COMP6202: Assignment 2

David Jones
dsj1n15@soton.ac.uk, ID: 27549836

1 Introduction

Paper Re-Implemented:
Coevolutionary Dynamics in a Minimal Substrate (Watson and Pollack 2001)

1.1 Experiments

The experiments described by Watson and Pollack 2001 are designed to demonstrate potential issues in co-evolutionary genetic algorithm (GA) implementations; usually stemming from a difference in the *subjective fitness* of an individual and its *objective fitness* ($f_{\text{obj}}(i) \neq f_{\text{subj}}(i)$). The former being the fitness as perceived by an individual co-evolving with other individuals, and the latter being the individual's actual fitness assessed via a fixed metric. The use of a subjective metric is key in GA applications as often an objective metric is not available. Although this provides benefits such as open-endedness (every time an individual improves it is the new target to beat), it can pose issues including:

A Disengagement: If all individuals in one population become able to beat all individuals in the other, there is a loss of gradient information and performance may drift without guidance.

B Over-Specialisation: Where an individual has multiple *traits*, other individuals may exploit a single one, leaving other traits undeveloped. These individuals are not generalised and may not learn the task at hand.

C 'Relativism': Co-adapting individuals are relatively scored against each other so $\text{P}_{\text{subj}}(a, b)$ may be equal if both a and b are good or if both a and b are bad; giving no selection incentive to be good.

In complex scenarios it is hard to identify when these scenarios occur so Watson and Pollack 2001 prove these phenomena on a "minimal substrate". Specifically, the problem of maximising the number of '1's in a fixed length bitstring or vector of bitstrings. This has a quantifiable objective fitness for external assessment: $f(i) = ones(i)$. It is a reasonable assumption that if pre-mentioned failures occur on this simple example, they are possible in complex examples.

1.2 Implementation

Reimplementation of the original paper was done in Python and can be found in Appendix B. Individuals are implemented as a vector of traits ($i = [t_1, t_2, \ldots, t_n]$) where a single trait is a scalar value representing the number of set bits ($t = t_{\text{value}} = ones(t_{\text{bitstring}})$). This is simpler than storing bitstrings with the drawback that crossover cannot be implemented. An individual is defined by the number of bits in each trait (t_l) and the number of traits (t_c); fully defined this is $i = \{t_c, t_l\}$. $f_{\text{obj}}(i)$ is defined as the sum of *ones* across all traits ($\sum_{n=1}^{t_c} ones(i(n))$). To allow assessment in a GA f_{subj} is defined as: $f(a, S)$ where a is an individual and S is a sample of the opponent population (with replacement), $|S| = 15$ unless stated. Scoring is fully defined by Equation 1. Intransitivity is *false* unless stated.

$$
\begin{aligned}
&f(a, S) = \sum_{i=1}^{|S|} \frac{score(a, S_i)}{|S|} \\
&\text{where } score(a, b) = 1 \text{ if } a(d) > b(d), 0 \text{ otherwise} \\
&\text{where } d = \begin{cases} \arg\min_d(a, b) = |a(d) - b(d)| & \text{if intransitive} \\ \arg\max_d(a, b) = |a(d) - b(d)| & \text{otherwise} \end{cases}
\end{aligned} \tag{1}
$$

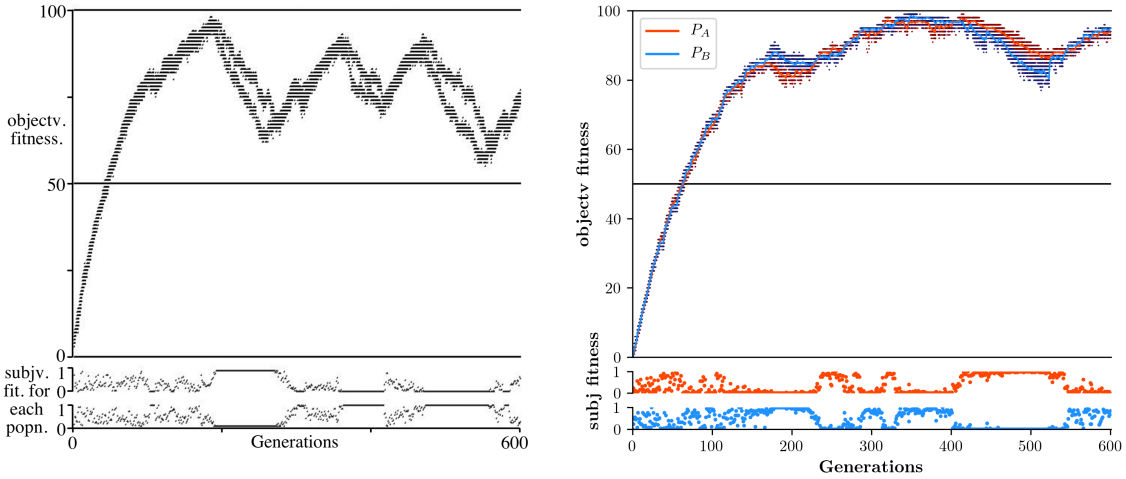
The two populations of individuals are defined as P_A and P_B; both of size 25 ($|P|$). Combined, the full GA algorithm is described as: **assess** the subjective fitness of each population, **select** $|P|$ individuals based on their fitness, **mutate** each individual, **insert** them into a new population, and **repeat**. This is a description of a generational GA. Selection is implemented as fitness proportionate selection where each individual has probability $p(i)$ of getting selected where $p(i) = \frac{f_{\text{subj}}(i)}{\sum_{j=1}^{n} f_{\text{subj}}(j)}$. This selection occurs $|P|$ times for each P_A and P_B to form the new populations. Note that as this selection is modelled as a *roulette wheel* the wheel will not *spin* when encountering individuals where $f_{\text{subj}}(i) = 0$. This issue is clearest when all elements have $f_{\text{obj}} = 0$, as $p(P(0)) = 1$ and $p(P > 0) = 0$. This is mitigated by adding a fixed bias that is close to 0 ($b = 1 \times 10^{-5}$) to all fitnesses. The mutation operator is defined as each bit in each trait having a chance to set to a new random value (0 or 1). This is implemented on the scalars as the original value having a 0.5% chance of having a 50% chance of decreasing $ones(t)$ times and increasing $t_l - ones(i)$ times. This captures inherit mutation bias that would be present in a bitstring, i.e. a string with more zeros than ones is likely to increase rather than decrease, meaning each trait will drift to $\frac{t_l}{2}$ This bias is key to patterns shown and is demonstrated in Figure 7.

2 Reimplementation Results

Overall the reimplementation of figures was successful with the expected patterns shown. For objective fitness all raw data points and an average trend line are plotted. For subjective fitness the average score for all individuals in the population is shown.

