**МИНОБРНАУКИ РОССИИ**

федеральное государственное бюджетное образовательное учреждение высшего образования

**«Национальный исследовательский университет «МЭИ»**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ЛАБОРАТОРНАЯ РАБОТА №4

По курсу: «Методы решения задач оптимизации»

Тема: «Генетический алгоритм»

|  |  |
| --- | --- |
| Выполнил: | Волков М.Л. |
| Вариант: | 1 |
| Группа: | Э-13м-23 |
| Проверил: | Нухулов С.М. |

Москва, 2024 г.

**Предварительный отчет**

**Цель:** получение практических навыков работы с генетическим алгоритмом.

**Задание:**

1. Написать генетический алгоритм на языке Python для поставленной задачи;
2. Решить поставленную задачу с использованием написанного алгоритма;

**Формулировка задачи:**

Персональный энергоблок (ПЭБ) имеет в своем составе аккумуляторную батарею емкости capacity [Вт∙ч] и уровнем заряда initCharge Вт∙ч. Цена за электроэнергию в течении дня изменяется согласно почасовому графику priceSchedule, нагрузка потребителя, подключенного к ПЭБ, изменяется согласно почасовому графику loadSchedule, также подключен потребитель с постоянной нагрузкой constantLoad [Вт∙ч]. ПЭБ способен каждый час либо заряжать свою аккумуляторную батарею (от 1 до 4 кВт∙ч), покупая электроэнергию из сети, либо разряжать (от 1 до 4 [кВт∙ч]) – продавая излишки электроэнергии в сеть, либо не производить торговых операций вовсе. Необходимо спланировать график торговых операций на следующий день, имея перечисленную информацию, на каждый час так, чтобы суммарное вознаграждение к концу дня было максимальным, а оставшийся заряд аккумуляторной батареи был выше значения targetCharge [Вт∙ч].

**Исходные данные:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Вар** | **capacity** | **initCharge** | **priceSchedule** | **loadSchedule** | **constantLoad** | **targetCharge** |
| 1 | 16000 | 6000 | 1.5, 1.5, 1.5, 1.5,  1.5, 1.5, 2, 3, 5,  5, 5, 4.5, 3, 3, 3,  3, 4.5, 5, 7, 9,  11, 12, 8, 4 | 480, 320, 320, 360, 360,  360, 420, 920, 1200, 720,  680, 720, 800, 820, 960,  1200, 1380, 1380, 1520,  1800, 1920, 1920, 1640, 1020 | 400 | 4800 |

**Теоретическая справка:**

Для реализации генетического алгоритма требуется использовать фреймворк Deap.

DEAP (сокращение от Distributed Evolutionary Algorithms in Python – распределенные эволюционные алгоритмы на Python).

Для реализации потребуется три следующих инструмента:

from deap import base

from deap import creator

from deap import tools

**creator** – позволяет создавать новые объекты (классы) в программе. Для этого необходимо использовать функцию **creator.create()**.

**create**(“Название класса”, <базовый класс>, [атрибуты нового класса])

После создания класса таким образом, к нему можно будет обращаться через **creator().название\_класса**

**base** – модуль пакета Deap, который предоставляет базовые инструменты для создания генетического алгоритма.

С помощью модуля **base**, можно зарегистрировать экземпляр класса **Toolbox**, который помогает регистрировать функции для работы генетического алгоритма. Делается это следующим образом, сначала определяется сам экземпляр, обычно называют его toolbox:

**toolbox = base.Toolbox()**

далее уже можно создавать функции для работы генетического алгоритма, делается это так:

**toolbox.register**(“новое имя функции”, функция, на основе которой создается псевдоним**).** По сути, берется стандартная функция, которая уже либо прописана нами, либо есть в пакете Deap и ей присваивается псевдоним, либо же создавать самим.

Помимо этого, можно определять функции для генерации списков, здесь поможет модуль tools. Используя **tools.initRepeat** можно создавать списки значений.

Подробнее о самом **initRepeat**:

**initRepeat**(<контейнер для хранения значения>, <функция, которая создает эти значения>, <число значений в контейнере>). Данные параметры прописываются после **tools.initRepeat.**

Оперируя псевдонимами, можно строить генетический алгоритм.

