assignment

November 22, 2021

1 Intelligent Systems Seminar Assignment 1

Genetic algorith written in python

Authors: *Nejc Jezeršek, Žan Oberstar*

1.0.1 Libraries

[70]:
```
import statistics
import math
import random
from matplotlib import pyplot as plt
import numpy as np
```

[71]:
```
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')
```

1.1 Population generation

Agent is an arithmetic expression constructed of numbers and symbols specified below.

Each number can be used `max_number_use` times, therefore the expression can contain from `1` to `len(numbers)*max_number_use` numbers.

After each number there is a symbol from `symbols`. The symbol after the last number is ignored, but may be used in future generations like an unexpressed gene in a human genome. That decision was made primarily to make crossover (based on permutations) easier to implement.

[72]:
```
numbers = ['10', '25', '100', '5', '3']
symbols = ['+', '-', '*', '/']

max_number_use = 1
min_numbers_in_expresion = 1

target_value = 2512
```

[73]:
```
def generate_agent():
    n_numbers = random.randint(min_numbers_in_expresion,␣
 ↪len(numbers)*max_number_use)
    selected_numbers = random.sample(numbers*max_number_use, n_numbers)
```

```
    return [(number, random.choice(symbols)) for number in selected_numbers]

def agent_to_expression(agent):
        expression = "".join(n + s for n, s in agent)
        return expression[:-1] # remove last operator

def evaluate_agent(agent):
    expression = agent_to_expression(agent)
    if len(expression) == 0: return 0
    else: return eval(expression)

def agent_to_string(agent):
    return f"{agent_to_expression(agent)} = {evaluate_agent(agent):.2f}"
```

Agent is constructed of tuples of a number and a symbol as seen below: - `generate_agent` generates a random agent - `agent_to_expression` function joins the tuples together into a string, ignoring the last symbol - `evaluate_agent` generates the expression and evaluates it - `agent_to_string` creates a string with expression and its value

[74]:
```
a = generate_agent()
print("Generate agent   : ", a)
print("Agent expression : ", agent_to_expression(a))
print("Evaluated        : ", evaluate_agent(a))
print("Pretty print     : ", agent_to_string(a))
```

```
Generate agent   :  [('25', '*')]
Agent expression :  25
Evaluated        :  25
Pretty print     :  25 = 25.00
```

Here are some randomly generated agents:

[75]:
```
for i in range(20):
    a = generate_agent()
    print(f"{i}. ", agent_to_string(a))
```

```
0.  3 = 3.00
1.  3*10+100 = 130.00
2.  10 = 10.00
3.  10+3 = 13.00
4.  10 = 10.00
5.  100/10/25 = 0.40
6.  5-100*25 = -2495.00
7.  25-100-5/10 = -75.50
8.  3-25+10+5 = -7.00
9.  5-3/10-100-25 = -120.30
10.  10 = 10.00
11.  25+10-3 = 32.00
```

```
12.  3-25+5 = -17.00
13.  25-5 = 20.00
14.  5 = 5.00
15.  25*10+100/3 = 283.33
16.  100/10+5 = 15.00
17.  5-100*3 = -295.00
18.  100 = 100.00
19.  25+100*5 = 525.00
```

1.2 Fitness

Fitness is used as a scoring function for the genetic algorithm to determine the best agents to use for reproduction.

We defined some different fitness functions, which will be compared in section *Evaluation*: - `fitness_abs` returns the negative of an absolute value of the error - `fitness_inverse` returns 1 over an absolute value of the error - `fitness_squared` returns negative of the error squared - `fitness_log` returns negative of the logarithm of the error

[76]:
```
def error(agent):
    return abs(evaluate_agent(agent) - target_value)

def fitness_abs(agent):
    return -error(agent)

def fitness_inverse(agent):
    return 1/(error(agent) + 0.0001) # + 0.0001 to avoid zero devision

def fitness_squared(agent):
    return -(error(agent)**2)

def fitness_log(agent):
    return -math.log2(error(agent)+1)
```

1.3 Mutation

We defined 7 different types of mutation: - `mutation_change_symbol` changes a random symbol - `mutation_change_number` changes a random number - `mutation_add_element` adds a new element to the agent - `mutation_remove_element` removes a random element from the agent - `mutation_swap_symbol` swaps two random symbols
- `mutation_swap_number` swaps two random numbers - `mutation_swap_element` swaps two random elements (i.e. tuples consistion of a number and a symbol ("5", "+"))

We add function `mutation` so that it randomly chooses between all of the mutations listed above

[77]:
```
def mutation_change_symbol(agent):
    a = agent.copy()
    i = random.randrange(len(a))
    number, _ = a[i]
```

```
    a[i] = (number, random.choice(symbols))

    return a

def mutation_change_number(agent):
    a = agent.copy()
    agent_numbers = [number for number, _ in a]
    # choose numbers that can be added without violating `max_number_use`
    candidate_numbers = list(filter(lambda n: agent_numbers.count(n) <␣
↪max_number_use, numbers))
    if len(candidate_numbers) == 0: return a
    i = random.randrange(len(a))
    _, symbol = a[i]
    a[i] = (random.choice(candidate_numbers), symbol)

    return a

def mutation_add_element(agent):
    a = agent.copy()
    agent_numbers = [number for number, _ in a]
    # choose numbers that can be added without violating `max_number_use`
    candidate_numbers = list(filter(lambda n: agent_numbers.count(n) <␣
↪max_number_use, numbers))
    if len(candidate_numbers) == 0: return a
    a.insert(random.randint(0, len(a)), (random.choice(candidate_numbers),␣
↪random.choice(symbols)))

    return a

def mutation_remove_element(agent):
    a = agent.copy()
    if len(a) <= 1: return a # nothing to remove
    del a[random.randrange(len(a))]

    return a

def mutation_swap_symbol(agent):
    a = agent.copy()
    i = random.randrange(len(agent))
    j = random.randrange(len(agent))

    a[i], a[j] = (a[i][0], a[j][1]), (a[j][0], a[i][1])

    return a

def mutation_swap_number(agent):
    a = agent.copy()
```

```
    i = random.randrange(len(agent))
    j = random.randrange(len(agent))

    a[i], a[j] = (a[j][0], a[i][1]), (a[i][0], a[j][1])

    return a

def mutation_swap_element(agent):
    a = agent.copy()
    i = random.randrange(len(agent))
    j = random.randrange(len(agent))

    a[i], a[j] = a[j], a[i]

    return a

def mutation(agent):
    mutation_functions = [mutation_change_symbol, mutation_change_number,␣
 ↪mutation_add_element, mutation_remove_element,
                          mutation_swap_number, mutation_swap_symbol,␣
 ↪mutation_swap_element]
    return random.choice(mutation_functions)(agent)
```

Here is an example of all the mutations made on the same agent:

[78]:
```
agent = generate_agent()
print("Generated agent : ", agent)
print("Change symbol   : ", mutation_change_symbol(agent))
print("Change number   : ", mutation_change_number(agent))
print("Add element     : ", mutation_add_element(agent))
print("Remove element  : ", mutation_remove_element(agent))
print("Swap symbol     : ", mutation_swap_symbol(agent))
print("Swap number     : ", mutation_swap_number(agent))
print("Swap element    : ", mutation_swap_element(agent))
print("Random mutation : ", mutation(agent))
```

```
Generated agent :  [('25', '-'), ('5', '+'), ('100', '-')]
Change symbol   :  [('25', '-'), ('5', '*'), ('100', '-')]
Change number   :  [('3', '-'), ('5', '+'), ('100', '-')]
Add element     :  [('25', '-'), ('10', '-'), ('5', '+'), ('100', '-')]
Remove element  :  [('5', '+'), ('100', '-')]
Swap symbol     :  [('25', '-'), ('5', '+'), ('100', '-')]
Swap number     :  [('100', '-'), ('5', '+'), ('25', '-')]
Swap element    :  [('5', '+'), ('25', '-'), ('100', '-')]
Random mutation :  [('25', '-'), ('5', '+'), ('100', '-')]
```

1.4 Crossover

Crossover takes 2 parents: `parent1` and `parent2`.
It then splits `parent1` and `parent2` into 2 parts and takes one part from `parent1` and another from `parent2` and stiches them together to create `child1`.
It does the same for `child2`, just that it takes the other 2 parts.

1.4.1 Functions

[147]:
```
def agent_subtract(agent1, agent2):
    agent2_numbers = [n for n, _ in agent2]
    res = []
    for number, symbol in agent1:
        if number in agent2_numbers: agent2_numbers.remove(number)
        else: res.append((number, symbol))
    return res

def crossover(parent1, parent2):
    min_len = min(len(parent1), len(parent2))
    selection_start = random.randint(0, min_len)
    selection_end = random.randint(selection_start, min_len)

    child1 = parent1[selection_start:selection_end]
    child2 = parent2[selection_start:selection_end]
    child1.extend(agent_subtract(parent2, child1))
    child2.extend(agent_subtract(parent1, child2))

    return (child1, child2)
```

1.4.2 Step by step example

We first generate two parents `parent1` and `parent2`:

[133]:
```
parent1 = generate_agent()
parent2 = generate_agent()
print("parent1: ", parent1, agent_to_string(parent1))
print("parent2: ", parent2, agent_to_string(parent2))
```

```
parent1:  [('5', '*'), ('100', '-'), ('25', '/'), ('10', '*'), ('3', '+')]
5*100-25/10*3 = 492.50
parent2:  [('5', '*'), ('10', '*')] 5*10 = 50.00
```

We then select a section that overlaps both parents and initializes `child1` to a section of `parent1` and `child2` to a section of `parent2`.

[143]:
```
min_len = min(len(parent1), len(parent2))
selection_start = random.randint(0, min_len)
selection_end = random.randint(selection_start, min_len)
```

```
child1 = parent1[selection_start:selection_end]
child2 = parent2[selection_start:selection_end]

print("child1: ", child1, agent_to_string(child1))
print("child2: ", child2, agent_to_string(child2))
```

```
child1:  [('5', '*'), ('100', '-')] 5*100 = 500.00
child2:  [('5', '*'), ('10', '*')] 5*10 = 50.00
```

The rest of the elements come from the other parent by removing the numbers that already came from the first parent.
We do that using the function `agent_subtract`, so that we stay within the bounds of the problem (all of the numbers can only be used once)

[144]:
```
parent1_remainder = agent_subtract(parent1, child2)
parent2_remainder = agent_subtract(parent2, child1)

print("parent1_remainder: ", parent1_remainder,␣
 ↪agent_to_string(parent1_remainder))
print("parent2_remainder: ", parent2_remainder,␣
 ↪agent_to_string(parent2_remainder))
```

```
parent1_remainder:  [('100', '-'), ('25', '/'), ('3', '+')] 100-25/3 = 91.67
parent2_remainder:  [('10', '*')] 10 = 10.00
```

In the end we stich both children together using the `parent_remainder`.

[145]:
```
child1.extend(parent2_remainder)
child2.extend(parent1_remainder)

print("child1 result: ", child1, agent_to_string(child1))
print("child2 result: ", child2, agent_to_string(child2))
```

```
child1 result:  [('5', '*'), ('100', '-'), ('10', '*')] 5*100-10 = 490.00
child2 result:  [('5', '*'), ('10', '*'), ('100', '-'), ('25', '/'), ('3', '+')]
5*10*100-25/3 = 4991.67
```

Example of the `crossover` function:

[149]:
```
child1, child2 = crossover(parent1, parent2)
print(f"parent1 : {agent_to_string(parent1)}; lenght: {len(parent1)}")
print(f"parent2 : {agent_to_string(parent2)}; length: {len(parent2)}")
print(f"child1  : {agent_to_string(child1)}; length: {len(child1)}")
print(f"child2  : {agent_to_string(child2)}; length: {len(child2)}")
```

```
parent1 : 5*100-25/10*3 = 492.50; lenght: 5
parent2 : 5*10 = 50.00; length: 2
child1  : 100-5*10 = 50.00; length: 3
child2  : 10*5*100-25/3 = 4991.67; length: 5
```

1.5 Selection

1.5.1 Top agents

The first naive approach is to order the agents by their fitness and select the top n agents.

[85]:
```
def select_best_agents(population_with_fitness, n_agents):
    return list(sorted(population_with_fitness, key=lambda e: e[1],␣
↪reverse=True))[:n_agents]
```

Lets create a 100 agents and select the best 10.

[86]:
```
population = [generate_agent() for _ in range(100)]
population_with_fitness = [(a, fitness_log(a)) for a in population]
population_fitness = [f for a, f in population_with_fitness]

best_agents = select_best_agents(population_with_fitness, 10)

for a, f in best_agents:
    print(f"error: {error(a):.3f}, fitness: {f:.3f}, agent:␣
↪{agent_to_string(a)}, ")
```

```
error: 12.000, fitness: -3.700, agent: 25*100 = 2500.00,
error: 27.000, fitness: -4.807, agent: 25*100-3*5 = 2485.00,
error: 488.000, fitness: -8.934, agent: 3*10*100 = 3000.00,
error: 1262.000, fitness: -10.303, agent: 25*10*5 = 1250.00,
error: 1512.000, fitness: -10.563, agent: 10*100 = 1000.00,
error: 1517.000, fitness: -10.568, agent: 10*100-5 = 995.00,
error: 1667.000, fitness: -10.704, agent: 100-5+25*10*3 = 845.00,
error: 2009.000, fitness: -10.973, agent: 3+5*100 = 503.00,
error: 2095.333, fitness: -11.034, agent: 10/3*25*5 = 416.67,
error: 2192.000, fitness: -11.099, agent: 25+100*3-10+5 = 320.00,
```

1.5.2 Weighted choice

Another approach is to compute wheits based on the fitness value and than make a random weighted choice of n agents.

