Решение задачи о рюкзаке генетическим алгоритмом

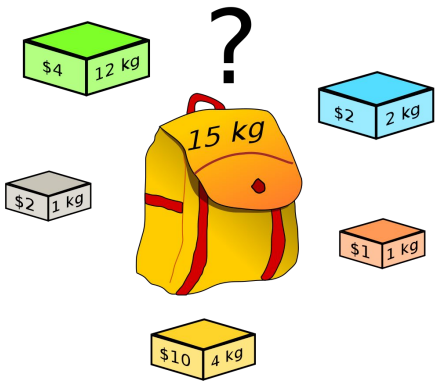
Постановка задачи

Дано N типов предметов, $w = \{w_1, \ldots, w_N\}$ - соответствующий набор положительных весов, $p = \{p_1, \ldots, p_N\}$ - набор положительных стоимостей. W - вместимость рюкзака.

Найти количество предметов каждого типа, чтобы их суммарная стоимость была максимальной, а суммарный вес при этом был меньше или равен W.

$$\sum_{i=1}^{N} p_i * x_i \to max$$

$$\begin{cases} \sum_{i=1}^{N} w_i * x_i \leq W \\ x_i \in \mathbb{N} \cup 0, \forall i \in 1..N \end{cases}$$



Для реализации использовались два класса:
Item - представляет собой один возможный элемент/предмет, который может попасть в рюкзак

Individual - особь в популяции - вектор, содержащий количество предметов каждого типа
Функция приспособленности особи (self.fitness()): $F = \sum p_i * x_i$

Условие допустимости решения: $\sum w_i * x_i <= W$, где W - максимальный вес рюкзака
Если решение недопустимо, F = 0

```python
class Item:
    def __init__(self, name, weight, value):
        self.name = name
        self.weight = weight
        self.value = value
```

```python
class Individual:
    def __init__(self, counts):
        self.counts = counts

    def __str__(self):
        return repr(self.counts)

    def __hash__(self):
        return hash(str(self.counts))

    def __eq__(self, other):
        return self.counts == other.counts

    def fitness(self) -> float:
        total_value = sum([
            count * item.value
            for item, count in zip(items, self.counts)
        ])

        total_weight = sum([
            count * item.weight
            for item, count in zip(items, self.counts)
        ])

        if total_weight <= MAX_KNAPSACK_WEIGHT:
            return total_value

        return 0
```

Глобальные параметры

```
MAX_KNAPSACK_WEIGHT = 503
DIFF_PRECISION = 0.05
STOP_EPOCH_NUM = 5

MUTATION_RATE = 0.01
REPRODUCTION_RATE = 0.2
POPULATION_COUNT = 25
NUM_OF_POPULATIONS = 50
MIGRATION_PERIOD = 5
BEST_IN_MIGRATION = 3
IMMIGRANTS_NUM = 1

EPOCHS = 500
```

MAX_KNAPSACK_WEIGHT - максимальный вес рюкзака
DIFF_PRECISION - точность для критерия остановки
STOP_EPOCH_NUM - количество эпох, в течение которых метрика не должна меняться для выхода из цикла обучений

MUTATION_RATE - вероятность мутации
REPRODUCTION_RATE - доля родителей, которые попадают в следующее поколение
POPULATION_COUNT - количество особей в популяции
NUM_OF_POPULATION - количество популяций
MIGRATION_PERIOD - период изоляции, после которого происходит миграция
BEST_IN_MIGRATION - количество лучших особей, которые мы берем из популяции для миграции
IMMIGRANTS_NUM - количество особей, которые приходят во время миграции из других популяций

EPOCHS - максимальное количество эпох

Генерация начальной популяции

```python
def generate_initial_population(population_count=POPULATION_COUNT):
    population = set()
    while len(population) != population_count:
        idxs = list(range(len(items)))
        random.shuffle(idxs)
        counts = [0 for i in range(len(items))]
        W = MAX_KNAPSACK_WEIGHT
        for idx in idxs[:-1]:
            i = items[idx]
            count = random.randint(0, W // i.weight)
            counts[idx] = count
            W -= count * i.weight
        counts[idxs[-1]] = W // items[idxs[-1]].weight
        population.add(Individual(counts))
    return list(population)
```