2.1 Experiment 1

The first experiment is configured to demonstrate disengagement resulting in a loss of gradient. The reproduced figure (Figure 1) shows this occurring in the form of drifting back to the neutral level during the polarised stages (generations 150-220 and 400-540) – polarisation can be identified by f_{subj} being 0 for one population and 1 for the other. Peaks are not as pronounced as the original, suggesting re-engagement happens quickly, however, as this occurs non-deterministically when the neutral position is approached the result is not considered an anomaly. For this problem disengagement can be fixed by providing a less polarising competition by increasing $|S|$; this is demonstrated in Figure 8. The clear contrast is that Figure 8 has a steady ascent until the maximum objective f_{obj} is reached, whereas this experiment does not reach the maximum due to occasional descents.



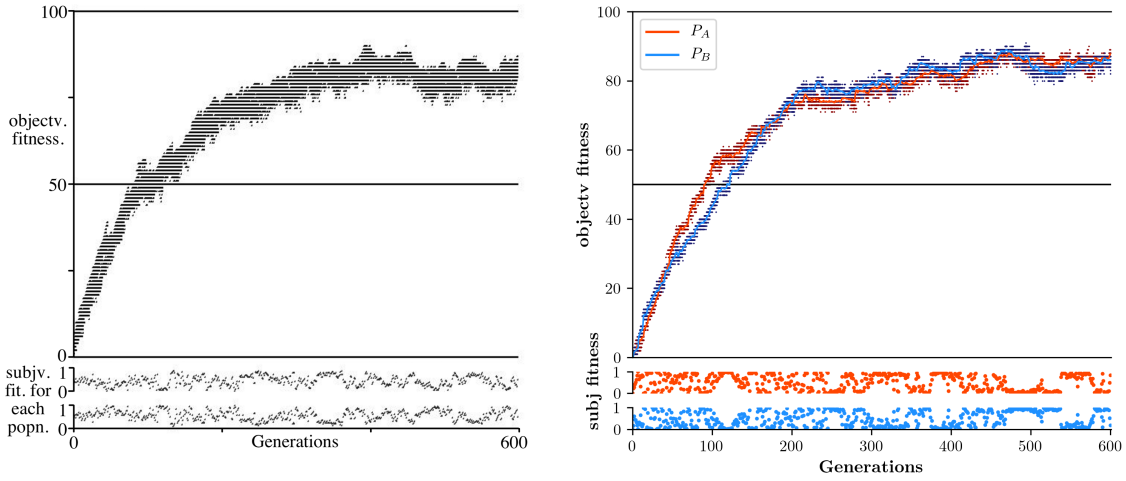
(a) Original (Watson and Pollack 2001) (b) Reimplementation

Figure 1: $i = \{1, 100\}$, $|S| = 1$ *(Originally Figure 3)*

2.2 Experiment 2

The second experiment concerns monitoring for over specialisation by increasing the number of dimensions in the individual. With intransitivity disabled, Equation 1 now applies scoring using the trait with the largest difference. This is implemented in Figure 2 and as expected, unlike Figure 8, the fitness does

not reach 100 due to an inability to maintain traits that are not being assessed as part of fitness assessments. This lack of dominance in all traits means even the elite of both populations is not guaranteed to beat an individual from a new population, i.e. they are overly focussed.

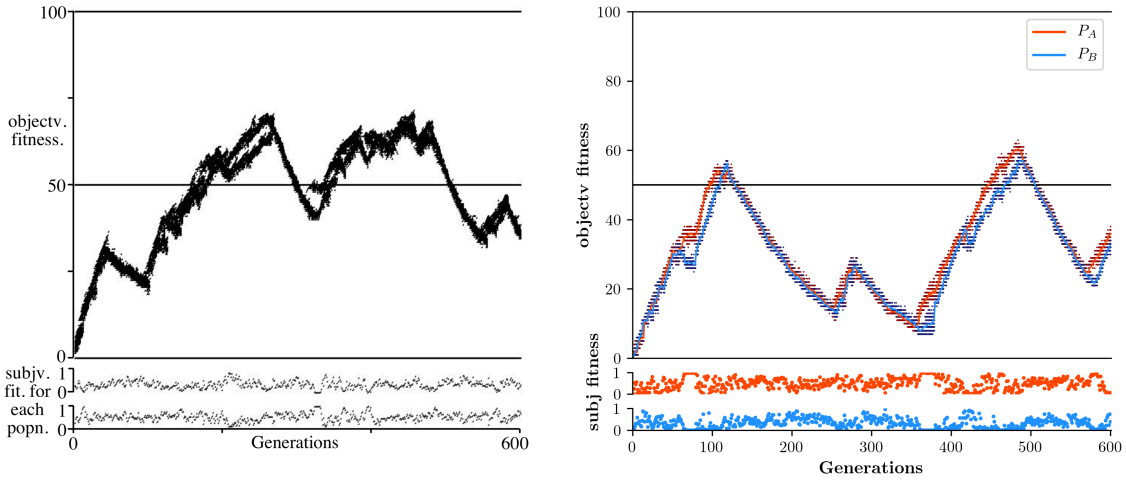


(a) Original (Watson and Pollack 2001) (b) Reimplementation

Figure 2: $i = \{10, 10\}$ *(Originally Figure 4)*

2.3 Experiment 3

The third experiment is an example of relativism via intransitivity. With intransitivity enabled, Equation 1 now applies scoring using the trait with the smallest difference. This is implemented in Figure 3 where the objective score is clearly being driven down away from the mutation bias level (generations 150-250) even more than that in the original paper – all whilst the subjective score is not polarised. This indicates that f_{subj} must be the opposite to f_{obj} and indicates an individual would rather lower its overall score to ensure its strongest trait is selected most times; this is a demonstration of cyclic superiority.



(a) Original (Watson and Pollack 2001) (b) Reimplementation

Figure 3: $i = \{2, 50\}$, intransitive=true *(Originally Figure 5)*

3 Extension

Hypothesis: Increasing diversity in populations during competitive coevolution can reduce the effects of relativism provided there is some selection pressure to improve.

3.1 Description

It is well accepted that high diversity in a population is key to avoiding premature convergence (Chong, Tino, and Yao 2009). Figure 3 illustrates a scenario where there is reduced pressure to improve due to cyclic behaviour. When the population starts clustering into specialising on single traits the selective pressure to increase overall fitness is sabotaged by the subjective fitness assessment. If this cyclic behaviour can be mitigated or introduced in a less constant manner the selection pressure for higher values as defined by $a(d) > b(d)$ should have more weight. In other terms volatility should be introduced to the population to avoid premature convergence where progress is punished.

Two main ideas are explored to handle this: **reduced virulence** (RV) to maintain balance and **hall of fame** (HOF) to encourage long term progress. Brief summaries of the two follow.

