Midterm

Artificial Intelligence

Part B : GA Experiments

Introduction

I think I should start by saying tweaking the fitness function, hyper parameters and genome settings can be done for an endless time span. I experimented with different fitness functions that included not only the distance travelled from the origin and/or the over all distance travelled but also the movement in each simulation step and all of those in different combinations. To sum these experiments up I was not satisfied with the outcome of any combined value so I reverted to the distance from zero for the following simulations. One approach to include multiple values would have been to return multiple values from the fitness function but that in turn would have made it necessary to adapt the rest of the code as well which was in the time available not feasible.

To run a simulation I stepped away from the unittest version presented in the lectures and wrote a dedicated script that is capable of reading a configuration file in JSON format, does some error handling when necessary and has helpful logging options. For the code please see the "Code Listing" section below.

The configuration file (see below for an example) allows for defining the simulation parameters (compute core pool size, population size, gene count and generation count) and the mutation parameters (point mutation rate and amount, shrink and grow mutation rate). A new parameter is if elitism is enabled or disabled for the simulation run.

To streamline the simulation runs with different hyper parameters 16 configuration files have been created. For each configuration file three simulation runs were attempted.

A (bash) shell script (see below) was written to put it all together and handle the output.

The analysis happened with another python script that iterated over the experiment directories, analysing the log files and producing the necessary statistical and graphical output.

The defaults for each simulation run were:

- Population size of 128 individuals
- Initial gene count of 6
- 100 Generations per simulation
- one config with elitsim enabled and one with elitism disabled

Each simulation configuration was run three times on a machine with 128 compute cores. It took 18h to finish.

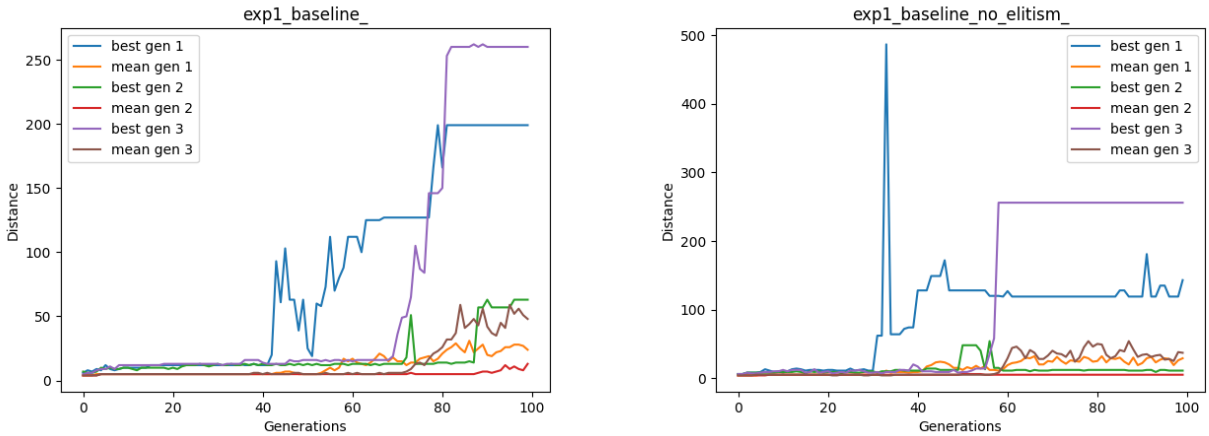
Hyperparameter Tweaks

To establish a baseline the following parameters were chosen:

- Point mutation rate 0.2 and mutation amount 0.1
- Shrink mutation rate 0.2

- Grow mutation rate 0.2

In these runs a clear trend that the creatures are improving their ability to move is visible. With elitism starting around generation 80. Interesting to see is that the increase in movement ability came in earlier generations without elitism. Even though runs with and without elitism show individuals from two populations that have outstanding fitness the mean of the population fits into the range other populations evaluated to.