Для каждой особи в популяции:

1. Перемешиваются индексы предметов (пусть мы получили массив индексов [3, 1, 0, 2, 4], тогда первым будет заполняться значение для предмета с индексом 3)

2. Случайным образом генерируется количество предметов x_i i типа в диапазоне от 0 до [W / w_i]

2. $W = W - x_i * w_i$

...

3. На N-ом шаге количество предметов x_N N типа = [W/w_n]

Алгоритм формирования нового поколения

```python
def next_generation(population):
    next_gen = []
    while len(next_gen) < len(population):
        children = []

        #selection
        parents = selection(population)

        # reproduction
        if random.random() < REPRODUCTION_RATE:
            children = parents
        else:
            # crossover
            children = crossover(parents)

            # mutation
            children = mutate(children)

        if children[0].fitness():
            next_gen.append(children[0])
        if children[1].fitness():
            next_gen.append(children[1])

    next_gen.extend(population)

    n_sorted = sorted(next_gen, key=lambda x: x.fitness(), reverse=True)

    new_population = []
    population_len = len(n_sorted)
    rank = 0
    while len(new_population) != POPULATION_COUNT:
        if random.random() < (population_len - rank) / population_len:
            new_population.append(n_sorted[rank])
        rank = (rank + 1) % population_len
    return sorted(new_population, key=lambda i: i.fitness(), reverse=True)
```

В цикле пока не будет заполнена популяция размера *POPULATION_COUNT*:

- выбирается пара родителей;
- с вероятностью *REPRODUCTION_RATE* они попадают в новую популяцию, иначе они проходят кроссовер;
- двое детей после кроссовера с вероятностью *MUTATION_RATE* проходят мутацию;
- если ребенок проходит проверку допустимости, он добавляется в промежуточную популяцию (вес особи не должен превышать максимальный вес рюкзака).

Затем из промежуточной популяции особи с помощью ранкового отбора (вероятность особи из промежуточной популяции попасть в новую обратно пропорциональна ее порядковому номеру в отсортированной по убыванию приспособленности промежуточной популяции) отбираются в новую популяцию.

Выбор родителей - турнирный отбор

```
def selection(population):
    parents = []

    random.shuffle(population)

    if population[0].fitness() > population[1].fitness():
        parents.append(population[0])
    else:
        parents.append(population[1])

    if population[2].fitness() > population[3].fitness():
        parents.append(population[2])
    else:
        parents.append(population[3])

    return parents
```

Случайно отбираем 2 пары потенциальных родителей (перемешиваем выборку и берем первых 4-х), в каждой паре отбираем лучшую особь.

Кроссовер - одноточечный

```
def crossover(parents):
    N = len(items)

    crosscover_point = random.randint(1, N-2)
    child1 = Individual(parents[0].counts[:crosscover_point] + parents[1].counts[crosscover_point:])
    child2 = Individual(parents[1].counts[:crosscover_point] + parents[0].counts[crosscover_point:])

    return [child1, child2]
```

Случайно выбираем точку разрыва и образуем 2-х потомков путем конкатенации левой и правой частей родителей. (Валидность детей проверяется в *next_generation*, если ребенок не проходит условие допустимости, то он не добавляется в новую популяцию).