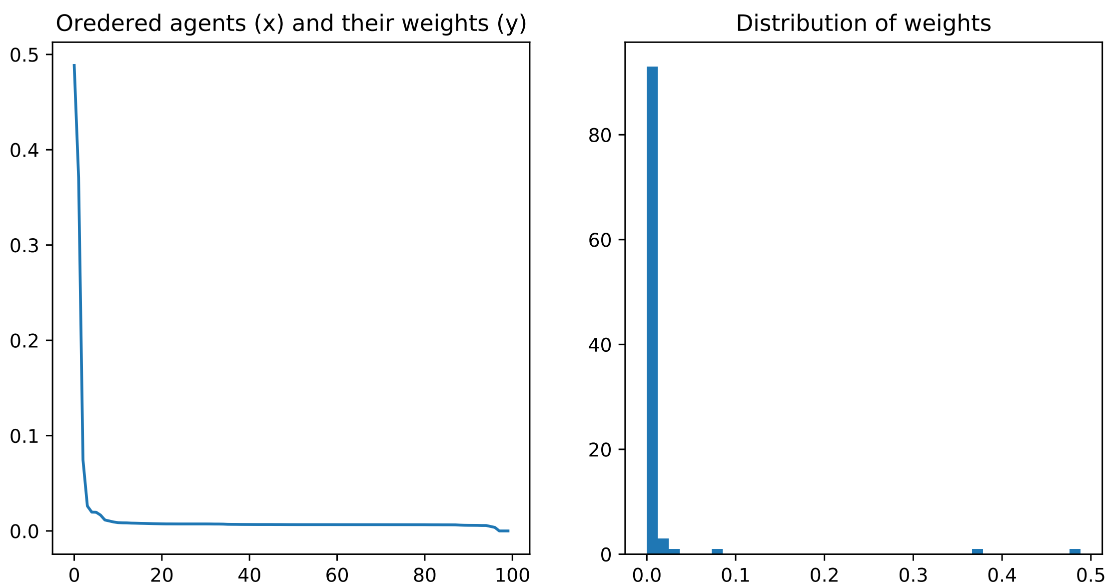
[87]:
```
def get_weights(population_with_fitness, verbose=False):
    k = 1.0 # if k is close to 1 best agents are more favored if k > 1 the␣
↪difference in weights between different agents gets smaller
    population_fitness = [f for a, f in population_with_fitness]
    min_fitness = min(population_fitness)
    base_fitness = -min_fitness*k
    population_with_weights = [(a, f, ((f+base_fitness)/base_fitness)**2) for␣
↪a, f in population_with_fitness]
    population_with_weights_ordered = sorted(population_with_weights,␣
↪key=lambda e: e[2], reverse=True)
```

```
    if verbose:
        for a, f, w in population_with_weights_ordered[:10]:
            print(f"fitness: {f:.3f}, weight: {w:.3f}, agent:␣
↪{agent_to_string(a)}")

        weights_ordered = [w for a, f, w in population_with_weights_ordered]
        plt.figure(figsize=(10, 5))
        plt.subplot(1,2,1)
        plt.title("Oredered agents (x) and their weights (y)")
        plt.plot(weights_ordered)
        plt.subplot(1,2,2)
        plt.title("Distribution of weights")
        _ = plt.hist(weights_ordered, bins=40)

    return [w for a, f, w in population_with_weights]
```

Lets look at the distribution of the weights with logarithmic fitness function:

[88]:
```
population_with_fitness = [(a, fitness_log(a)) for a in population]
_ = get_weights(population_with_fitness, verbose=True)
```

```
fitness: -3.700, weight: 0.488, agent: 25*100 = 2500.00
fitness: -4.807, weight: 0.371, agent: 25*100-3*5 = 2485.00
fitness: -8.934, weight: 0.075, agent: 3*10*100 = 3000.00
fitness: -10.303, weight: 0.026, agent: 25*10*5 = 1250.00
fitness: -10.563, weight: 0.020, agent: 10*100 = 1000.00
fitness: -10.568, weight: 0.020, agent: 10*100-5 = 995.00
fitness: -10.704, weight: 0.017, agent: 100-5+25*10*3 = 845.00
fitness: -10.973, weight: 0.011, agent: 3+5*100 = 503.00
fitness: -11.034, weight: 0.010, agent: 10/3*25*5 = 416.67
fitness: -11.099, weight: 0.009, agent: 25+100*3-10+5 = 320.00
```

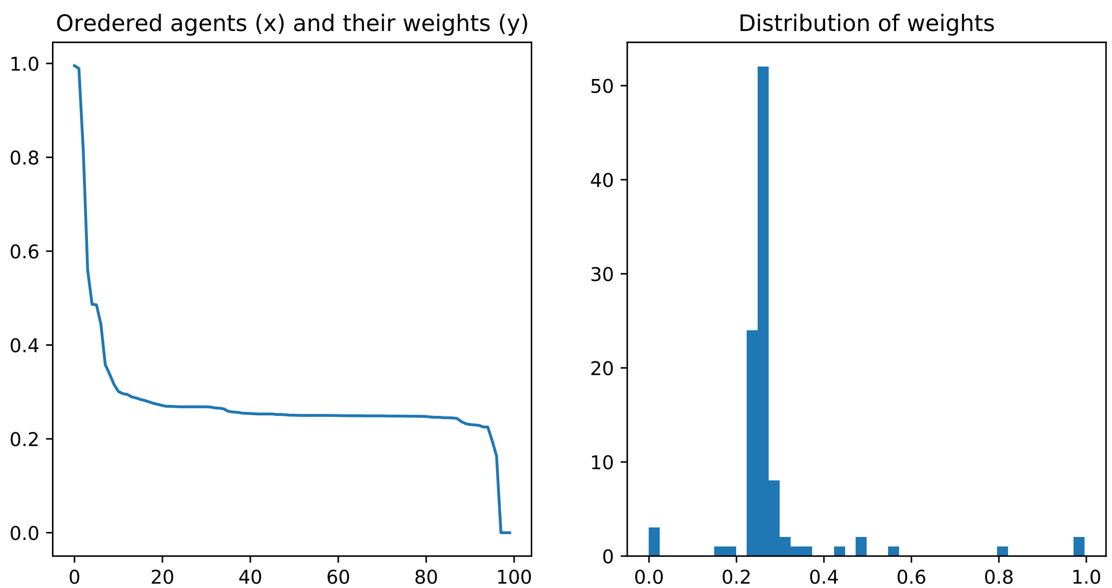


We can see that the best few agents get very large weight and the most agents get weights around 0.2 to 0.5 (depends on the population). There are some very bad agents that gat almost 0 weight.