(a) with elitism

(b) without elitism

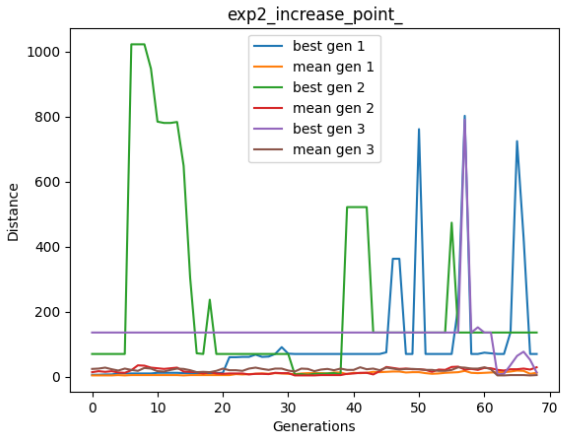
Figure 1: Baseline

In the next two runs only the point mutation rate and and amount have been modified:

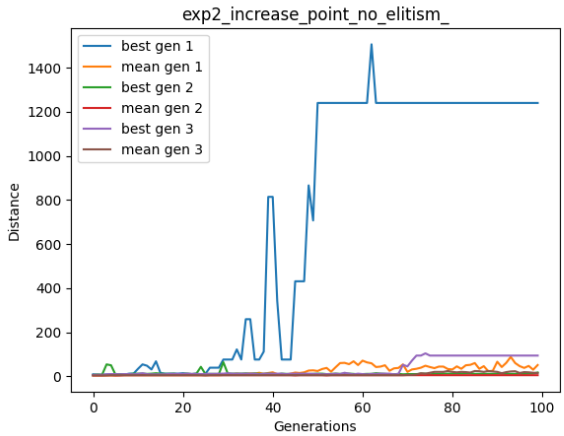
- Point mutation rate **0.4** and mutation amount **0.2**
- Shrink mutation rate 0.2
- Grow mutation rate 0.2

The first observation in the runs with increased point mutation rate and amount is that it resulted in individuals with much higher fitness. However the population mean still resides in the same range as the baseline experiment populations. Elitism doesn't seem to have an impact as the fitter individuals come and go and do not live longer than a couple of generations.

Increasing the point mutation rate even more came up with an interesting result. With and without elitism the mean population fitness was increased. Significantly in the runs without elitism. As expected was a higher frequency of capability changes in the fittest individuals due to the high mutation rate.



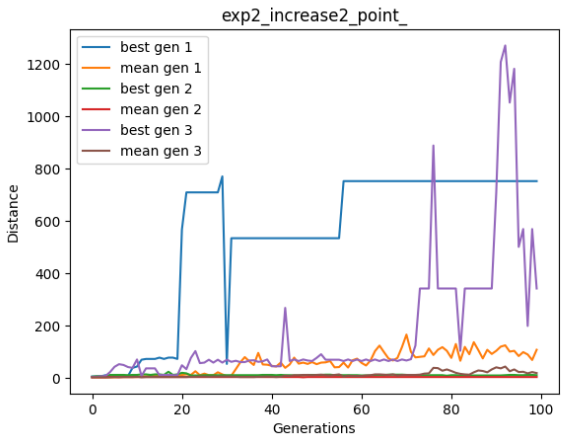
(a) with elitism



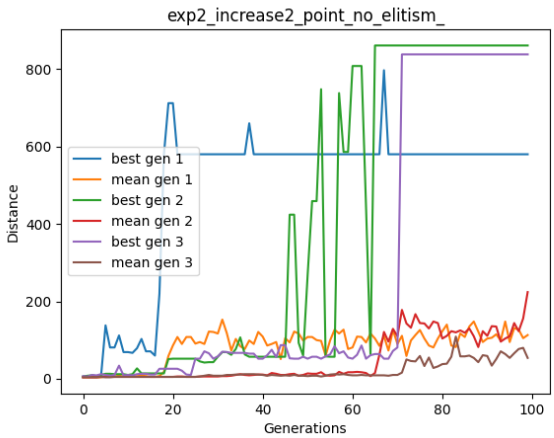
(b) without elitism

Figure 2: Point Mutation Increase

- Point mutation rate **0.6** and mutation amount **0.2**
- Shrink mutation rate 0.2
- Grow mutation rate 0.2