Пример кроссовера:

```
parents selection:
[91, 0, 77, 0, 13, 0, 0, 0, 0, 0]
[119, 1, 0, 0, 15, 0, 6, 0, 0, 15]
crossover point: 4
children:
[91, 0, 77, 0, 15, 0, 6, 0, 0, 15]
[119, 1, 0, 0, 13, 0, 0, 0, 0, 0]
```

Мутация

```
def mutate(individuals):
    for individual in individuals:
        if random.random() < MUTATION_RATE:
            for _ in range(random.randint(len(individual.counts) // 2, len(individual.counts))):
                w = MAX_KNAPSACK_WEIGHT
                gen = random.randint(0, len(individual.counts)-1)
                for i in range(len(individual.counts)):
                    if i == gen:
                        continue
                    w -= individual.counts[i] * items[i].weight
                individual.counts[gen] = random.randint(0, w // items[gen].weight)
    return individuals
```

Случайно выбирается от [N/2] до N генов, после чего их значения заменяются на случайные в диапазоне от 0 до x_k, где x_k - максимальное допустимое число предметов типа k для данной особи.

Пример мутации

```
mutate: [4, 61, 3, 10, 0, 1, 0, 16, 1, 0]
change gen 1
mutate: after: [4, 35, 3, 10, 0, 1, 0, 16, 1, 0]
change gen 4
mutate: after: [4, 35, 3, 10, 8, 1, 0, 16, 1, 0]
change gen 3
mutate: after: [4, 35, 3, 19, 8, 1, 0, 16, 1, 0]
change gen 4
mutate: after: [4, 35, 3, 19, 7, 1, 0, 16, 1, 0]
change gen 8
mutate: after: [4, 35, 3, 19, 7, 1, 0, 16, 14, 0]
change gen 5
mutate: after: [4, 35, 3, 19, 7, 2, 0, 16, 14, 0]
change gen 0
mutate: after: [9, 35, 3, 19, 7, 2, 0, 16, 14, 0]
```

Критерий остановки

```
def solve_knapsack(population):
    avg_fitness = []
    for _ in range(MIGRATION_PERIOD):
        population = next_generation(population)
        avg_fitness.append(average_fitness(population))
        # среднее значение функции приспособленности за последние STOP_EPOCH_NUM эпох
        # не отличается от последнего значения: average fitness[last 5 epoch] / last avg fitness - 1
        if len(avg_fitness) > STOP_EPOCH_NUM and \
            abs((sum(avg_fitness[-STOP_EPOCH_NUM:]) / STOP_EPOCH_NUM) / avg_fitness[-1] - 1) < DIFF_PRECISION:
            break
    return sorted(population, key=lambda i: i.fitness(), reverse=True)
```

1. Пока число шагов ГА не будет равно *EPOCHS*

2. Пока не будет достигнута стабилизация функции припособленности с *DIFF_PRESICION* для последних *STOP_EPOCH_NUM*

Распараллеливание

```python
def parallel_knapsack():
    populations = []
    for _ in range(NUM_OF_POPULATIONS):
        populations.append(generate_initial_population())
        print_generation(populations[-1])


    pool = mp.Pool(processes=len(populations))

    prev_fitness = 0
    for _ in range(EPOCHS // MIGRATION_PERIOD):
        # print(f"migration period {i}")
        populations = pool.map(solve_knapsack, populations)
        migrations(populations)

        best = populations[0][0]
        for population in populations[1:]:
            if population[0].fitness() > best.fitness():
                best = population[0]
        if best.fitness() == prev_fitness:
            break
        prev_fitness = best.fitness()

    # for i in range(len(populations)):
    #     print(f"популяция {i}")
    #     print_generation(populations[i])

    return best
```

Генерируются начальные популяции в количестве *NUM_OF_POPULATIONS*.

Используется объект *Pool* из модуля *multiprocessing*, который запускает алгоритм для каждой популяции в отдельном процессе.

Через *MIGRATION_PERIOD* шагов происходит миграция.

В качестве итогового результата берется лучшая особь среди всех популяций

Миграции

```
def migrations(populations):
    bestI = []
    for population in populations:
        bestI.extend(population[:BEST_IN_MIGRATION])

    for population in populations:
        print_generation(population)
        for i in range(IMMIGRANTS_NUM):
            best = random.choice(bestI)
            if population[-i-1].fitness() < best.fitness():
                population[-i-1] = best
        print_generation(population)
```

Из каждой популяции отбирается *BEST_IN_MIGRATION* лучших особей (отбирается множество лучших).