Cartlidge and Bullock 2004 proposes avoiding divergent behaviour by reducing virulence using the phantom parasite originally proposed by Rosin and Belew 1997. This works by making subjective fitness a function of both the normal score and a virulence parameter λ (see Equation 2). Fundamentally, this equation will penalise those scoring higher than a fraction (λ) of the highest individual.

$$f(x, \lambda) = \frac{2x}{\lambda} - \frac{x^2}{\lambda^2} \tag{2}$$

λ can take any value however the three main values usually considered are: 1.0 [Maximum Virulence], 0.75 [Moderate Virulence] and 0.5 [Null Virulence]. The original paper covers application of the concept to the counting ones domain using single trait individuals ($i = \{1, 100\}$). Two modifications are made that complicate the domain: a bias is introduced for a single population that encourages 1 alleles during mutation and half points are awarded in f_{subj} when f_{obj} is equal for two individuals. This extension studies the effects of reducing virulence onto the multi-trait intransitive problem (Experiment 3) without these extra biases.

A hall of fame (HOF) can be considered as an extra population made up of individuals from previous generations, these are usually the best and are otherwise known as elites. These individuals can be incorporated into fitness functions to reduce the chances of 'forgetting' and drive progress (Ranjeet et al. 2011).

Also touched upon are the drawbacks of using fitness proportionate selection (FPS) due to it encouraging proliferation of individuals that have a much higher fitness than the rest of the population (Blickle and Thiele 1996). Both tournament selection (TS) and stochastic universal sampling (SUS) are considered as alternatives. SUS as it avoids over selection of high fitness individuals due to noise (a dominant individual will at most only get selected one extra time than it should) (Blickle and Thiele 1996). Along with TS as when using a small tournament size, e.g. 3, lower probability elements have a high probability of being selected.

3.2 Implementation

Virulence was implemented as a wrapper for the selection method, where the fitness values are modified before being passed to the test selection method (e.g. FPS, SUS, TS). Fitness values were first normalised across each population between 0 and 1 before applying Equation 2.

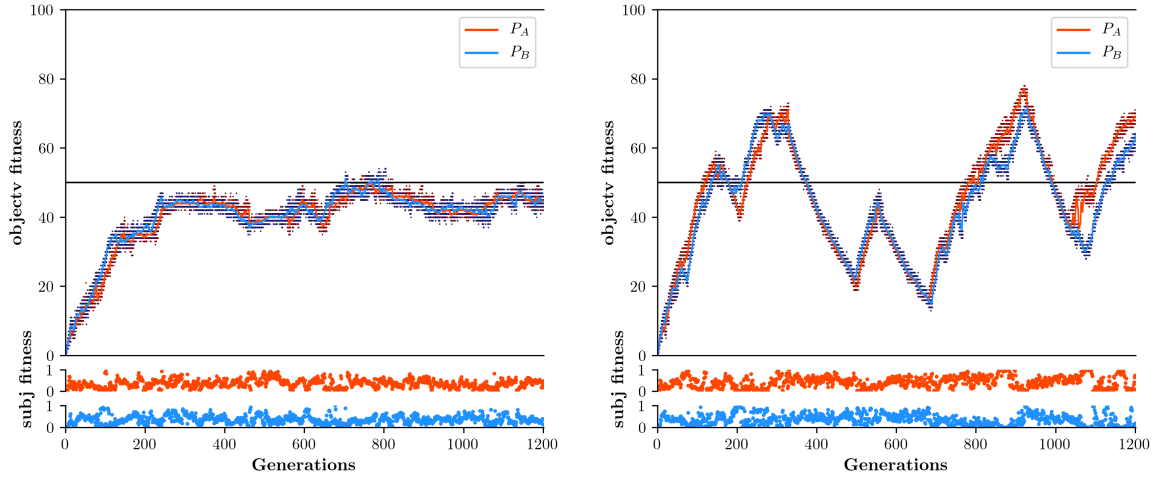
The HOF method captures a rolling window of elites for the last $|H|$ generations and then uses a random sample of (S_H) in conjunction with Equation 1. Rosin and Belew 1997 suggests random sampling is preferable as usually the computational effort to maintain performance information is not worth it. In an ideal world the elites picked for the HOF (H) would be based on f_{obj}, however with the intention of not introducing potentially undefinable information into the selection, f_{subj} is used. The downside of this is that relativism is likely still going to be present in the system. A HOF is only used where stated.

The selections methods implementations are based on pseudo-code from Blickle and Thiele 1996.

3.3 Results

The following results are described in the order that experimentation occurred. Initially targeting reduced virulence, introducing new selection methods and then incorporating a hall of fame. Note that all plots regarding the extension were created using the same random seed so all differences stem from the modifications introduced. Contrasts are made against Figure 3 from the base experiment.

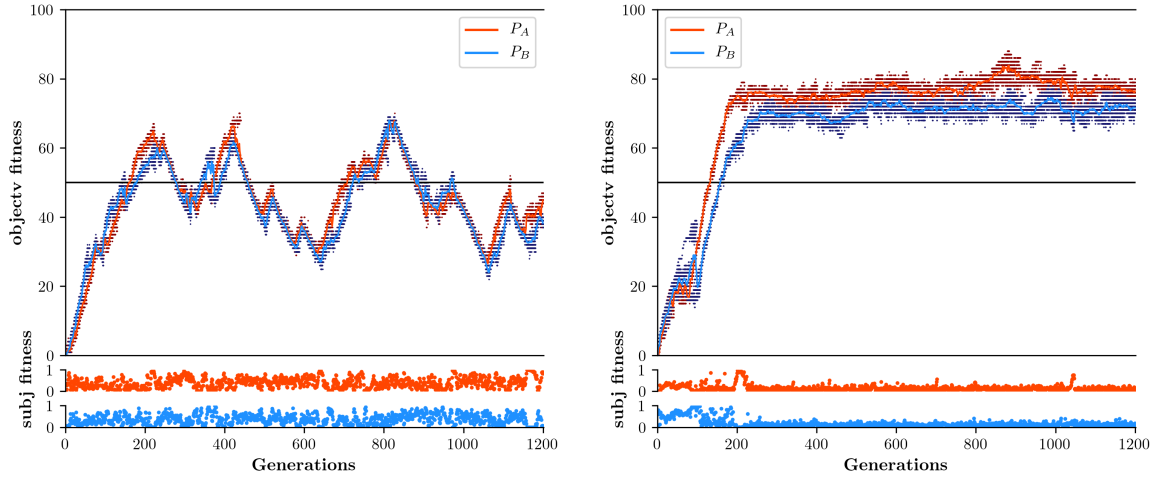
Starting with Figure 4 the results of incorporating reduced virulence can be seen for two values of λ: 0.5 and 0.75. The effects of null virulence are clear from Figure 4a, there is little selection pressure, meaning little incentive to move away from the mutual bias position once it is reached. Although this stops steep dives like in Experiment 3, there is not enough incentive for progress in the objective metric. Conversely, 0.75 yielded little if any improvement in regards to stability, however the overall max objective reached was higher 78 vs 64. An assumption was made that a more stable selection method may mitigate this.



(a) $\lambda = 0.5$ (b) $\lambda = 0.75$

Figure 4: $i = \{2, 50\}$, intransitive=true
Demonstration of different lambda values used for virulence reduction.