(a) with elitism



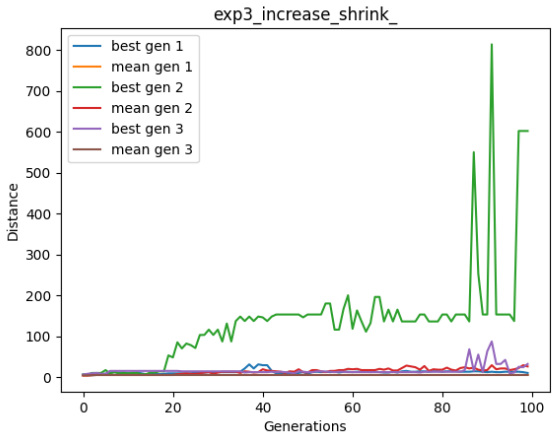
(b) without elitism

Figure 3: Point Mutation Increase

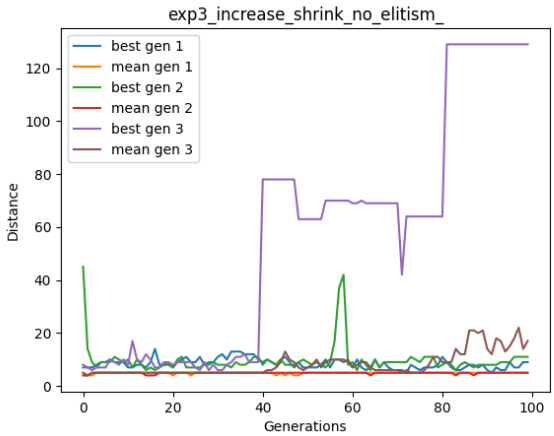
Shrink mutation rate increase

- Point mutation rate 0.2 and mutation amount 0.1
- Shrink mutation rate **0.4**
- Grow mutation rate 0.2

It's clearly visible that shrinking a genome in this experiment lead to mediocre individuals. Even the fittest were not outstanding. But a second increase in the chance of loosing a gene didn't make the over all outcome worse. In these runs elitism did have a positive effect as it can be seen that the mean population fitness rose toward the end.



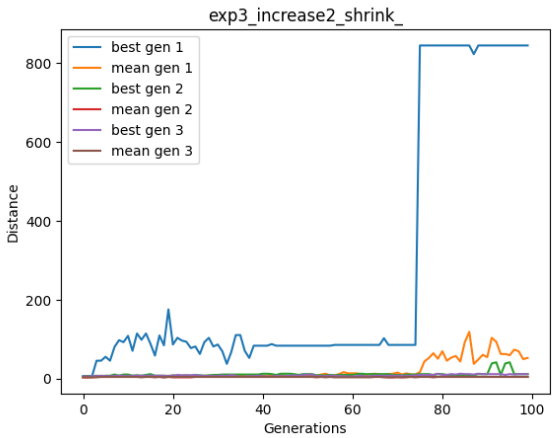
(a) with elitism



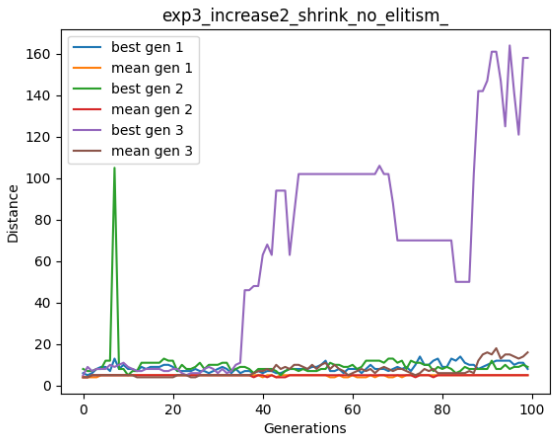
(b) without elitism

Figure 4: Shrink Mutation Increase

- Point mutation rate 0.2 and mutation amount 0.1
- Shrink mutation rate **0.6**
- Grow mutation rate 0.2