В каждой популяции *IMMIGRANTS_NUM* худших особей заменяется на *IMMIGRANTS_NUM* случайных особей из множества лучших.

Пример миграции:

```
best individuals
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0]
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0]
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0]
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0]
[4, 11, 16, 0, 49, 0, 0, 0, 2, 0]
[4, 11, 16, 0, 49, 0, 0, 0, 2, 0]
[4, 11, 16, 0, 49, 0, 0, 0, 2, 0]
[4, 11, 16, 0, 49, 0, 0, 0, 2, 0]
migration: init population:
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 1, 0] fitness 1166 total_weight 496
[21, 0, 1, 2, 55, 0, 0, 2, 1, 0] fitness 1166 total_weight 496
[21, 0, 1, 2, 55, 0, 0, 2, 1, 0] fitness 1166 total_weight 496
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[0, 150, 0, 7, 0, 0, 0, 0, 2, 0] fitness 1101 total_weight 502
[0, 150, 0, 7, 0, 0, 0, 0, 2, 0] fitness 1101 total_weight 502
[0, 150, 0, 7, 0, 0, 0, 0, 2, 0] fitness 1101 total_weight 502
[0, 150, 0, 7, 0, 0, 0, 0, 2, 0] fitness 1101 total_weight 502
[0, 150, 0, 7, 0, 0, 0, 0, 2, 0] fitness 1101 total_weight 502
```

```
migration: after migration:
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[21, 0, 1, 2, 55, 0, 0, 2, 1, 0] fitness 1166 total_weight 496
[21, 0, 1, 2, 55, 0, 0, 2, 1, 0] fitness 1166 total_weight 496
[21, 0, 1, 2, 55, 0, 0, 2, 1, 0] fitness 1166 total_weight 496
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[21, 0, 1, 2, 55, 0, 0, 0, 2, 0] fitness 1156 total_weight 487
[0, 150, 0, 7, 0, 0, 0, 0, 2, 0] fitness 1101 total_weight 502
[0, 150, 0, 7, 0, 0, 0, 0, 2, 0] fitness 1101 total_weight 502
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
[4, 11, 16, 0, 49, 0, 0, 0, 2, 0] fitness 1221 total_weight 503
[21, 0, 1, 2, 55, 0, 0, 2, 2, 0] fitness 1174 total_weight 501
```

Пример работы:

```
best_fitness = 1256
decision = [0, 1, 1, 0, 62, 0, 0, 0, 0, 0]

error = 0
counter = 0
for i in range(100):
    solution = parallel_knapsack()
    if solution.counts == decision:
        counter += 1
    error += abs(solution.fitness() - best_fitness)
    print(solution.counts, solution.fitness())

counter, error / 100
```

```
# best result 1256
items = [
    # name, weight, value
    Item("A", 1, 1),      # 0
    Item("B", 3, 7),      # 1
    Item("C", 4, 9),      # 1
    Item("D", 6, 5),      # 0
    Item("E", 8, 20),     # 62
    Item("F", 10, 13),    # 0
    Item("G", 21, 27),    # 0
    Item("H", 7, 9),      # 0
    Item("I", 5, 8),      # 0
    Item("K", 9, 13),     # 0
]
```

Результат:
counter = 17 (в 17% случаев полное совпадение решения)
error / 100 = 3.5 (средняя ошибка)