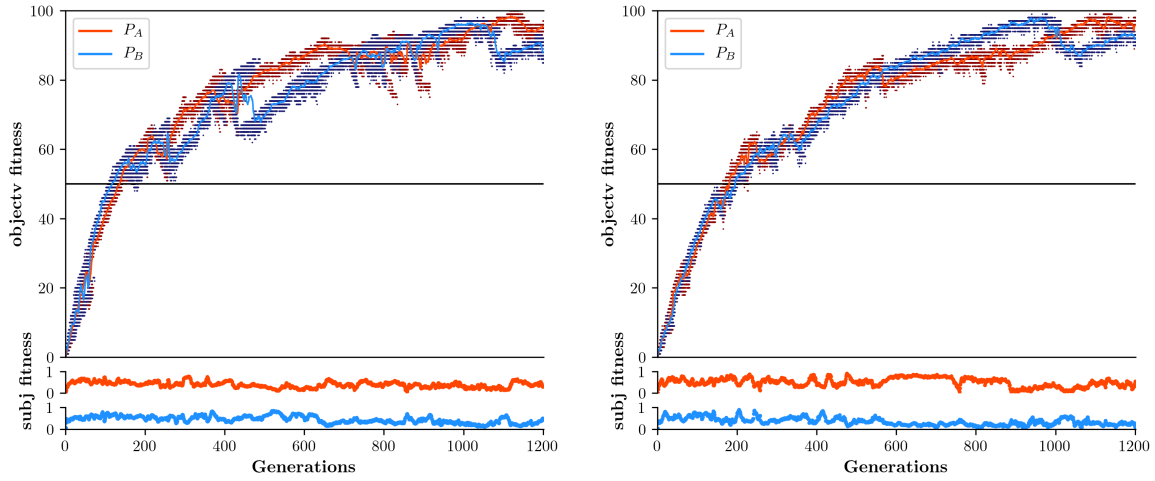
The results of applying TS and SUS alongside RV with $\lambda = 0.75$ can be seen in Figure 5. It is clear that TS causes a high level of instability in the evolution with more frequent periods of shorter descent; this is also at the expense of a high objective score. On the other hand SUS starts exhibiting good behaviour, with generation 200 onwards staying at a consistent level above the mutation bias point; this indicates each individual is able to keep a constant pressure on maintaining both traits. As new generations are not able to be suddenly dominated by one lucky individual, a period of stability can occur all whilst enjoying the benefits of a more diverse population.



(a) TS ($n = 3$) (b) SUS

Figure 5: $i = \{2, 50\}$, intransitive=true, $\lambda = 0.75$
Demonstration of different selection techniques with reduced virulence.

The final addition is the introduction of HOF. Figure 9 is provided in the appendix as a control – it shows that although HOF increases the overall objective it does not provide the diversity maintenance required to stop relativism causing downward spikes. Figure 6 shows what happens when reduced virulence provides this required mechanism. Figure 6a is a natural progression from Figure 5b with HOF added on top of SUS. The ability to now reach high objective values of 90-100 is present however there is large inter-population noise; the reason for this is unknown but could be attributed to HOF implementation making use of the selected selection method – i.e. the HOF implementation may just work better with FPS. Taking away SUS and leaving just HOF and VS gives the best result. Although there is occasional drops the high diversity gives a high chance of the populations re-engaging competitively as opposed to cooperatively. Notably adding HOF for $\lambda = 0.5$ (plot not shown), causes the average objective fitness to rise above 50; this is likely because once a good subset of previous generations is available it becomes the new neutral point.



(a) HOF + SUS

(b) HOF

Figure 6: $i = \{2, 50\}$, intransitive=true, $|H| = 50$, $S_H = 10$, $\lambda = 0.75$
Demonstration of hall of fame (HOF) introduction alongside reduced virulence.

4 Conclusion

This paper started by reimplementing the work of Watson and Pollack 2001 to demonstrate potential failings that may affect a coevolution GA. The implementation was verified against the original work to allow further exploration of the problem domain.

The problem was posed of reducing the effects of relativism through diversity. It was found that using a combination of mechanisms, one to drive diversity and one to ensure progress was being made that indeed the effects could be partially mitigated. However, in the punitive problem studied, it was not possible to completely avoid drift towards the mutation bias and potentially below. This suggests that in some scenarios the subjective scoring function may just be too contradictory to blindly trust a coevolutionary setup. That being said, the clear trend for higher values and rapid recovery from downward trajectory satisfies the hypothesis. Note that not one of the added mechanics could fix the problem by itself: either RV + HOF or RV + SUS was required for a reasonable improvement to be made.

There are many areas that can be expanded on to further understand the results shown and increase confidence. These should cover: optimisation of the many parameters controlling the mechanisms, trials of more complex HOF systems, such as those demonstrated by Nogueira, Cotta, and Fernández-Leiva 2013, and more detailed focus on the individuals make-up to verify if the expected distribution of trait values are present. Branching from this, individuals are currently only considered with two traits, the ideas proposed may fail when more are introduced due to even more tight binding from the intransitive relationships.

References

Blickle, Tobias and Lothar Thiele (Dec. 1996). "A Comparison of Selection Schemes Used in Evolutionary Algorithms". In: *Evol. Comput.* 4.4, pp. 361–394. ISSN: 1063-6560. DOI: 10.1162/evco.1996.4.4.361. URL: https://doi.org/10.1162/evco.1996.4.4.361.

Cartlidge, John and Seth Bullock (2004). "Combating coevolutionary disengagement by reducing parasite virulence". In: *Evolutionary Computation* 12.2, pp. 193–222. URL: https://eprints.soton.ac.uk/261440/.

Chong, Siang Yew, Peter Tino, and Xin Yao (Oct. 2009). "Relationship Between Generalization and Diversity in Coevolutionary Learning". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 1, pp. 214–232. DOI: 10.1109/TCIAIG.2009.2034269.

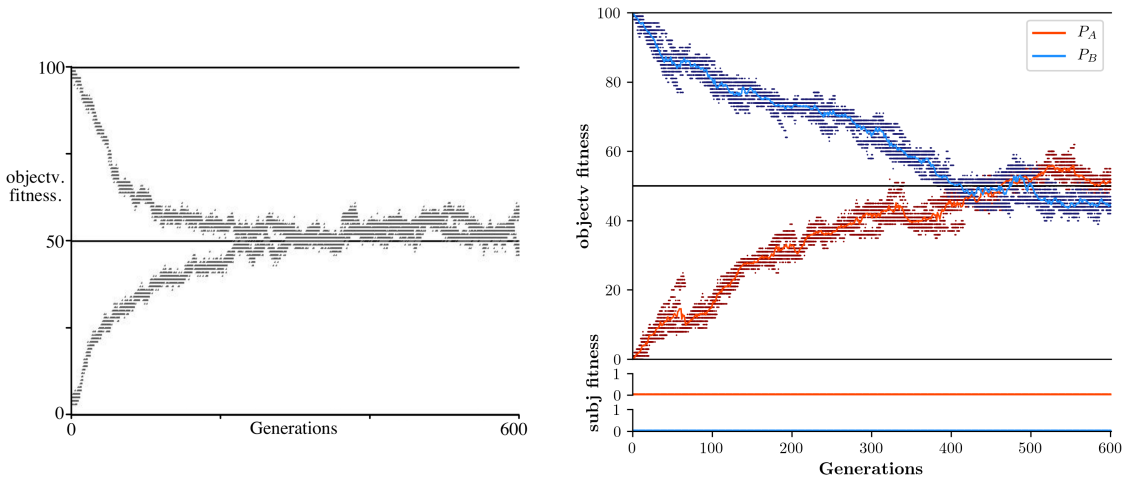
Nogueira, Mariela, Carlos Cotta, and Antonio Fernández-Leiva (Jan. 2013). "An Analysis of Hall-of-Fame Strategies in Competitive Coevolutionary Algorithms for Self-Learning in RTS Games". In: vol. 7997, pp. 174–188. DOI: 10.1007/978-3-642-44973-4_19.