(a) with elitism



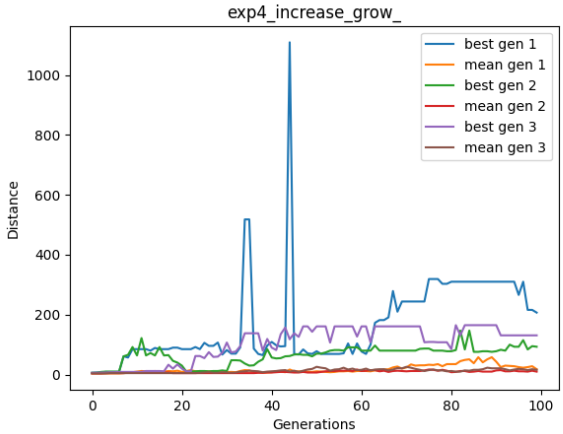
(b) without elitism

Figure 5: Shrink Mutation Increase

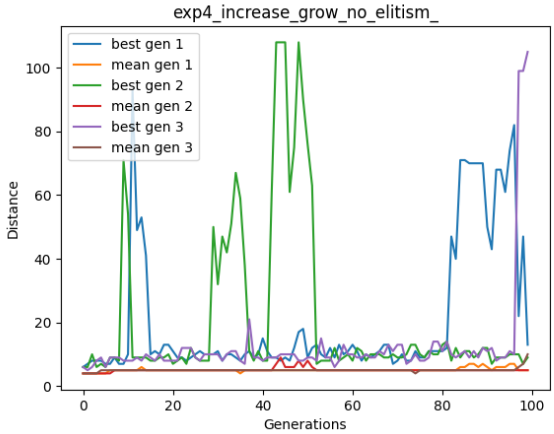
Grow mutation rate increase

With the exception of two outlier individuals growing the genome seemed to have a stabilizing effect where the best individuals fitness didn't undulate as much as in the other experiments. However the over all performance of individuals wasn't improved.

- Point mutation rate 0.2 and mutation amount 0.1
- Shrink mutation rate 0.2
- Grow mutation rate **0.4**



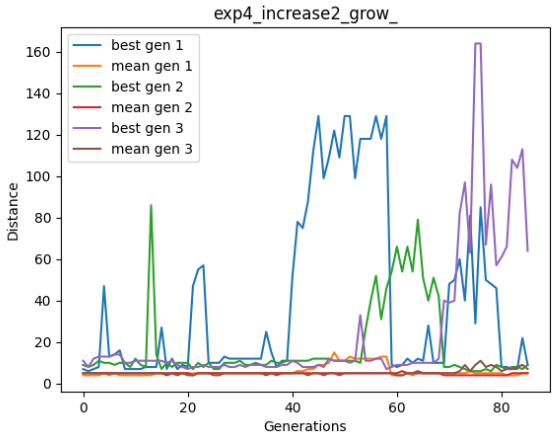
(a) with elitism



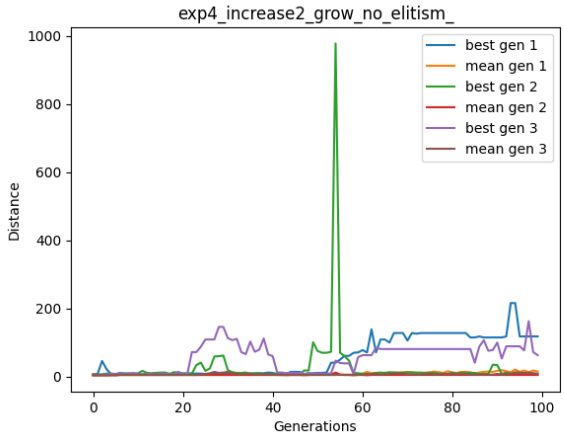
(b) without elitism

Figure 6: Grow Mutation Increase

- Point mutation rate 0.2 and mutation amount 0.1
- Shrink mutation rate 0.2
- Grow mutation rate **0.6**