Часть решений:

```
[4, 1, 0, 0, 62, 0, 0, 0, 0, 0] 1251
[1, 0, 0, 0, 62, 0, 0, 0, 1, 0] 1249
[0, 1, 3, 0, 61, 0, 0, 0, 0, 0] 1254
[1, 2, 0, 0, 62, 0, 0, 0, 0, 0] 1255
[3, 0, 1, 0, 62, 0, 0, 0, 0, 0] 1252
[1, 2, 0, 0, 62, 0, 0, 0, 0, 0] 1255
[1, 2, 0, 0, 62, 0, 0, 0, 0, 0] 1255
[3, 0, 3, 0, 61, 0, 0, 0, 0, 0] 1250
[0, 6, 0, 0, 60, 0, 0, 0, 1, 0] 1250
[1, 2, 0, 0, 62, 0, 0, 0, 0, 0] 1255
[0, 1, 1, 0, 62, 0, 0, 0, 0, 0] 1256
[0, 1, 1, 0, 62, 0, 0, 0, 0, 0] 1256
[0, 0, 0, 0, 62, 0, 0, 1, 0, 0] 1249
[1, 2, 0, 0, 62, 0, 0, 0, 0, 0] 1255
[2, 0, 2, 0, 61, 0, 0, 0, 1, 0] 1248
[0, 1, 1, 0, 62, 0, 0, 0, 0, 0] 1256
[0, 1, 1, 0, 62, 0, 0, 0, 0, 0] 1256
[0, 5, 0, 0, 61, 0, 0, 0, 0, 0] 1255
[3, 0, 1, 0, 62, 0, 0, 0, 0, 0] 1252
[3, 4, 0, 0, 61, 0, 0, 0, 0, 0] 1251
[1, 3, 2, 0, 60, 0, 0, 0, 1, 0] 1248
[1, 2, 0, 0, 62, 0, 0, 0, 0, 0] 1255
[0, 1, 1, 0, 62, 0, 0, 0, 0, 0] 1256
[1, 2, 0, 0, 62, 0, 0, 0, 0, 0] 1255
[1, 2, 0, 0, 62, 0, 0, 0, 0, 0] 1255
[3, 0, 1, 0, 62, 0, 0, 0, 0, 0] 1252
[0, 1, 1, 0, 62, 0, 0, 0, 0, 0] 1256
```

Подбор гиперпараметров ГА:

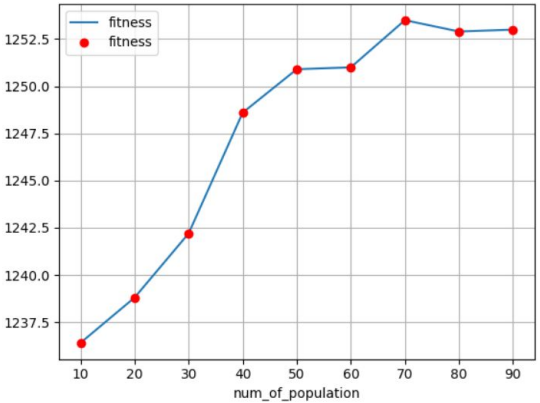
```
params = {
    "mutation_rate": [0.005, 0.02, 4],
    "reproduction_rate": [0.1, 0.4, 4],
    "population_count": [20, 30, 2],
    "num_of_populations": [10, 10, 1],
    "migration_period": [5, EPOCHS//10, EPOCHS//10//11],
    "best_in_migration": [3, 5, 3],
    "immigrants_num": [1, 3, 3],
}
```

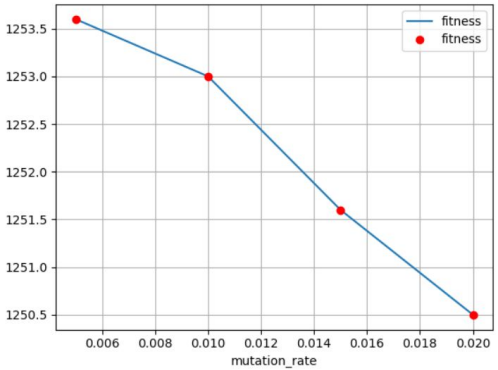
Наилучшие гиперпараметры ГА были определены путем полного перебора всех сочетаний гиперпараметров из заданного диапазона.