Алгоритм будет осуществляться по следующей схеме:



**Отчет**

Сравнение результатов работы генетического алгоритма и результатов динамического программирования:

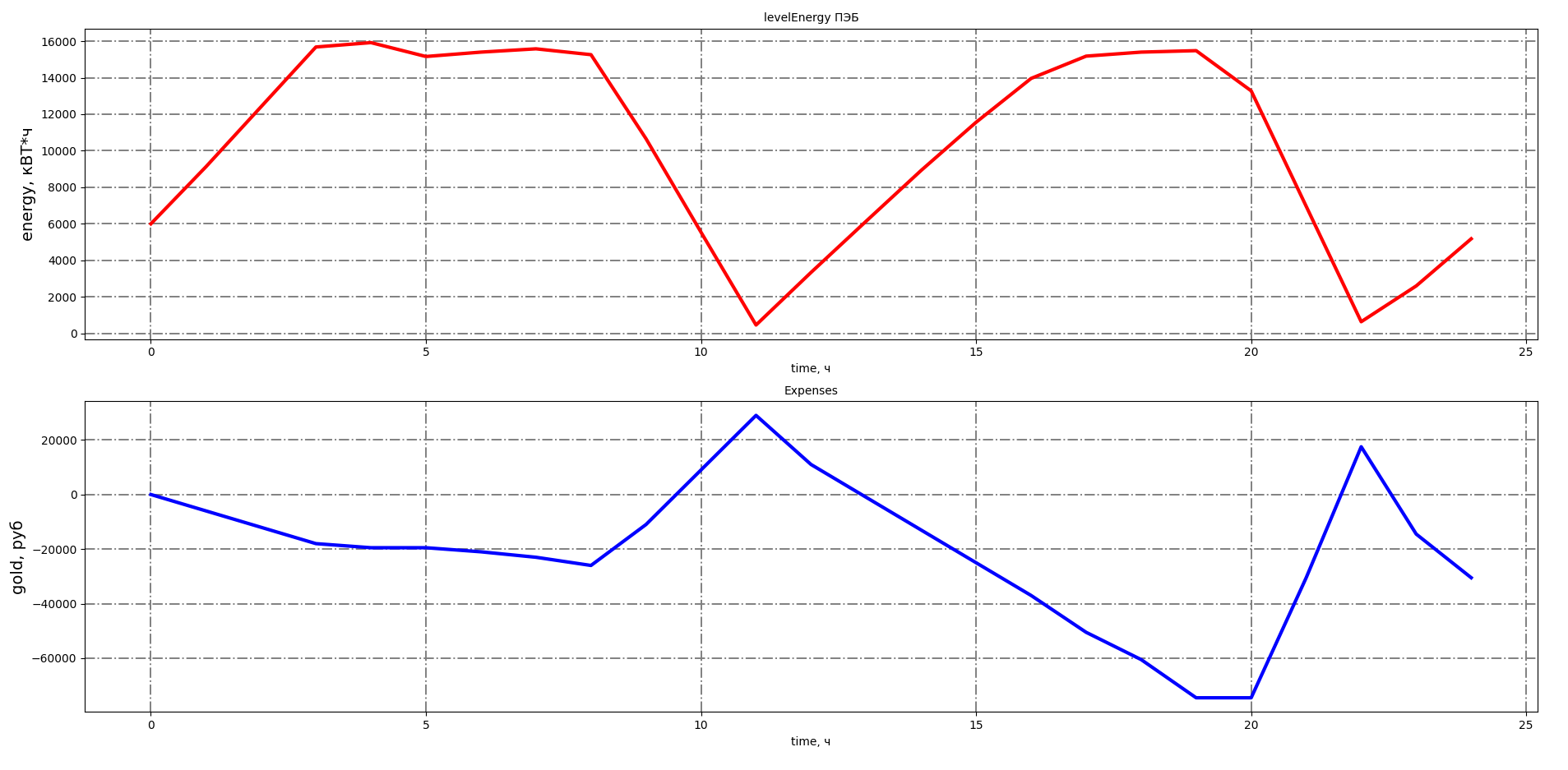


Рис. 1 – Результаты работы динамического программирования

**Конечный доход:** -30500

**Конечный заряд:** 5180

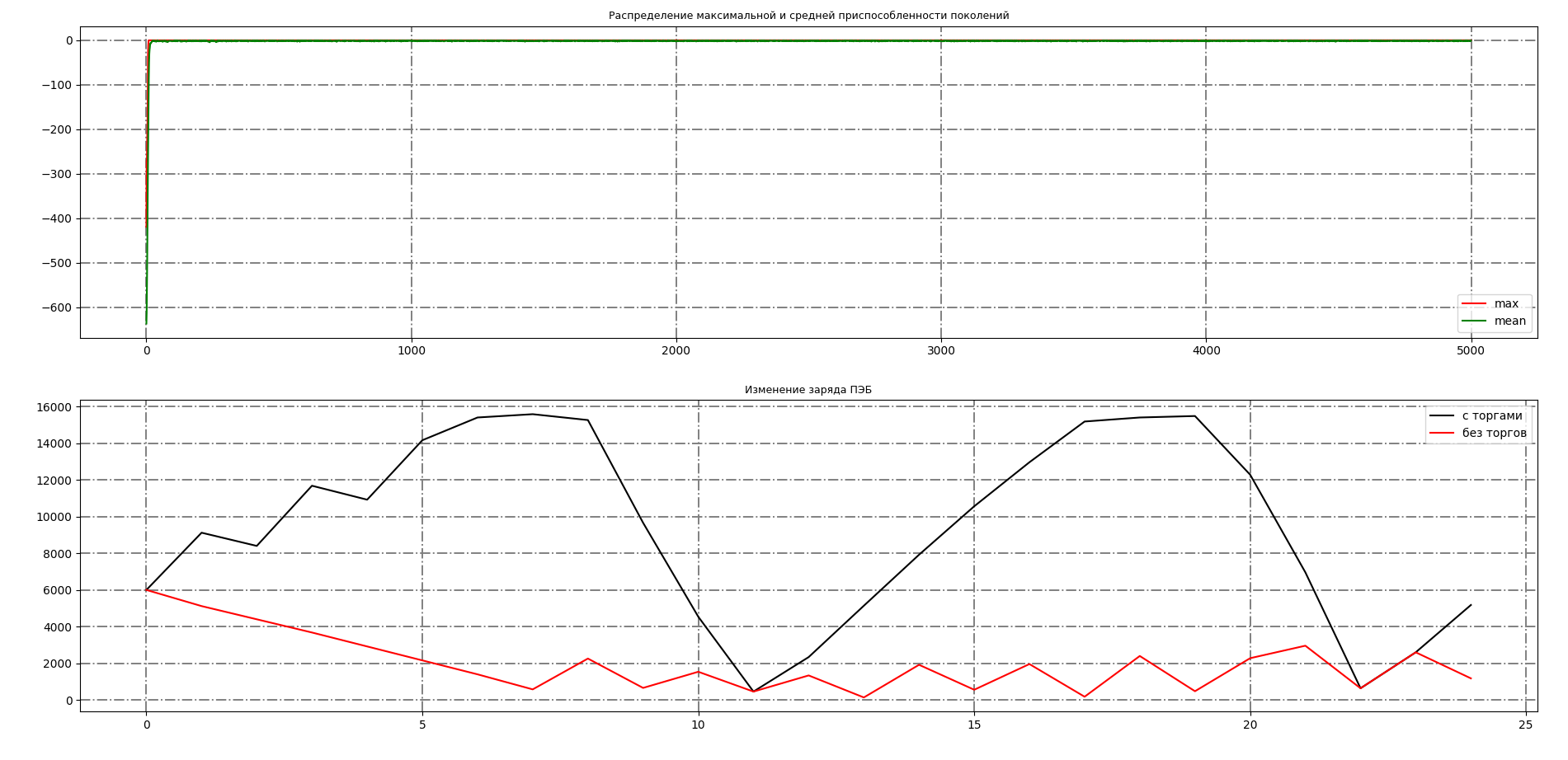


Рис. 2 – Результаты работы генетического алгоритма при 5000 поколений

**Конечный доход:** -32500

**Конечный заряд:** 5180

Результаты работы генетического алгоритма очень близки к результатам работы алгоритма из 3 лабораторной работы. При увеличении числа поколений, расчет генетического алгоритма будет точнее и станет равным результатам работы динамического программирования.