Ranjeet, Tirtha et al. (Dec. 2011). "The Effects of Diversity Maintenance on Coevolution for an Intransitive Numbers Problem". In: vol. 7106, pp. 331–340. DOI: 10.1007/978-3-642-25832-9_34.

Rosin, Christopher Darrell and Richard K. Belew (1997). "Coevolutionary Search among Adversaries". PhD thesis. USA.

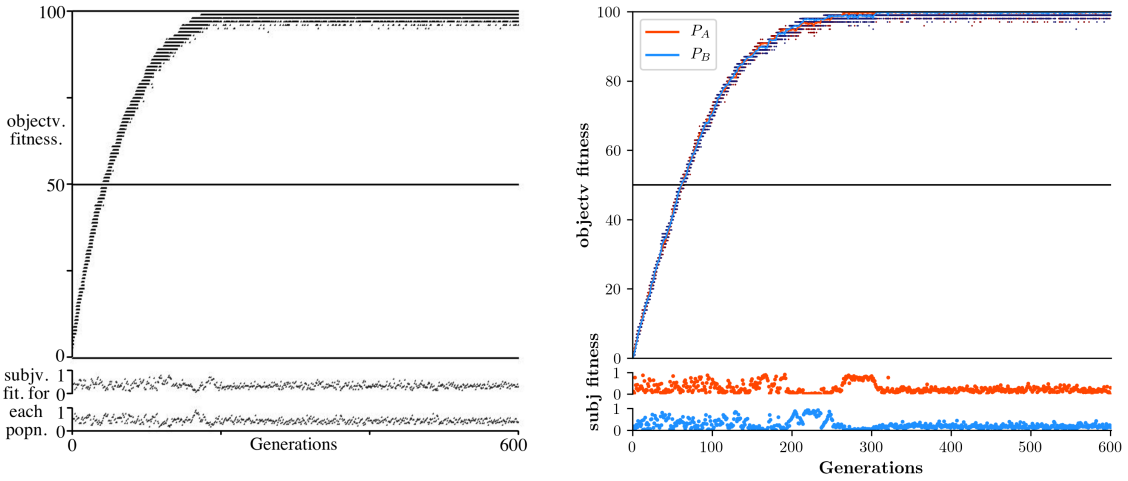
Watson, Richard A. and Jordan B. Pollack (2001). "Coevolutionary Dynamics in a Minimal Substrate". URL: https://eprints.soton.ac.uk/262011/.

Appendix A Extra Figures



(a) Original (Watson and Pollack 2001) (b) Reimplementation

Figure 7: $i = \{1, 100\}$ Control figure where $f_{\text{subj}} = 0$ (no scoring function) demonstrating natural drift to $t_l/2 = 50$ due to mutation bias. *(Originally Figure 1)*



(a) Original (Watson and Pollack 2001) (b) Reimplementation

Figure 8: $i = \{1, 100\}$ Control figure showing GA working in ideal conditions. *(Originally Figure 2)*