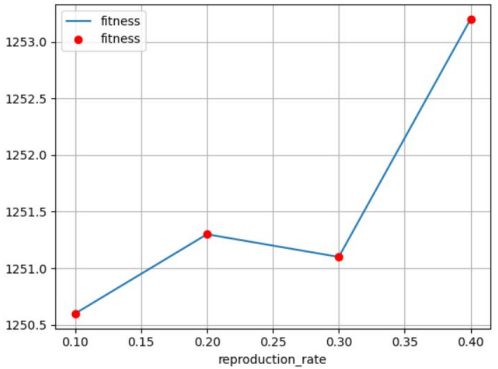
Результат подбора гиперпараметров ГА:

	mutation_rate	reproduction_rate	population_count	migration_period	best_in_migration	immigrants_num	fitness	time
0	0.010	0.1	30.0	5.0	4.0	3.0	1249.6	0.075718
1	0.010	0.1	20.0	20.0	4.0	2.0	1249.0	0.069363
2	0.015	0.2	30.0	5.0	5.0	1.0	1247.2	0.071800
3	0.005	0.4	30.0	35.0	4.0	3.0	1246.3	0.077156
4	0.010	0.2	20.0	50.0	3.0	1.0	1246.2	0.067084

Влияние гиперпараметров на сходимость ГА



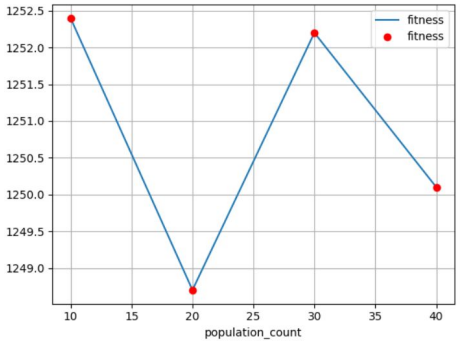


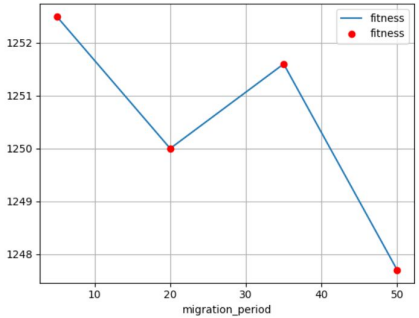


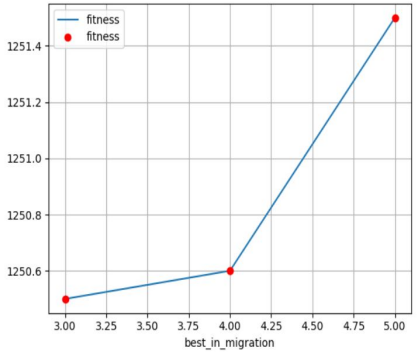
При фиксированных остальных гиперпараметрах:

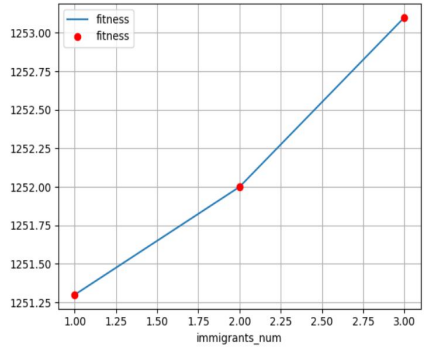
- Увеличение *NUM_OF_POPULATION* в среднем улучшает сходимость.
- Увеличение *MUTATION_RATE* в среднем ухудшает сходимость.
- *Увеличение REPRODUCTION_RATE в среднем может как улучшить, так и ухудшить сходимость.*

Влияние гиперпараметров на сходимость ГА









При фиксированных остальных гиперпараметрах:

- Увеличение *POPULATION_COUNT в* среднем может как улучшить, так и ухудшить сходимость.
- Увеличение *MIGRATION_PERIOD* в среднем может как улучшить, так и ухудшить сходимость.
- *Увеличение BEST_IN_MIGRATION* в среднем улучшает сходимость.
- *Увеличение IMMIGRANTS_NUM* в среднем улучшает сходимость.

Спасибо за внимание