Результаты работы генетического алгоритма при 100000 поколений:

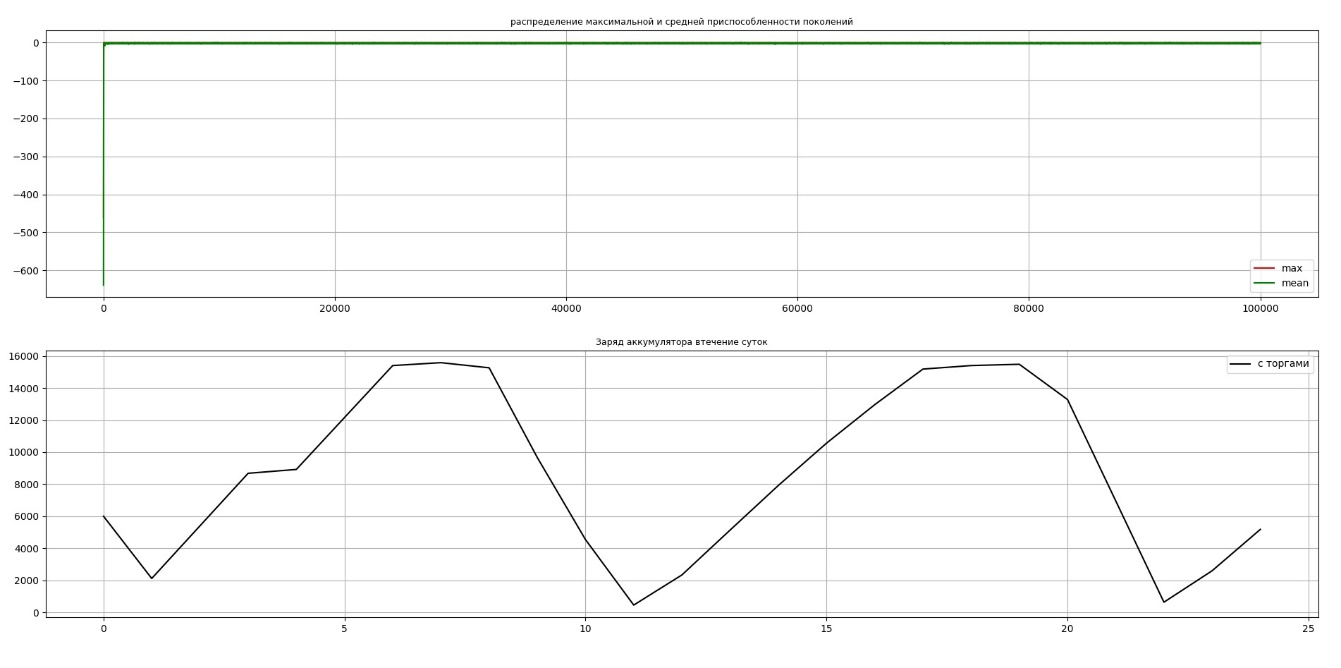


Рис. 3 – Результаты работы генетического алгоритма при 100000 поколений

**Конечный доход:** -30500

**Конечный заряд:** 5180

Вывод: результаты одинаковы.

Ссылка на репозиторий: <https://github.com/Aglomiras/LR4_Optimize>

Код:

import random  
  
import matplotlib.pyplot as plt  
from deap import base  
from deap import creator  
from deap import tools  
  
# ----------------------------------------------------------------------------------------------------------------------  
# ----------------------------------------Блок инициализации исходных данных--------------------------------------------  
# ----------------------------------------------------------------------------------------------------------------------  
capacity = 16000 # емкость персонального энергоблока (ПЭБ)  
initCharge = 6000 # уровень заряда ПЭБ  
  
'''Почасовая цена за электроэнергию'''  
priceSchedule = [1.5, 1.5, 1.5, 1.5, 1.5, 1.5,  
 2.0, 3.0, 5.0, 5.0, 5.0, 4.5,  
 3.0, 3.0, 3.0, 3.0, 4.5, 5.0,  
 7.0, 9.0, 11.0, 12.0, 8.0, 4.0]  
  