(a) with elitism



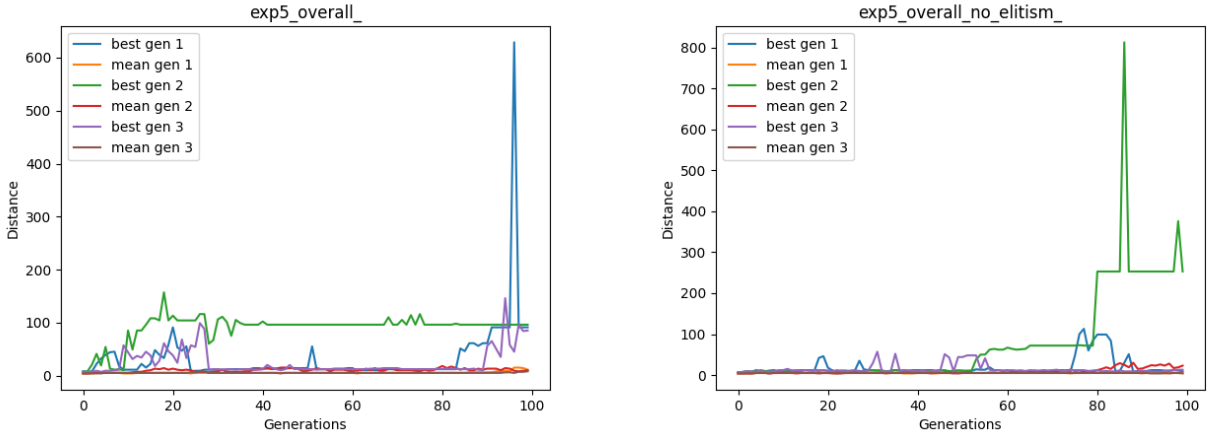
(b) without elitism

Figure 7: Grow Mutation Increase

Over all mutation rate increase

- Point mutation rate **0.3** and mutation amount **0.25**
- Shrink mutation rate **0.3**
- Grow mutation rate **0.3**

Slightly increasing all mutation parameters in comparison the the baseline runs lead to much fitter individuals but the fitness mean was not changed at all.



(a) with elitism

(b) without elitism

Figure 8: Mutation Increase Overall

In conclusion it can be said that the best effect for the mean fitness of all individuals was to increase the point mutation rate and amount. It's clear that the very limited number of experiments doesn't shed much light on what the exact parameters would have to be. In that regard more research with gradually increasing point mutation rate and amount is necessary to find the optimal values.

As nature is rarely this stable over so many generations it could be argued that starting out with a stable parameter combination and then varying the hyperparameters dynamically during a 500 to 1000 generations simulation might bring out some interesting creature developments.

Genome Tweaks

In the lecture only a cylinder was covered as possible shape in the link creation. The genome is already capable of handling different values so in this case it was an extension to the phenotype rather than the genotype. I extended the `to_link_element` method to produce boxes and spheres as well as cylinders. The mass is calculated dynamically for each shape.

The `gene_spec` dictionary specifies the available options of a gene. Each of those options has a scaling factor. For some options it doesn't make sense to change the scaling factor. In example the above mentioned *link-shape* is a value between 0 and 1 and it is scaled with 1 (equalling the original value). The `to_link_element` method expects values between 0 and 1 to distinguish between the three shapes (0-0.33 box, 0.33-0.66 cylinder, 0.66-1 sphere). In this case it does not make sense to scale the parameter because it would be outside the expected range and would not result in any shapes drawn. Other parameters like the joint properties are hard coded so changing those values doesn't have any effect. As a note: it would be easy to change the link generation method to use the available parameters dynamically but to get viable results this would need many more experiments.

But there are parameters like *link-length*, *link-radius*, *link-recurrence* that change the appearance of the creatures dramatically. The impact of the scaling factor is it to adjust the gene value that originally lies between 0 and 1 to factor times this value. In case of radius and length the creatures get much larger whereas the recurrence factor changes the complexity of the creature.

Scaling the link element sizes certainly lead to some interesting creatures but comes with problems. The current implementation drops each creature from a certain height to make sure the physics engine doesn't go bonkers with creatures "stuck" in the floor (which would be the default position). When the scaling factor(s) gets to high the currently static drop height is not sufficient any more and the same problem as when starting without a drop occurs again, giving the creature an unfair advantage. Another side note: During the evolution runs for the mutation experiments I encountered creatures that used the default drop to make a huge first leap. Subsequent forward motion was

much less impressive. To make it truly fair one would need to position the creature exactly on the floor.