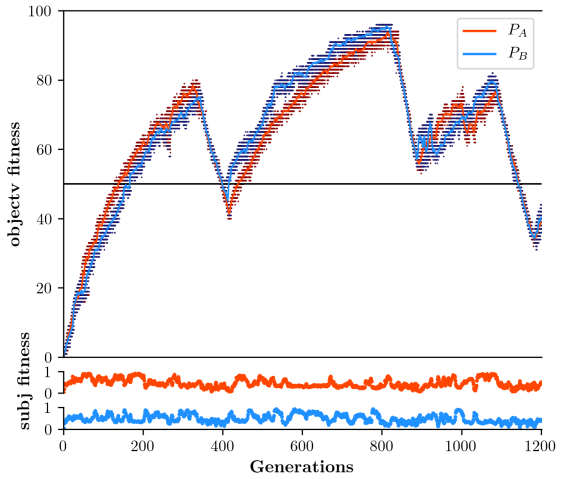


Figure 9: $i = \{2, 50\}$ intransitive=true, $|H| = 50$, $S_H = 10$

Appendix B Code Listings

B.1 assignment2.py

```
"""COMP6202

Reimplementation of the paper:
    Coevolutionary Dynamics in a Minimal Substrate (Watson and Pollack, 2001)

Extension:
    Implementation of Reduced Virilisation and 'Hall of Fame'.

File to execute as main to start coevolution.
"""
__authors__ = "David Jones <dsj1n15@ecs.soton.ac.uk>"


import os
import cv2 as cv
import matplotlib.pyplot as plt

from coevolution import Coevolution, HOF
from representation import Generator
from mutation import Mutator
from scoring import Scorer, FOScorer
from selection import (
    Selector,
    VirulenceSelector,
    FitnessProportionateSelection,
    StochasticUniversalSampling,
    TournamentSelection,
)

from plot import Plotter
from utils import seed_random


def run():
    plotter = Plotter(show_avgs=True, plot_elites=False)
    fig_1 = 1
    fig_2 = 1
    fig_3 = 1
    fig_4 = 1
    fig_5 = 1
    extension = 1

    use_fixed_seed = 1
    enable_hof = 0
    enable_virulence = 0
    generations = 600

    export_folder = "../report/plots/"
    generator_a = Generator(100, 1)
    generator_b = Generator(10, 10)
    generator_c = Generator(50, 2)
    mutator = Mutator(mutation_rate=0.005, bit_flip=False)
```

```
selector = FitnessProportionateSelection()
n = 25

if enable_hof:
    hof = HOF(scorer=Scorer(sample_size=10), size=50)
else:
    hof = None

# Wrap selector with virulence handling
if enable_virulence:
    selector = VirulenceSelector(selector, 0.75, normalise=True)

if fig_1:
    seed = seed_random(0, use_fixed_seed)
    description = "Figure 1 : [Seed {}]".format(seed)
    print(description)
    pop_a = generator_a.population(n, 0)
    pop_b = generator_a.population(n, 100)
    executor = Coevolution(scorer=F0Scorer(), selector=selector)
    executor.run(pop_a, pop_b, generations)
    plotter.make_plot(
        executor,
        fig_name=description,
        export_path=os.path.join(export_folder, "fig1.png"),
    )

if fig_2:
    seed = seed_random(1, use_fixed_seed)
    description = "Figure 2 : [Seed {}]".format(seed)
    print(description)
    pop_a = generator_a.population(n, 0)
    pop_b = generator_a.population(n, 0)
    executor = Coevolution(mutator=mutator, hof=hof, selector=selector)
    executor.run(pop_a, pop_b, generations)
    plotter.make_plot(
        executor,
        fig_name=description,
        export_path=os.path.join(export_folder, "fig2.png"),
    )

if fig_3:
    seed = seed_random(8486058433753192762, use_fixed_seed)
    description = "Figure 3 : [Seed {}]".format(seed)
    print(description)
    pop_a = generator_a.population(n, 0)
    pop_b = generator_a.population(n, 0)
    executor = Coevolution(
        mutator=mutator, scorer=Scorer(sample_size=1), hof=hof, selector=selector
    )
    executor.run(pop_a, pop_b, generations)
    plotter.make_plot(
        executor,
        fig_name=description,
        export_path=os.path.join(export_folder, "fig3.png"),
    )
```

```
if fig_4:
    seed = seed_random(59759543964706904, use_fixed_seed)
    description = "Figure 4 : [Seed {}]".format(seed)
    print(description)
    pop_a = generator_b.population(n, 0)
    pop_b = generator_b.population(n, 0)
    executor = Coevolution(mutator=mutator, hof=hof, selector=selector)
    executor.run(pop_a, pop_b, generations)
    plotter.make_plot(
        executor,
        fig_name=description,
        export_path=os.path.join(export_folder, "fig4.png"),
    )

if fig_5:
    seed = seed_random(5706501168717675099, use_fixed_seed)
    description = "Figure 5 : [Seed {}]".format(seed)
    print(description)
    pop_a = generator_c.population(n, 0)
    pop_b = generator_c.population(n, 0)
    executor = Coevolution(
        mutator=mutator,
        scorer=Scorer(intransitive=True),
        hof=hof,
        selector=selector,
    )
    executor.run(pop_a, pop_b, generations)
    plotter.make_plot(
        executor,
        fig_name=description,
        export_path=os.path.join(export_folder, "fig5.png"),
    )

if extension:
    hof = HOF(scorer=Scorer(sample_size=10), size=50)
    generations = 1200
    fp_selector = FitnessProportionateSelection()
    sus_selector = StochasticUniversalSampling()
    t_selector = TournamentSelection(3)
    ext_cfgs = []
    ext_cfgs += [
        (
            "Virulence [0.5]",
            "fig_5_v0.5.png",
            VirulenceSelector(fp_selector, 0.5),
            False,
        )
    ]
    ext_cfgs += [
        (
            "Virulence [0.75]",
            "fig_5_v0.75.png",
            VirulenceSelector(fp_selector, 0.75),
            False,
        )
    ]
```

```
        ext_cfgs += [
            (
                "Virulence [0.75] + SUS",
                "fig_5_v0.75_sus.png",
                VirulenceSelector(sus_selector, 0.75),
                False,
            )
        ]
        ext_cfgs += [
            (
                "Virulence [0.75] + TS",
                "fig_5_v0.75_ts.png",
                VirulenceSelector(t_selector, 0.75),
                False,
            )
        ]
        ext_cfgs += [
            (
                "Virulence [0.75] + SUS + HOF",
                "fig_5_v0.75_sus_hof.png",
                VirulenceSelector(sus_selector, 0.75),
                True,
            )
        ]
        ext_cfgs += [
            (
                "Virulence [0.75] + HOF",
                "fig_5_v0.75_hof.png",
                VirulenceSelector(fp_selector, 0.75),
                True,
            )
        ]
        ext_cfgs += [("HOF", "fig_5_hof.png", fp_selector, True)]
        for (cfg_txt, export_name, selector, use_hof) in ext_cfgs:
            seed = seed_random(8985012493578745191, use_fixed_seed)
            description = "Figure 5 - {} : [Seed {}]".format(cfg_txt, seed)
            print(description)
            pop_a = generator_c.population(n, 0)
            pop_b = generator_c.population(n, 0)
            executor = Coevolution(
                mutator=mutator,
                scorer=Scorer(intransitive=True),
                selector=selector,
                hof=hof if use_hof else None,
            )
            executor.run(pop_a, pop_b, generations)
            plotter.make_plot(
                executor,
                fig_name=description,
                export_path=os.path.join(export_folder, export_name),
            )


def run_loop(repeat: bool = False, executions: int = 1):
    while True:
        for e in range(executions):
```

```
            run()
        plt.show()
        cv.waitKey()
        if not repeat:
            break


if __name__ == "__main__":
    run_loop()
```

B.2 coevolution.py

```
from __future__ import annotations

"""COMP6202 - Evolution of Complexity
Holds main execution class [Coevolution] for running a GA.
"""

__authors__ = "David Jones <dsj1n15@ecs.soton.ac.uk>"

import copy
import numpy as np

from typing import List, Tuple

from representation import Population, Individual
from mutation import Mutator
from scoring import Scorer
from selection import Selector, FitnessProportionateSelection


class Coevolution:
    def __init__(
        self,
        scorer: Scorer = Scorer(intransitive=False),
        selector: Selector = FitnessProportionateSelection(),
        mutator: Mutator = Mutator(),
        hof: HOF = None,
    ):
        self.scorer = scorer
        self.selector = selector
        self.mutator = mutator
        self.pops_a = []
        self.pops_b = []
        # Use HOF if provided
        self.hof = hof is not None
        self.hof_a = copy.deepcopy(hof)
        self.hof_b = copy.deepcopy(hof)

    def run(self, pop_a: Population, pop_b: Population, generations: int):
        self.pops_a, self.pops_b = [pop_a], [pop_b]
        self.subj_a, self.subj_b = [], []
        for g in range(generations + 1):
            # Use the last populations
            pop_a = self.pops_a[-1]
            pop_b = self.pops_b[-1]
            # Record average subjective score
            f_ab, f_ba = self.assess_fitness(pop_a, pop_b)
            self.subj_a += [np.mean(f_ab)]
            self.subj_b += [np.mean(f_ba)]
            if self.hof:
                self.hof_a.add(pop_a[np.argmax(f_ab)])
                self.hof_b.add(pop_b[np.argmax(f_ba)])
            if g == generations:
                break
            # Get next generations
```

```
            self.pops_a += [self.next_generation(pop_a, f_ab)]
            self.pops_b += [self.next_generation(pop_b, f_ba)]

    def next_generation(self, pop: Population, f: List[float]) -> Population:
        idxs = self.selector.select(f)
        new_pop = Population()
        for idx in idxs:
            new_pop += [self.mutator.mutate(pop[idx])]
        return new_pop

    def assess_fitness(
        self, pop_a: Population, pop_b: Population
    ) -> Tuple[List[float], List[float]]:
        pop_ab = self._assess_fitness(pop_a, pop_b, self.hof_a)
        pop_ba = self._assess_fitness(pop_b, pop_a, self.hof_b)
        return (pop_ab, pop_ba)

    def _assess_fitness(
        self, pop: Population, opponents: Population, hof: HOF = None
    ) -> List[float]:
        pop_scores = []
        for i in pop:
            score = self.scorer.subj_fitness(i, opponents)
            if hof:
                score += hof.scorer.subj_fitness(i, hof.pop)
            pop_scores += [np.mean(score)]
        return pop_scores


class HOF:
    def __init__(self, scorer: Scorer, size: int):
        self.scorer = scorer
        self.size = size
        self.pop = Population()

    def add(self, ind: Individual):
        self.pop += [ind]
        if len(self.pop) > self.size:
            self.pop.pop(0)
```

