 1: Entropy plot

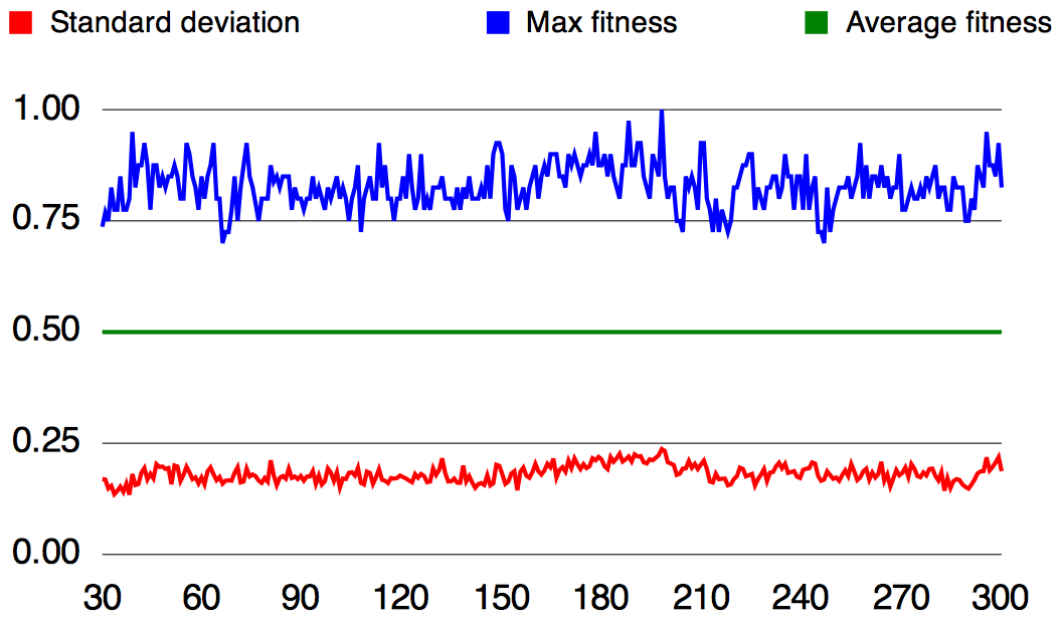

Figure 6: Signature Run 2: Fitness plot

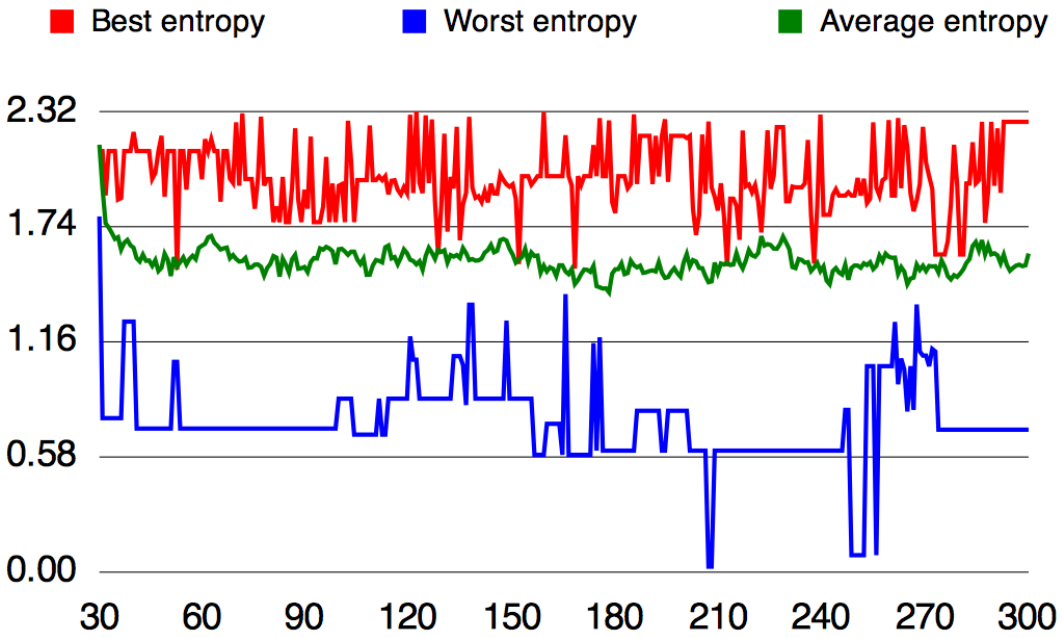

Figure 7: Signature Run 2: Entropy plot