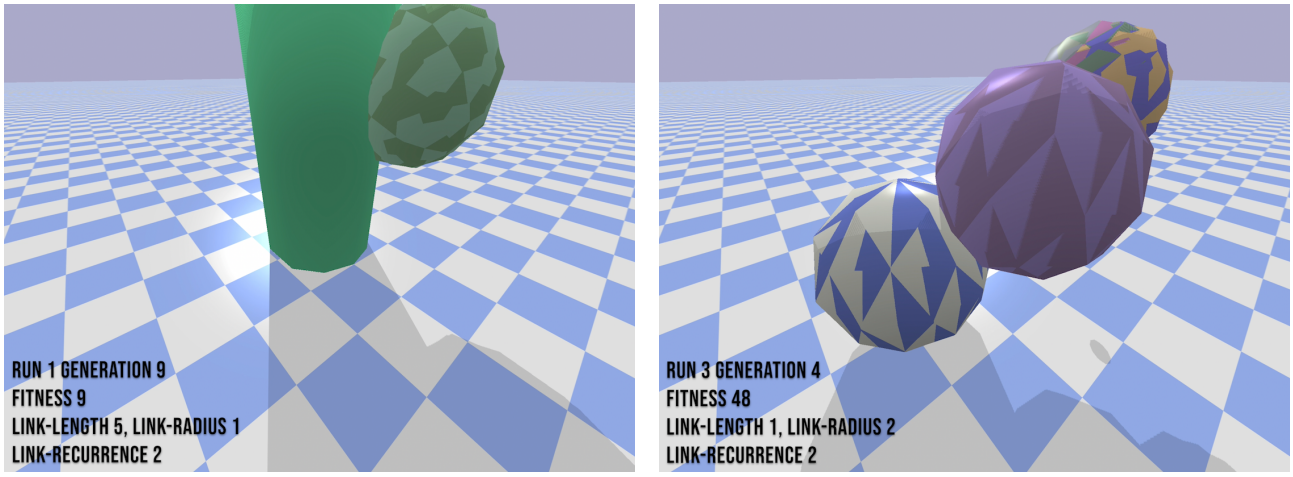


Figure 9: Genome Tweaks

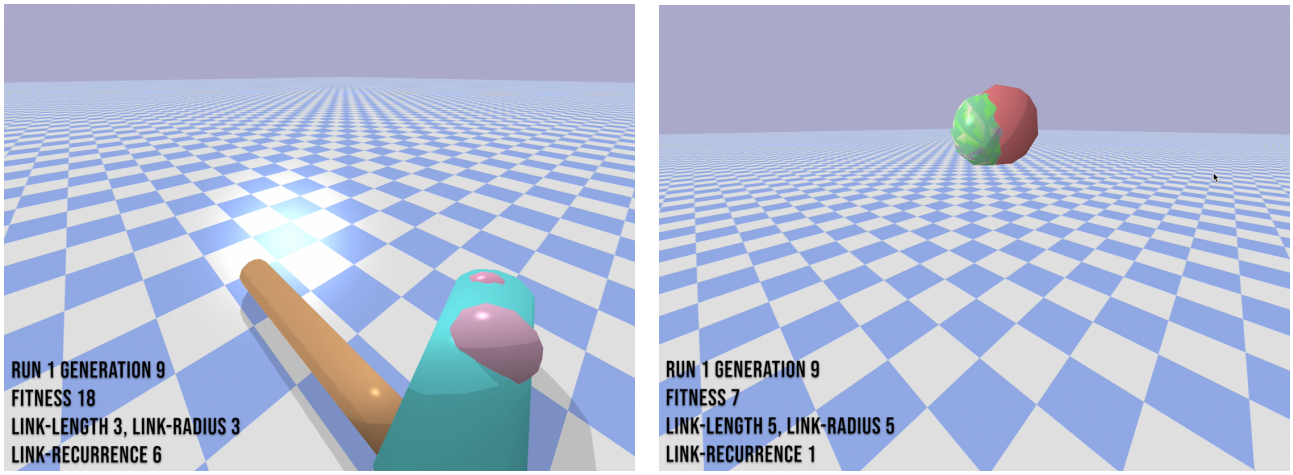


Figure 10: Genome Tweaks

More evolved creatures from the mutation and genome experiments can be seen in the video.

Code Listing

`run.sh`

The following (bash) Shell script takes each simulation configuration (see below), runs it three times and stores the output in a dedicated subdirectory.

```
#!/bin/bash

# number of runs with the same configuration
run_count=3

# iterate over all configs
for config in *.json
do
    for (( r=1 ; r<=$run_count ; r++ ))
    do
        # get config name without extension
        base=$(basename $config .json)
        run_dir="${base}_$r"

        # run the simulation
```

```
        python run_ga.py $config 2>&1 | tee run.log

        # create directory for simulation output
        mkdir "$run_dir"

        # move simulation output
        mv elite*.csv run.log "$run_dir"
    done
done
```