If we use different fitness function (negative absolute error for example), we get differnet distrubution.

[89]:
```
population_with_fitness = [(a, fitness_abs(a)) for a in population]
_ = get_weights(population_with_fitness, verbose=True)
```

```
fitness: -12.000, weight: 0.995, agent: 25*100 = 2500.00
fitness: -27.000, weight: 0.989, agent: 25*100-3*5 = 2485.00
fitness: -488.000, weight: 0.814, agent: 3*10*100 = 3000.00
fitness: -1262.000, weight: 0.559, agent: 25*10*5 = 1250.00
fitness: -1512.000, weight: 0.487, agent: 10*100 = 1000.00
fitness: -1517.000, weight: 0.486, agent: 10*100-5 = 995.00
fitness: -1667.000, weight: 0.445, agent: 100-5+25*10*3 = 845.00
fitness: -2009.000, weight: 0.358, agent: 3+5*100 = 503.00
fitness: -2095.333, weight: 0.338, agent: 10/3*25*5 = 416.67
fitness: -2192.000, weight: 0.316, agent: 25+100*3-10+5 = 320.00
```



Now we can implement a function that makes a weighted random choice based on before computed wheights:

[90]:
```
def select_weighted(population_with_fitness, n_agents):
    weights = get_weights(population_with_fitness)
    return random.choices(population_with_fitness, weights=weights, k=n_agents)
```

[91]:
```
population_with_fitness = [(a, fitness_log(a)) for a in population]
best_agents_weighted = select_weighted(population_with_fitness, 10)

for a, f in best_agents_weighted:
    print(f"error: {error(a):.3f}, fitness: {f:.3f}, agent:␣
 ↪{agent_to_string(a)}, ")
```

```
error: 27.000, fitness: -4.807, agent: 25*100-3*5 = 2485.00,
error: 2502.000, fitness: -11.289, agent: 10 = 10.00,
error: 488.000, fitness: -8.934, agent: 3*10*100 = 3000.00,
error: 2507.000, fitness: -11.292, agent: 3*10-5*25+100 = 5.00,
error: 1262.000, fitness: -10.303, agent: 25*10*5 = 1250.00,
error: 12.000, fitness: -3.700, agent: 25*100 = 2500.00,
error: 2412.000, fitness: -11.237, agent: 100 = 100.00,
error: 12.000, fitness: -3.700, agent: 25*100 = 2500.00,
error: 2502.000, fitness: -11.289, agent: 10 = 10.00,
error: 2423.667, fitness: -11.244, agent: 100-5/3-10 = 88.33,
```

1.5.3 Best plus random selection

Another approach is to comose the selection of half best agents and other half of randomly chosen agents to avoid elitisem.

[92]:
```
def select_best_plus_random(population_with_fitness, n_agents):
    best_agents = select_best_agents(population_with_fitness, n_agents//2)
    best_agents.extend(random.choices(population_with_fitness, k=(n_agents -␣
↪n_agents//2)))

    return best_agents
```

1.6 Evolution function

Parameters: - `population`: initaial population on which the evolutioin process will be performed - `fitness_function`: a function that takes an agent and returns its fitness value - `selection_function`: a function that takes set of agents with their fitness value and retrns a subset of a few survivers - `crossover_function`: a function that takes two agents and returnes two children - `mutation_function`: a function that takes an agent and performs a random mutation - `mutation_probability`: in mutation process each agent has a probability of mutation - `survival_probability`: percentage of agents that will survive and have offspring - `max_iterations`: self explanatory - `fitness_threshold`: stop when agent with such fitnes value is found - `keep_best`: will parents be part of next population or not - `verbose`: if true, print metrics for each generation - `plot`: if true, plot mean and max fitness in each generation

[108]:
```
def evolve(
    population, fitness_function, selection_function, crossover_function,␣
↪mutation_function,
    mutation_probability=0.4, survival_probability=0.1, max_iterations=100,␣
↪fitness_threshold=0, keep_best = False,
    verbose=False, plot=False
):
    population_size = len(population)

    min_fitness_log = []
    max_fitness_log = []
    mean_fitness_log = []

    string_log = []

    iterations = 0

    for generation in range(max_iterations):
        iterations += 1
        # evaluate population
        population_with_fitness = [(a, fitness_function(a)) for a in population]
        population_fitness = [f for a, f in population_with_fitness]
```

```
        # compute metrics
        mean_fitness = statistics.mean(population_fitness)
        max_fitness = max(population_fitness)
        min_fitness = min(population_fitness)

        # log metrics
        mean_fitness_log.append(mean_fitness)
        max_fitness_log.append(max_fitness)
        min_fitness_log.append(min_fitness)

        # make selection
        best_agents_with_fitness = selection_function(population_with_fitness,␣
↪math.ceil(population_size*survival_probability))
        best_agents = [a for a, _ in best_agents_with_fitness]

        best_agents_with_fitness_sorted = list(sorted(best_agents_with_fitness,␣
↪key=lambda e: e[1], reverse=True))

        # print metrics
        string_log.append(f"Generation {generation}: fitness min: {min_fitness:.
↪2f}, max: {max_fitness:.2f}, mean: {mean_fitness:.2f}; best agent:␣
↪{agent_to_string(best_agents_with_fitness_sorted[0][0])}")

        if max_fitness >= fitness_threshold: break


        if keep_best:
            population = best_agents.copy()
        else:
            population = []

        # crossover
        for i in range(len(population), population_size, 2):
            parent1, parent2 = random.sample(best_agents, 2)
            child1, child2 = crossover_function(parent1, parent2)
            population.append(child1)
            population.append(child2)

        # mutation
        for i, agent in enumerate(population):
            if random.random() < mutation_probability:
                population[i] = mutation_function(agent)

    if verbose:
        if len(string_log) > 10:
            for line in string_log[:5]:
                print(line)
```

```
            print(" ... lines omitted ... ")
            for line in string_log[-5:]:
                print(line)
        else:
            for line in string_log:
                print(line)

    if plot:
        plt.figure(figsize=(10,10))
        plt.plot(max_fitness_log, label="max")
        plt.plot(mean_fitness_log, label="mean")
        #plt.plot(min_fitness_log, label="min")
        plt.legend()

    return (iterations, (max_fitness_log, mean_fitness_log, min_fitness_log))
```