B.3 representation.py

```python
from __future__ import annotations

"""COMP6202 - Evolution of Complexity

Holds individual representation and population holder.
"""

from collections import UserList
from typing import List, Tuple


class Generator:
    def __init__(self, trait_bits: int, trait_count: int):
        self.trait_bits = trait_bits
        self.trait_count = trait_count

    def individual(self, value: int) -> Individual:
        traits = []
        for i in range(self.trait_count):
            traits += [Trait(value, self.trait_bits)]
        return Individual(traits)

    def population(self, n: int, value: int) -> Population:
        return Population([self.individual(value)] * n)


class Population(UserList):
    """ Populations are just lists of individuals.
    """

    def __init__(self, elements: List[Individual] = [], **kwargs):
        super().__init__(elements, **kwargs)


class Individual:
    def __init__(self, traits: List[Trait]):
        """[summary]

        Args:
            traits (List[Trait]): [description]
        """
        self.traits = traits

    @property
    def values(self) -> Tuple[int]:
        return tuple(map(lambda x: x.value, self.traits))

    @property
    def value(self) -> int:
        return sum(self.values)

    @property
    def total_bits_by_trait(self) -> Tuple[int]:
        return tuple(map(lambda x: x.total_bits, self.traits))
```

```
    @property
    def total_bits(self) -> int:
        return sum(self.total_bits_by_trait)

    def as_bits(self):
        bits = []
        for t in self.traits:
            bits += t.as_bits()
        return bits

    def __repr__(self) -> str:
        return str(self.traits)


class Trait:
    def __init__(self, value: int, total_bits: int):
        """Each trait's bitstring represented by their unitations i.e.
        the number of set bits. The total number of bits must be tracked.

        Args:
            value (int): Number of set bits.
            total_bits (int): Total number of bits in the trait.
        """
        if value > total_bits or value < 0:
            raise ValueError("Value > total_bits or < 0")
        self._value = value
        self.total_bits = total_bits

    @property
    def value(self) -> int:
        return self._value

    @value.setter
    def value(self, value: int):
        self._value = max(0, min(value, self.total_bits))

    def as_bits(self):
        return [1] * self.value + [0] * (self.total_bits - self.value)

    def __repr__(self) -> str:
        return str(self.value)
```

B.4 mutation.py

```
"""COMP6202 - Evolution of Complexity
Holds mutation class for mutating an individual.
"""

__authors__ = "David Jones <dsj1n15@ecs.soton.ac.uk>"

import copy
import random

from representation import Individual


class Mutator:
    def __init__(self, mutation_rate: float = 0.005, bit_flip: bool = False):
        self.mutation_rate = mutation_rate
        self.bit_flip = bit_flip

    def mutate(self, i: Individual, inplace: bool = False) -> Individual:
        if not inplace:
            i = copy.deepcopy(i)
        for t in i.traits:
            set_bits = t.value
            unset_bits = t.total_bits - set_bits
            t.value = 0
            for b in range(set_bits):
                t.value += self._mutate_bit(1)
            for b in range(unset_bits):
                t.value += self._mutate_bit(0)
        return i

    def _mutate_bit(self, bit_value: int) -> int:
        if random.uniform(0, 1) > self.mutation_rate:
            # No mutation
            return bit_value
        # Return new bit value
        if self.bit_flip:
            return not bit_value
        else:
            return random.choice([0, 1])
```

B.5 scoring.py

```
"""COMP6202 - Evolution of Complexity
Holds subjective scoring classes: Scorer, F0Scorer.
"""

__authors__ = "David Jones <dsj1n15@ecs.soton.ac.uk>"

import operator
import random

from typing import List
from representation import Population, Individual


class Scorer:
    """Scorer that implements `score2` and `score3`.
    """

    def __init__(self, sample_size: int = 15, intransitive: bool = False):
        if intransitive:
            self.op = operator.lt
        else:
            self.op = operator.gt
        self.sample_size = sample_size

    def score(self, a: Individual, b: Individual) -> int:
        a = a.values
        b = b.values
        mi = 0
        for (i, (ta, tb)) in enumerate(zip(a, b)):
            delta = abs(tb - ta)
            if self.op(delta, abs(a[mi] - b[mi])):
                mi = i
        return Scorer._score(a[mi], b[mi])

    @staticmethod
    def _score(a: int, b: int) -> int:
        return 1 if a > b else 0

    def subj_fitness(self, a: Individual, pop: Population) -> List[int]:
        """Calculate the subjective fitness for an individual using an
        opponent population.

        Args:
            a (Individual): The individual to score against the opponents.
            pop (Population): The population to select opponents from.

        Returns:
            List[int]: The individual score against each selected individual
                in the population. This will be a list of length sample size.
        """
        if len(pop) == 0:
            return []
        # Allow repeats, same as applet
        pop = random.choices(pop, k=self.sample_size)
```

```
        return list(map(lambda b: self.score(a, b), pop))


class F0Scorer(Scorer):
    """Scorer that always returns 0 for subjective score (Experiment 0).
    """

    def subj_fitness(self, *args) -> int:
        return 0
```