exp1_baseline.json

This is an example simulation configuration file in JSON format.

```
{
    "pool_size": 128,
    "pop_size": 128,
    "gene_count": 6,
    "generations": 100,
    "mutation_point_rate": 0.2,
    "mutation_point_amount": 0.1,
    "mutation_shrink_rate": 0.2,
    "mutation_grow_rate": 0.2,
    "elitism": true
}
```

run_ga.py

Instead of the unittest version of the GA run I wrote a dedicated script for running the simulation with variable configuration files and logging.

```
from population import Population
from simulation import ThreadedSim
from genome import Genome
from creature import Creature
import numpy as np
import logging
import json
import sys
import os

# default configuration
config = {
    # number of CPU cores to use
    'pool_size' : 8,
    # size of the population
    'pop_size' : 20,
    # how many genes each individual has
    'gene_count' : 6,
    # how many generations should be evolved
    'generations' : 100,

    # mutation settings
    'mutation_point_rate' : 0.3,
    'mutation_point_amount' : 0.2,
    'mutation_shrink_rate' : 0.25,
    'mutation_grow_rate' : 0.25,

    # elitism of the fittest [True|False]
    'elitism' : True
}

# check if there is a command line argument
if len(sys.argv) < 2:
    print(sys.argv[0] + ' <config.json>')
    sys.exit(1)

# get the first argument
config_file = sys.argv[1]
```

```python

# check if the file exists
if not os.path.exists(config_file):
    print('The given config file does not exist.')
    sys.exit(1)

# open and read the file
with open(config_file, 'r') as f:
    new_config = json.load(f)

# update the default config
config.update(new_config)

# fix value types
config = {
    # number of CPU cores to use
    'pool_size' : int(config['pool_size']),
    # size of the population
    'pop_size' : int(config['pop_size']),
    # how many genes each individual has
    'gene_count' : int(config['gene_count']),
    # how many generations should be evolved
    'generations' : int(config['generations']),

    # mutation settings
    'mutation_point_rate' : float(config['mutation_point_rate']),
    'mutation_point_amount' : float(config['mutation_point_amount']),
    'mutation_shrink_rate' : float(config['mutation_shrink_rate']),
    'mutation_grow_rate' : float(config['mutation_grow_rate']),

    # elitism of the fittest [True|False]
    'elitism' : bool(config['elitism'])
}

# set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(
    message)s', datefmt='%d-%b-%y %H:%M:%S')

# log the configuration
logging.info('Pool Size %s', str(config['pool_size']))
logging.info('Population Size %s', str(config['pop_size']))
logging.info('Gene Count %s', str(config['gene_count']))
logging.info('Point Mutation Rate %s', str(config['mutation_point_rate']))
logging.info('Point Mutation Amount %s', str(config['mutation_point_amount']))
logging.info('Point Shrink Rate %s', str(config['mutation_shrink_rate']))
logging.info('Point Grow Rate %s', str(config['mutation_grow_rate']))
logging.info('Elitism %s', str(config['elitism']))

# setup population and simulation
logging.info('Initializing Pool')
pop = Population(pop_size=config['pop_size'], gene_count=config['gene_count'])
logging.info('Initializing Simulation')
sim = ThreadedSim(pool_size=config['pool_size'])

# evolve
for generation in range(config['generations']):
    sim.eval_population(pop, 2400)
    fits = [cr.get_distance_travelled() for cr in pop.creatures]
    links = [len(cr.get_expanded_links()) for cr in pop.creatures]
    fitmap = Population.get_fitness_map(fits)

    logging.info('Generation: %s, max: %s, mean: %s, max links: %s, mean links: %s',
        str(generation),
        str(np.round(np.max(fits))),
        str(np.round(np.mean(fits))),
        str(np.round(np.max(links))),
```

```
        str(np.round(np.mean(links)))
    )

    # find and extract the fittest (elitism)
    fmax = np.max(fits)
    for cr in pop.creatures:
        if cr.get_distance_travelled() == fmax:
            elite = cr
            break

    # temporary list of the new population
    new_generation = []
    # iterate over all creatures
    for creature_index in range(len(pop.creatures)):
        # select parents
        p1_ind = Population.select_parent(fitmap)
        p2_ind = Population.select_parent(fitmap)

        logging.debug('Parent 1 Genome:\n%s', str(pop.creatures[p1_ind].dna))
        logging.debug('Parent 2 Genome:\n%s', str(pop.creatures[p2_ind].dna))

        # crossover
        dna = Genome.crossover(
            pop.creatures[p1_ind].dna,
            pop.creatures[p2_ind].dna
        )

        logging.debug('Post Crossover:\n%s', str(dna))

        # mutate
        dna = Genome.point_mutate(dna, config['mutation_point_amount'], config['
    mutation_point_rate'])
        logging.debug('Post Point Mutation:\n%s', str(dna))
        dna = Genome.grow_mutate(dna, config['mutation_grow_rate'])
        logging.debug('Post Grow Mutation:\n%s', str(dna))
        dna = Genome.shrink_mutate(dna, config['mutation_shrink_rate'])
        logging.debug('Post Shrink Mutation:\n%s', str(dna))

        # create a dummy creature and replace genome
        cr = Creature(1)
        cr.update_dna(dna)

        # add dummy to the new population
        new_generation.append(cr)

    # replace the first in the generation with the elite (if enabled)
    if config['elitism']:
        new_generation[0] = elite

    # save the fittest of this generation (the elite)
    Genome.to_csv(elite.dna, 'elite_' + str(generation) + '.csv')

    # update the population
    pop.creatures = new_generation

logging.info('Simulation done')
```