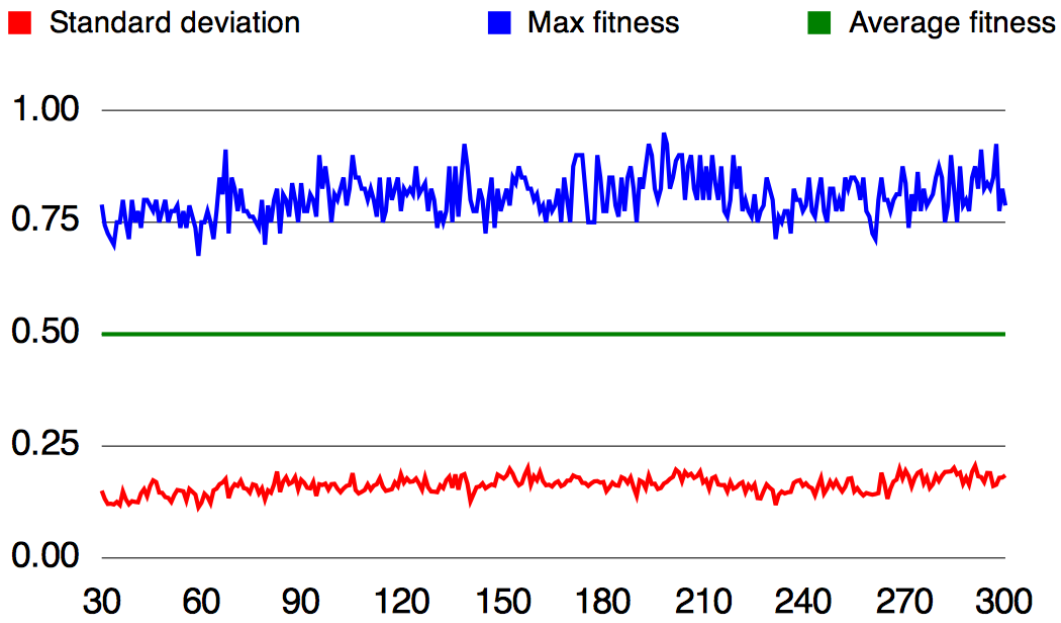

Figure 8: Signature Run 3: Fitness plot

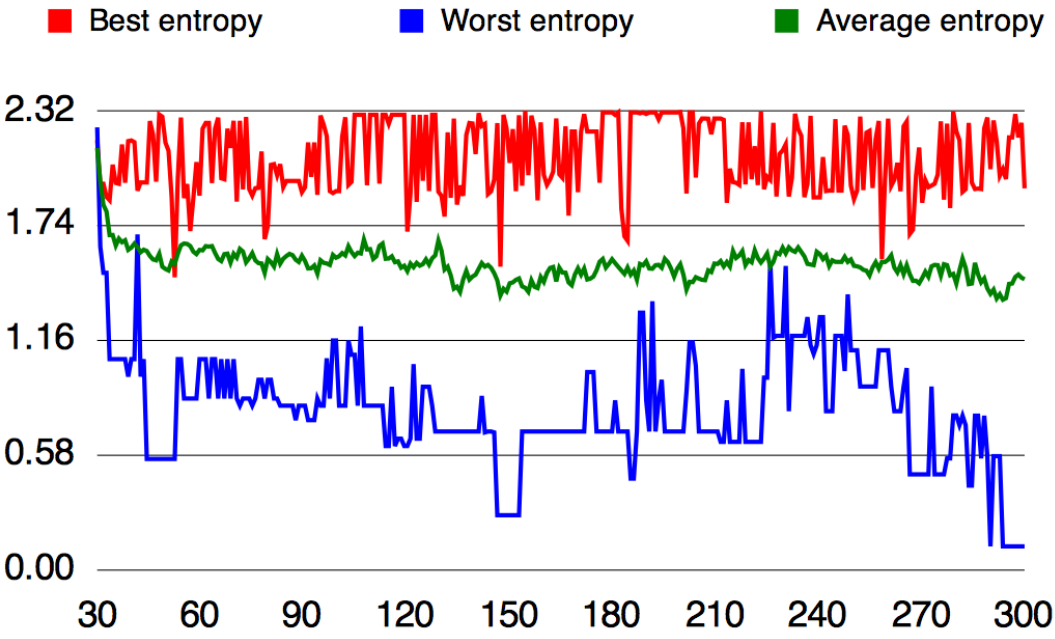

Figure 9: Signature Run 3: Entropy plot