B.6 selection.py

```
"""COMP6202 - Evolution of Complexity

Holds different selection classes.
"""

__authors__ = "David Jones <dsj1n15@ecs.soton.ac.uk>"

import random
import numpy as np

from typing import List

DEFAULT_BIAS = 0.000001


class Selector:
    def select(self, fs: List[float], k: int = None) -> List[int]:
        pass


class FitnessProportionateSelection(Selector):
    def __init__(self, bias: float = DEFAULT_BIAS):
        self.bias = bias

    def select(self, fs: List[float], k: int = None) -> List[int]:
        if not k:
            k = len(fs)
        # Make wheel
        total = 0
        wheel = []
        for f in fs:
            total += f + self.bias
            wheel += [total]
        # Spin wheel
        selected = []
        for s in range(k):
            i = 0
            pick = random.random() * total
            while wheel[i] < pick:
                i += 1
            selected += [i]
        return selected


class StochasticUniversalSampling(Selector):
    def __init__(self, bias: float = DEFAULT_BIAS):
        self.bias = bias

    def select(self, fs: List[float], k: int = None) -> List[int]:
        if not k:
            k = len(fs)

        # Make wheel
        total = 0
```

```
        wheel = []
        for f in fs:
            total += f + self.bias
            wheel += [total]
        # Make pointers
        dist = total / k
        start = random.random() * dist
        pointers = [start + i * dist for i in range(k)]
        # Select
        selected = []
        for p in pointers:
            i = 0
            while wheel[i] < p:
                i += 1
            selected += [i]
        return selected


class TournamentSelection(Selector):
    def __init__(self, n: int):
        self.n = n

    def select(self, fs: List[float], k: int = None) -> List[int]:
        if not k:
            k = len(fs)
        selected = []
        for s in range(k):
            tournament = random.choices(list(enumerate(fs)), k=self.n)
            winner = max(tournament, key=lambda x: x[1])
            selected += [winner[0]]
        return selected


class VirulenceSelector(Selector):
    def __init__(self, selector: Selector, lamb: float, normalise: bool = True):
        self.selector = selector
        self.do_normalise = normalise
        self.lamb = lamb

    def select(self, fs: List[float], k: int = None) -> List[int]:
        fs = np.array(fs)
        if self.do_normalise:
            fs = VirulenceSelector.normalise(fs)
        la = self.lamb
        fs = 2 * fs / la - np.square(fs) / (la * la)
        # Use base selection scheme
        return self.selector.select(fs, k)

    @staticmethod
    def normalise(vals: List[float]):
        mi = min(vals)
        vals = vals - mi
        mx = max(vals)
        if mx > 0:
            vals = vals / mx
        return vals
```

B.7 plot.py

```
"""COMP6202 - Evolution of Complexity
Plotting methods for coevolution report graphs.
"""

__authors__ = "David Jones <dsj1n15@ecs.soton.ac.uk>"

import cv2 as cv
import matplotlib.pyplot as plt
import numpy as np
import os

from typing import List
from matplotlib.pyplot import figure
from representation import Population
from coevolution import Coevolution


class Plotter:
    def __init__(self, show_avgs: bool = True, plot_elites: bool = False):
        self.show_avgs = show_avgs
        self.show_elite_plots = plot_elites

        self.a_color = "darkred"
        self.a_avg_color = "orangered"
        self.b_color = "midnightblue"
        self.b_avg_color = "dodgerblue"
        if not show_avgs:
            self.a_color = self.a_avg_color
            self.b_color = self.b_avg_color

    def make_plot(
        self, coevolution: Coevolution, fig_name: str = None, export_path: str = None
    ) -> figure:
        # Prepare data
        generations = len(coevolution.pops_a)
        total_bits = coevolution.pops_a[-1][0].total_bits
        avgs_x = range(generations)
        raw_x, a_raw, b_raw, a_avgs, b_avgs = ([] for i in range(5))
        for g in range(generations):
            pop_a = list(map(lambda x: x.value, coevolution.pops_a[g]))
            pop_b = list(map(lambda x: x.value, coevolution.pops_b[g]))
            # X axis
            raw_x += [g] * len(pop_b)
            # Append raw data
            a_raw += pop_a
            b_raw += pop_b
            # Record averages
            a_avgs += [np.mean(pop_a)]
            b_avgs += [np.mean(pop_b)]

        # Use LaTeX fonts in the plot
        plt.rc("text", usetex=True)
        plt.rc("font", family="serif")
```

```
w, h = plt.figaspect(0.88)
fig = plt.figure(fig_name, figsize=(w, h))
ax = fig.subplots(3, 1, sharex=True, gridspec_kw={"height_ratios": [16, 1, 1]})
fig.subplots_adjust(hspace=0.11)
# Fix all x axis limits (shared axis)
ax[0].set_xlim([0, generations])

# Configure Objective Score Plot
plt.subplot(ax[0])
ax[0].set_ylim([0, total_bits])
ax[0].tick_params(
    axis="x", which="both", bottom=False, top=False, labelbottom=False
)
ax[0].set_ylabel(r"\textbf{objectv fitness}", fontsize=11)
ax[0].spines["right"].set_visible(False)
plt.hlines(total_bits / 2, 0, generations, colors="black", linewidth=1)

# Plot Raw
plt.scatter(raw_x, a_raw, s=1, c=self.a_color, alpha=1, lw=0)
plt.scatter(raw_x, b_raw, s=1, c=self.b_color, alpha=1, lw=0)
# Plot Averages
if self.show_avgs:
    plt.plot(a_avgs, c=self.a_avg_color, lw=1)
    plt.plot(b_avgs, c=self.b_avg_color, lw=1)

# Make a legend
plt.plot([], [], c=self.a_avg_color, label="$P_A$")
plt.plot([], [], c=self.b_avg_color, label="$P_B$")
plt.legend()
# Plot Subjective Score (A)
plt.subplot(ax[1])
plt.scatter(avgs_x, coevolution.subj_a, c=self.a_avg_color, s=2)
plt.setp(ax[1].get_xticklabels(), visible=False)
ax[1].set_ylim([0, 1])
ax[1].spines["bottom"].set_visible(False)
ax[1].spines["right"].set_visible(False)
ax[1].spines["top"].set_visible(False)
ax[1].tick_params(
    axis="x", which="both", bottom=False, top=False, labelbottom=False
)
# Plot Subjective Score (B)
plt.subplot(ax[2], sharey=ax[1])
plt.scatter(avgs_x, coevolution.subj_b, c=self.b_avg_color, s=2)
ax[2].set_xlabel(r"\textbf{Generations}", fontsize=11)
ax[2].set_ylim([0, 1])
ax[2].figure.text(
    0.05, 0.2, r"\textbf{subj fitness}", fontsize=11, va="center", rotation=90,
)
ax[2].spines["right"].set_visible(False)
ax[2].spines["top"].set_visible(False)

# Plot elites
if self.show_elite_plots:
    self.plot_elites(fig_name + "- Elites [A]", coevolution.pops_a)
    self.plot_elites(fig_name + "- Elites [B]", coevolution.pops_b)
```

```
    # Export
    if export_path:
        folder_path = "/".join(export_path.split("/")[0:-1])
        if folder_path and not os.path.exists(folder_path):
            os.makedirs(folder_path)
        fig.savefig(export_path, bbox_inches="tight", dpi=300)
    return fig

def plot_elites(self, fig_name: str, pops: List[Population]):
    elites = [max(pop, key=lambda x: x.value) for pop in pops]
    bitmap = None
    for elite in elites:
        bits = elite.as_bits()
        if bitmap is None:
            bitmap = bits
        else:
            bitmap = np.vstack((bitmap, elite.as_bits()))
    bitmap = bitmap.astype(float)
    show_bitmap = cv.resize(
        bitmap, (0, 0), fx=7.5, fy=0.25, interpolation=cv.INTER_NEAREST
    )
    cv.imshow(fig_name, show_bitmap)
```

B.8 utils.py

```python
import random


def seed_random(seed: int, use_fixed: bool) -> int:
    if not use_fixed:
        import sys  # noqa

        seed = random.randrange(sys.maxsize)
    random.seed(seed)
    return seed
```