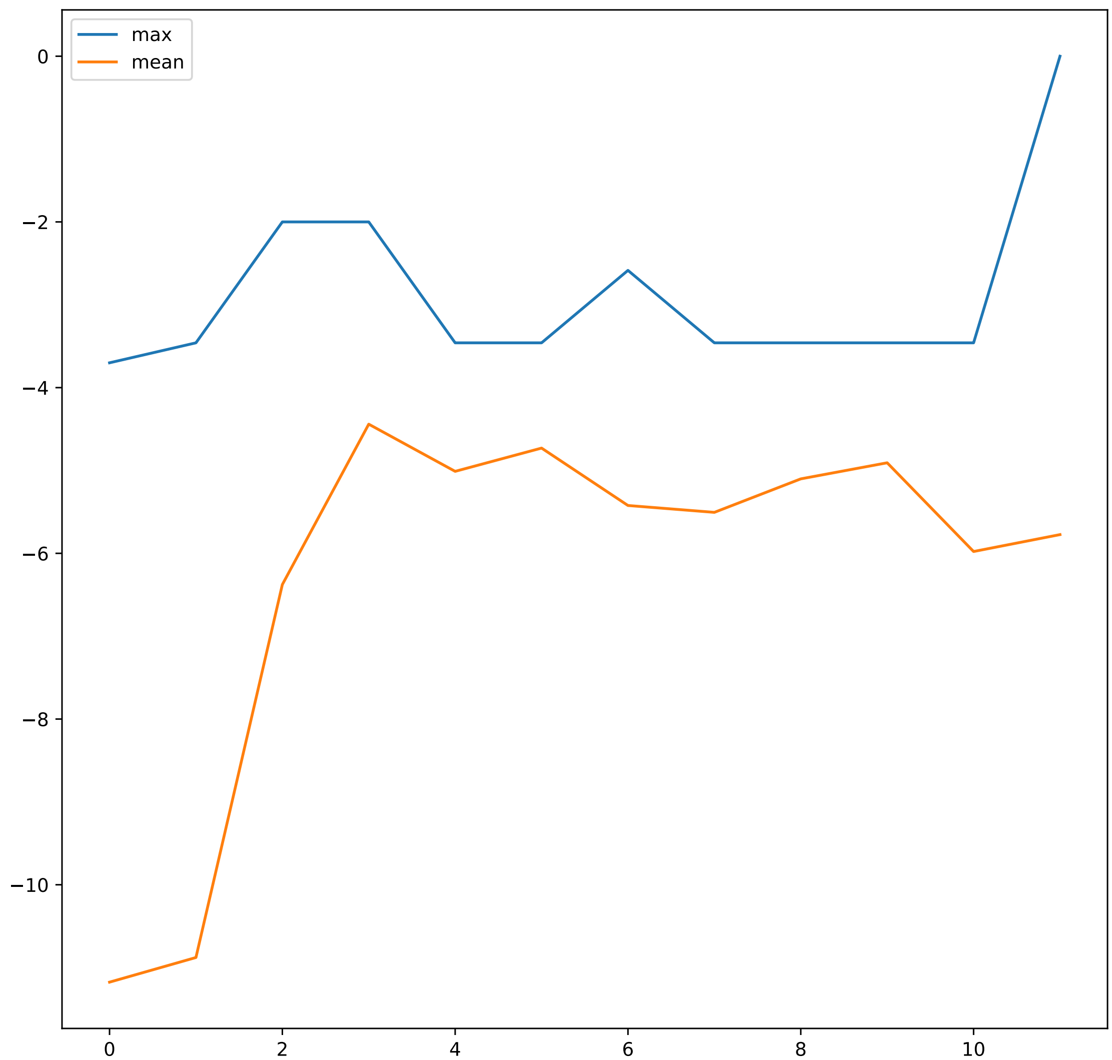
1.7 Evaluation

[113]:
```
population_size = 100
max_iterations = 300

max_number_use = 1
```

[114]:
```
population = [generate_agent() for _ in range(population_size)]
_ = evolve(
    population,
    fitness_log,
    select_weighted,
    crossover,
    mutation,
    fitness_threshold=0,
    mutation_probability=0.2, survival_probability=0.2, max_iterations=300,
    verbose=True, keep_best=False, plot=True
)
```

```
Generation 0: fitness min: -16.90, max: -3.70, mean: -11.18; best agent: 100*25
= 2500.00
Generation 1: fitness min: -12.29, max: -3.46, mean: -10.88; best agent:
100*25-3+5 = 2502.00
Generation 2: fitness min: -11.78, max: -2.00, mean: -6.38; best agent:
100*25+10+5 = 2515.00
Generation 3: fitness min: -12.29, max: -2.00, mean: -4.44; best agent:
100*25-3+5 = 2502.00
Generation 4: fitness min: -14.46, max: -3.46, mean: -5.01; best agent:
100*25-3+5 = 2502.00
 … lines omitted …
```

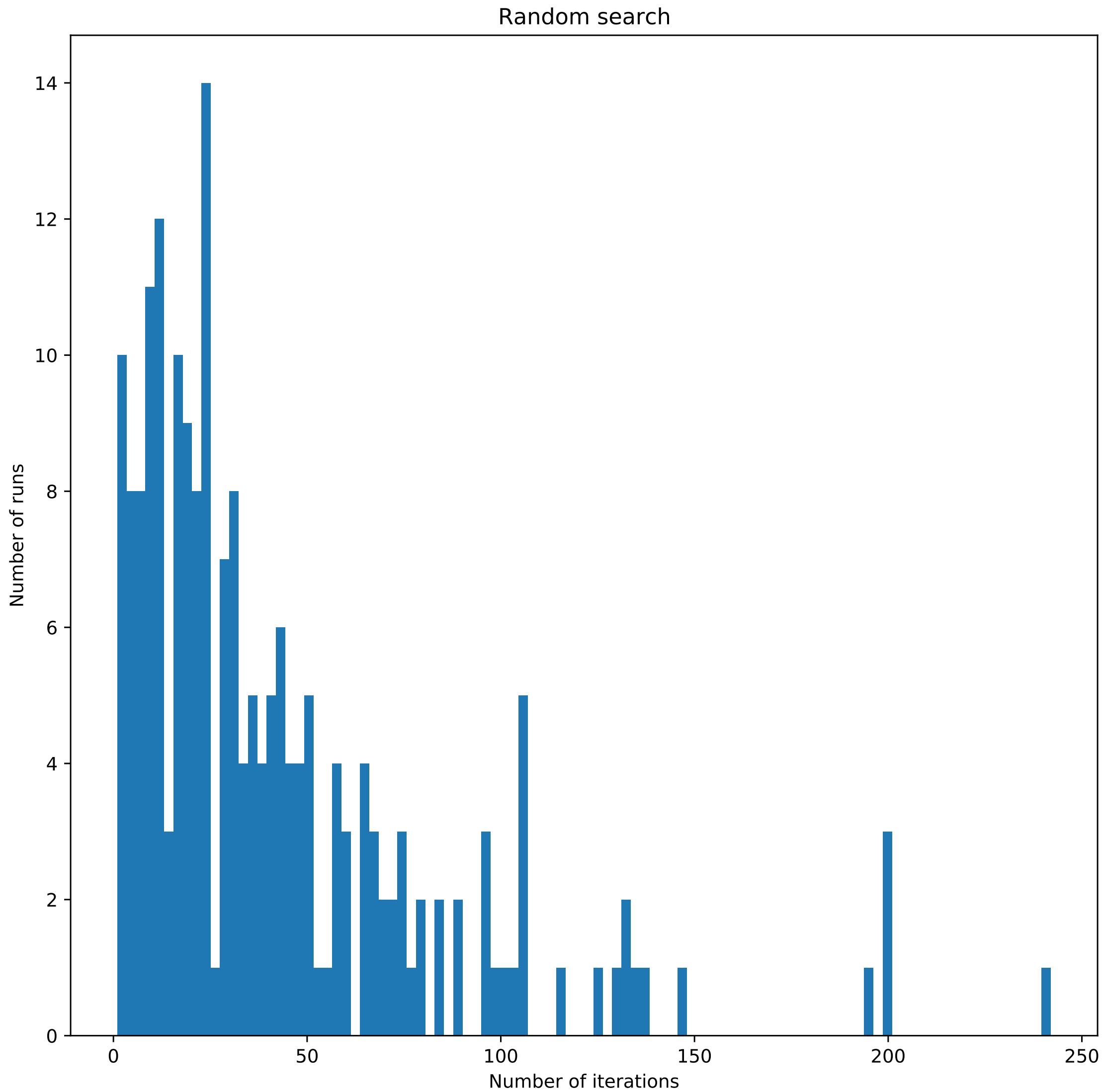
```
Generation 7: fitness min: -14.46, max: -3.46, mean: -5.51; best agent:
100*25-3+5 = 2502.00
Generation 8: fitness min: -13.29, max: -3.46, mean: -5.10; best agent:
100*25-3+5 = 2502.00
Generation 9: fitness min: -13.29, max: -3.46, mean: -4.91; best agent:
100*25-3+5 = 2502.00
Generation 10: fitness min: -14.46, max: -3.46, mean: -5.98; best agent:
100*25-3+5 = 2502.00
Generation 11: fitness min: -13.29, max: -0.00, mean: -5.77; best agent:
100*25-3+10 = 2507.00
```