`analysis.py`

The following code analyses the log file, produces the graphs and prints some statistical details.

```
import glob
import math
import matplotlib.pyplot as plt
import numpy as np
import re

# plot and statistic output method
```

```python
# takes the experiment directory name and optionally how many runs (default = 3)
def analyse(dirname, runs=3):
    # number of log files found
    file_count = 0
    # regular expression matches
    matches = []

    # find all log files
    for file in glob.iglob(f'GAResults/{dirname}[1,2,3]/*.log'):
        # read the file
        with open(file, 'r') as f:
            data = f.read()

        # use a regex to extract details for each generation from this run
        m = re.findall(r'Generation: \d+, max: (\d+\.\d), mean: (\d+\.\d), max links: (\d+), mean links: (\d+\.\d)', data)

        matches.extend(m)
        file_count += 1

    # convert re match array to numpy array with floats
    a = np.array(matches, dtype=float)

    # extract the mean and max values
    mean_val = np.mean(a, axis=0)
    max_val = np.max(a, axis=0)

    # calculate how many generations have been run
    gens = int(len(matches) / runs)

    # find the generation that has the individuum with the highes score
    run = None
    gen = None
    for i in range(0, len(matches)):
        if float(matches[i][0]) == float(max_val[0]):
            run = math.ceil(i / gens)
            gen = i % gens

    # plot x axis
    x = np.arange(0, gens)

    # clear the plot
    plt.clf()

    # add the data for each simulation run to the plot
    labels = []
    for i in range(0, runs):
        start = int(0 + gens * i)
        stop = int(gens + gens * i)
        generation = i + 1
        y = a[start:stop, [0,1]]
        labels.append('best gen ' + str(generation))
        labels.append('mean gen ' + str(generation))
        plt.plot(x, y)

    # add some more information to the plot
    plt.legend(labels)
    plt.title(dirname)
    plt.xlabel('Generations')
    plt.ylabel('Distance')

    # save the plot
    plt.savefig(dirname + '.png')

    # statistical information
    print(dirname, 'mean', mean_val[0], 'best', max_val[0], 'generation', gen, 'run',
```

```
    run)

# experiment output directories
dirnames = [
    'exp1_baseline_',
    'exp1_baseline_no_elitism_',
    'exp2_increase_point_',
    'exp2_increase_point_no_elitism_',
    'exp2_increase2_point_',
    'exp2_increase2_point_no_elitism_',
    'exp3_increase_shrink_',
    'exp3_increase_shrink_no_elitism_',
    'exp3_increase2_shrink_',
    'exp3_increase2_shrink_no_elitism_',
    'exp4_increase_grow_',
    'exp4_increase_grow_no_elitism_',
    'exp4_increase2_grow_',
    'exp4_increase2_grow_no_elitism_',
    'exp5_overall_',
    'exp5_overall_no_elitism_'
]

# iterate over all experiment directories, produce the graphs and statistical output
for dir in dirnames:
    analyse(dir)
```

Legal

Intro image in the video is from Unsplash - Markus Spisk, audio is from Free Music Archive - A. Shamaluev. Both are CC free use.