1.7.1 Comparison with random search

[115]:

```
N = 200
iterations_log = []
for i in range(N):
    iterations = 0
    for i in range(max_iterations):
        iterations += 1
        population = [generate_agent() for _ in range(population_size)]
        population_with_fitness = [(a, fitness_abs(a)) for a in population]
        population_fitness = [f for a, f in population_with_fitness]
        if max(population_fitness) == 0: break

    iterations_log.append(iterations)

plt.figure(figsize=(10,10))
plt.title("Random search")
plt.xlabel('Number of iterations')
plt.ylabel('Number of runs')
_ = plt.hist(iterations_log, bins=100)
```



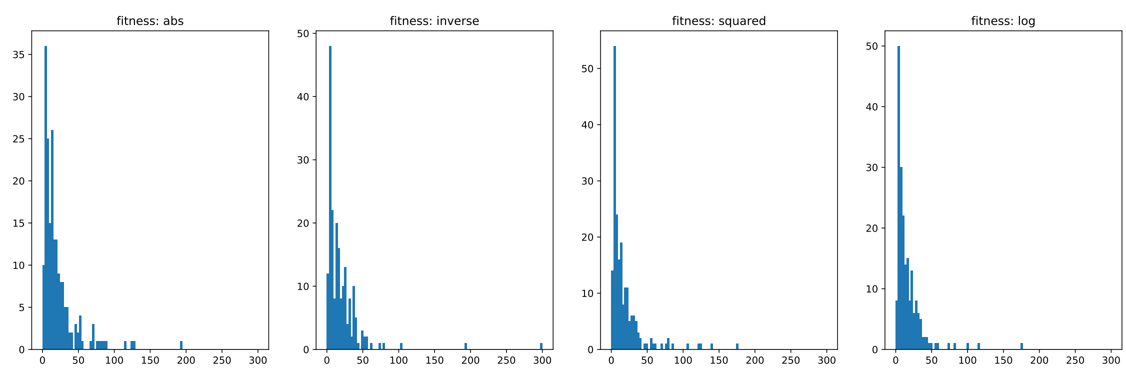
1.7.2 Different fitness functions

Lets see how does the choice of a fitness function effect performance. We run genetic alghorithm with each fitness function a 200 times and plot a histogram of iterations it took to find the optimal agent.

[119]:
```
N = 200
fitness_functions = [(fitness_abs, 0, "abs"), (fitness_inverse, 10000,␣
 ↪"inverse"), (fitness_squared, 0, "squared"),  (fitness_log, 0, "log")]

plt.figure(figsize=(20,6))
```

```
for j, (fitness_function, fitness_threshold, label) in␣
↪enumerate(fitness_functions):
    iterations_log = []

    for i in range(N):
        population = [generate_agent() for _ in range(population_size)]
        iterations, _ = evolve(
            population,
            fitness_function,
            select_best_plus_random,
            crossover,
            mutation,
            mutation_probability=0.2, survival_probability=0.2,␣
↪max_iterations=300, keep_best=False, fitness_threshold=fitness_threshold
        )

        iterations_log.append(iterations)

    print(f"{label} - mean: {statistics.mean(iterations_log)}, min:␣
↪{min(iterations_log)}, max: {max(iterations_log)}, stdev: {statistics.
↪stdev(iterations_log)}")

    plt.subplot(1, 4, j+1)
    plt.title(f"fitness: {label}")
    _ = plt.hist(iterations_log, bins=100, range=(0,300))
```

```
abs - mean: 20.62, min: 1, max: 192, stdev: 24.77933974891098
inverse - mean: 19.41, min: 1, max: 300, stdev: 28.3343621844927
squared - mean: 17.95, min: 1, max: 174, stdev: 24.397205128671036
log - mean: 16.115, min: 1, max: 175, stdev: 19.526602778783666
```

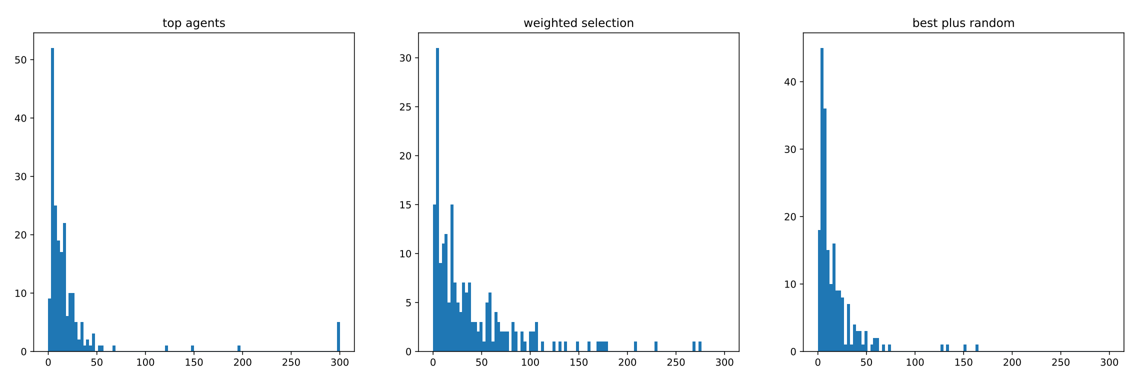


We found out that all fitness functions perform about the same.

1.7.3 Different selection functions

[99]:
```python
N = 200
selection_functions = [(select_best_agents, "top agents"), (select_weighted, "weighted selection"), (select_best_plus_random, "best plus random")]

plt.figure(figsize=(20,6))

for j, (selection_function, label) in enumerate(selection_functions):
    iterations_log = []

    for i in range(N):
        population = [generate_agent() for _ in range(population_size)]
        iterations, _ = evolve(
            population,
            fitness_log,
            selection_function,
            crossover,
            mutation,
            mutation_probability=0.2, survival_probability=0.2, max_iterations=300, keep_best=False, fitness_threshold=0
        )

        iterations_log.append(iterations)

    print(f"{label} - mean: {statistics.mean(iterations_log)}, min: {min(iterations_log)}, max: {max(iterations_log)}, stdev: {statistics.stdev(iterations_log)}")

    plt.subplot(1, 3, j+1)
    plt.title(f"{label}")
    _ = plt.hist(iterations_log, bins=100, range=(0, 300))
```

```
top agents - mean: 22.635, min: 1, max: 300, stdev: 49.25221495924787
weighted selection - mean: 39.615, min: 1, max: 275, stdev: 48.38151122216733
best plus random - mean: 17.1, min: 1, max: 162, stdev: 23.203274764545522
```



We can see from the results that by selecting just the top few agents sometimes the evolution process does not converge. We think that is because we loose the diversity of the population and get stuck in a local optimum.

By also selecting some worse agents the evolution always converges.

We found that the best selection method for this problem is selecting half of the best agents and half of random agents.

1.8 Bonus task: minimize expression length

[100]:
```python
max_number_use = 5
min_numbers_in_expresion = 1
```

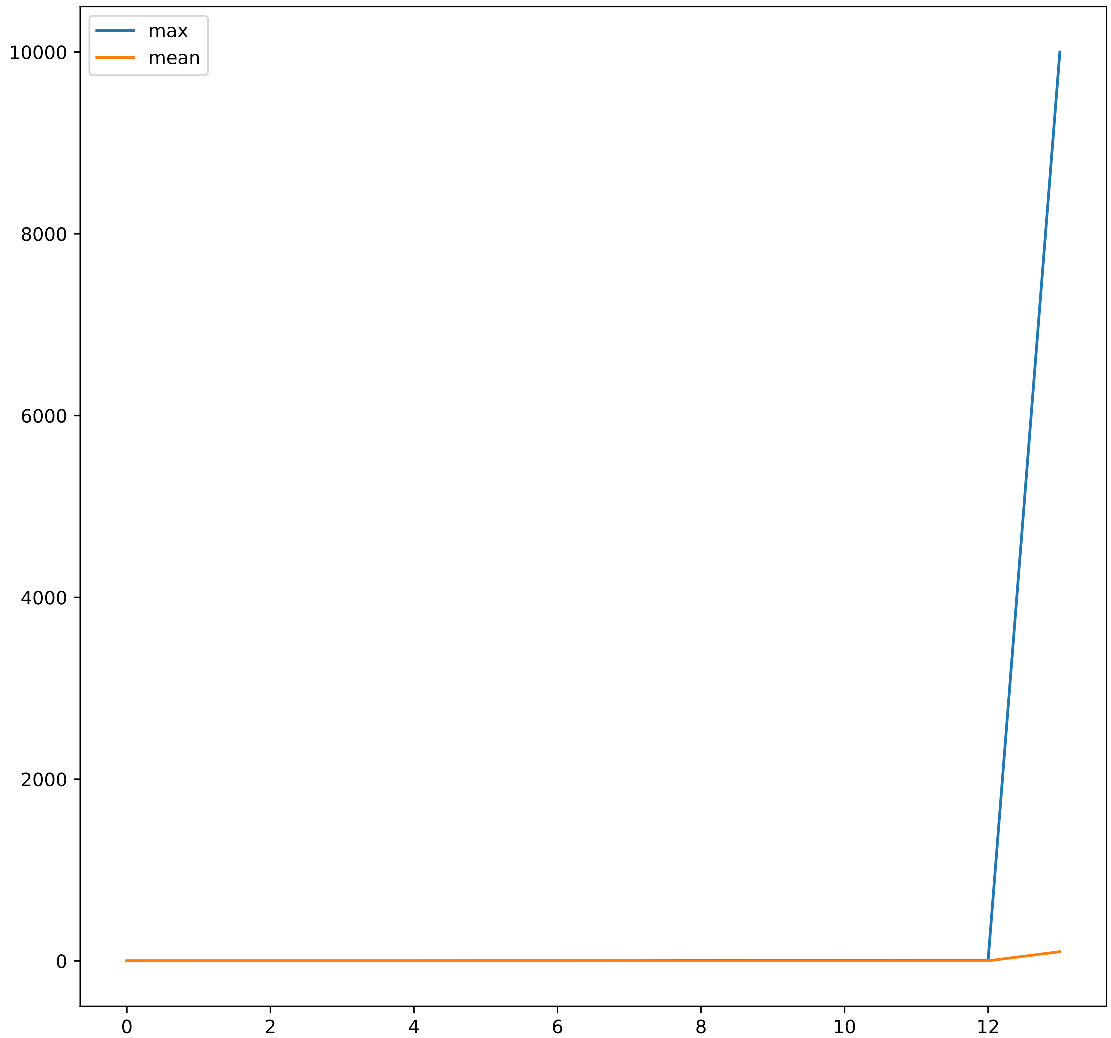
For this task we have to define a new fitness function that also takes into account the expression length.

[101]:
```python
def fitness_length(agent):
    return fitness_inverse(agent) + 3/(len(agent)+5)
```

[120]:
```python
population = [generate_agent() for _ in range(population_size)]
_ = evolve(
    population,
    fitness_length,
    select_best_plus_random,
    crossover,
    mutation,
    fitness_threshold=10000.00,
    mutation_probability=0.5, survival_probability=0.2, max_iterations=300,
    verbose=True, keep_best=True, plot=True
)
```

```
Generation 0: fitness min: 0.30, max: 0.67, mean: 0.38; best agent: 25*100+5*3 =
2515.00
Generation 1: fitness min: 0.30, max: 0.67, mean: 0.45; best agent: 25*100+5*3 =
2515.00
Generation 2: fitness min: 0.30, max: 0.67, mean: 0.48; best agent: 25*100+5*3 =
2515.00
Generation 3: fitness min: 0.32, max: 0.67, mean: 0.48; best agent: 25*100+5*3 =
2515.00
Generation 4: fitness min: 0.30, max: 0.67, mean: 0.46; best agent: 25*100+5*3 =
2515.00
 … lines omitted …
Generation 9: fitness min: 0.30, max: 3.30, mean: 0.82; best agent:
25*100+10+5/3 = 2511.67
Generation 10: fitness min: 0.30, max: 3.30, mean: 1.12; best agent:
```

```
25*100+10+5/3 = 2511.67
Generation 11: fitness min: 0.30, max: 3.30, mean: 1.66; best agent:
25*100+10+5/3 = 2511.67
Generation 12: fitness min: 0.30, max: 3.30, mean: 1.66; best agent:
25*100+10+5/3 = 2511.67
Generation 13: fitness min: 0.30, max: 10000.30, mean: 101.39; best agent:
25*100+10+5-3 = 2512.00
```

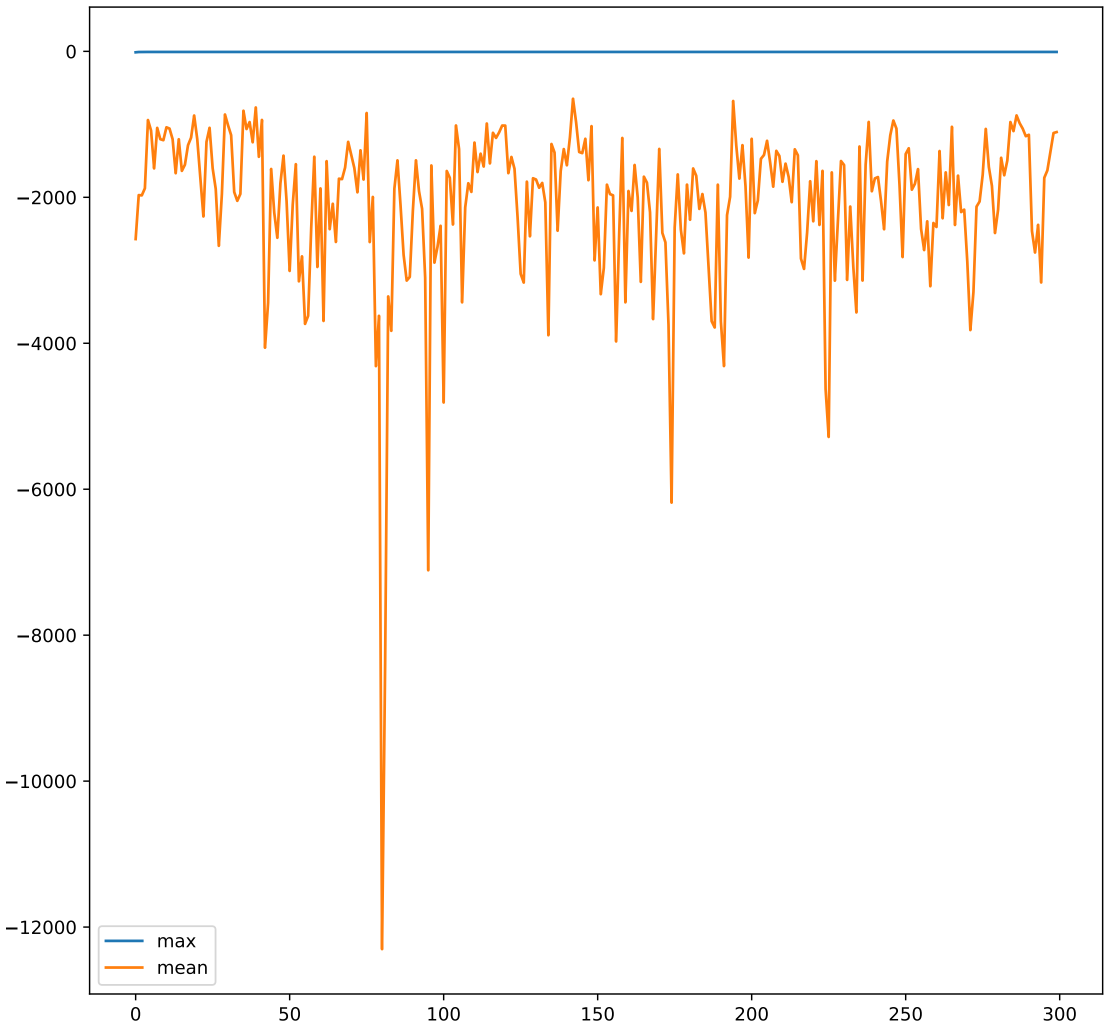


[103]:
```
def fitness_length_2(agent):
    return fitness_abs(agent) - len(agent)*2
```

[121]:
```
population = [generate_agent() for _ in range(population_size)]
_ = evolve(
    population,
    fitness_length_2,
```

```
    select_best_plus_random,
    crossover,
    mutation,
    fitness_threshold=0,
    mutation_probability=0.4, survival_probability=0.2, max_iterations=300,
    verbose=True, keep_best=False, plot=True
)
```

```
Generation 0: fitness min: -22513.00, max: -13.00, mean: -2572.26; best agent:
100*25+10-3 = 2507.00
Generation 1: fitness min: -5013.67, max: -9.00, mean: -1969.73; best agent:
100*25+3+10 = 2513.00
Generation 2: fitness min: -22493.00, max: -8.00, mean: -1973.65; best agent:
100*25+10 = 2510.00
Generation 3: fitness min: -22493.00, max: -8.00, mean: -1876.92; best agent:
100*25+10 = 2510.00
Generation 4: fitness min: -5007.00, max: -8.00, mean: -941.01; best agent:
100*25+10 = 2510.00
 … lines omitted …
Generation 295: fitness min: -22494.00, max: -8.00, mean: -1732.33; best agent:
100*25+10 = 2510.00
Generation 296: fitness min: -22494.00, max: -8.00, mean: -1631.30; best agent:
25*100+10 = 2510.00
Generation 297: fitness min: -10006.00, max: -8.00, mean: -1373.29; best agent:
25*100+10 = 2510.00
Generation 298: fitness min: -5008.00, max: -8.00, mean: -1118.83; best agent:
25*100+10 = 2510.00
Generation 299: fitness min: -10006.00, max: -8.00, mean: -1106.19; best agent:
25*100+10 = 2510.00
```


0
−2000
−4000
−6000
−8000
−10000
−12000
0
50
100
150
200
250
300
max
mean