'''Почасовое потребление электроэнергии'''  
loadSchedule = [480, 320, 320, 360, 360, 360,  
 420, 920, 1200, 720, 680, 720,  
 800, 820, 960, 1200, 1380, 1380,  
 1520, 1800, 1920, 1920, 1640, 1020]  
  
constantLoad = 400 # потребитель с постоянной нагрузкой  
targetCharge = 4800 # конечный заряд аккумулятора  
  
'''Временная ось'''  
time = [0]  
for i in range(len(loadSchedule)):  
 time.append(i + 1)  
  
'''Константы для генетического алгоритма'''  
POPULATION\_SIZE = 200 # Размер популяции  
P\_CROSSOVER = 0.9 # Шанс скрещивания  
P\_MUTATION = 0.1 # Шанс мутации  
MAX\_GENERATION = 500 # Максимальное количество поколений  
  
operations = [-4000, -3000, -2000, -1000, 0, 1000, 2000, 3000, 4000] # Индивид  
  
NUMBER\_OF\_HOURS = 24  
  
  
# ----------------------------------------------------------------------------------------------------------------------  
# ----------------------------------------------------------------------------------------------------------------------  
  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Инициализация начального индивида (24 гена составляют одну хромосому(индивид))  
# Геном является массив из 9 элементов, содержащий 0 и 1  
# [0] - соответствует торговой операции, которая не выполняется в текущий час;  
# [1] - соответствует торговой операции, которая производится в текущий час;  
# единица может быть только одна, так как можно проводить только одну операцию в час  
# ----------------------------------------------------------------------------------------------------------------------  
def gen\_Initial():  
 operation = []  
 for i in range(len(operations) - 1):  
 operation.append(0)  
  
 operation.append(1)  
 return operation  
  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Расчет загрузки на данный час (динамическое потребление + статическое потребление)  
# ----------------------------------------------------------------------------------------------------------------------  
def calc\_load\_given\_hour(load):  
 return constantLoad + load  
  
  
# Получение листа загрузки на текущий час  
load\_present\_schedule = list(map(calc\_load\_given\_hour, loadSchedule))  
  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Производим мутацию: находим индекс, под которым в индивиде содержит 1 -> меняем на 0  
# выбираем новый индекс, отличный от старого и присваиваем ему 1, тем самым получая новую последовательность  
# ----------------------------------------------------------------------------------------------------------------------  
def mutation\_gen\_operation(muta):  
 mutate\_list = muta  
 ind = 0  
 for i in range(len(muta)):  
 if mutate\_list[i] == 1:  
 mutate\_list[i] = 0  
 ind = i  
  
 ind\_change = ind  
 while ind == ind\_change:  
 ind\_change = random.randint(0, 8)  
  
 mutate\_list[ind\_change] = 1  
 return mutate\_list  
  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Базисные значения заряда и дохода для расчета приспособленности особей в популяции  
# ----------------------------------------------------------------------------------------------------------------------  
basis\_charge = 0  
for i in range(len(load\_present\_schedule)):  
 basis\_charge = basis\_charge + max(operations) - load\_present\_schedule[i]  
basis\_charge = initCharge - targetCharge  
  
basis\_income = 0  
for i in range(len(load\_present\_schedule)):  
 basis\_income = basis\_income - min(operations) \* priceSchedule[i]  
  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Функция мутации:  
# [1] на вход поступает индивид (это 24 торговых операции)  
# [2] с вероятностью 1 / (число генов в хромосоме = 24 ч) выбирается ген, над которым будет производиться мутация  
# [3] мутация производится и ген перезаписывается в индивиде  
# ----------------------------------------------------------------------------------------------------------------------  
def mutation(ind\_mutation, indpb):  
 for i in range(len(ind\_mutation)):  
 if random.random() < indpb:  
 ind\_mutation[i] = mutation\_gen\_operation(ind\_mutation[i])  
 return ind\_mutation  
  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Функция расчета приспособленности индивида  
# ----------------------------------------------------------------------------------------------------------------------  
def funct\_Fitness(individual):  
 income = 0 # Доход текущего решения  
 load\_charge = 0 # Загрузка текущего решения  
 flag\_possibility = True # Флаг допустимости решения  
  
 for i in range(len(individual)):  
 charge\_operate = 0 # Изменение заряд после проведенной операции  
 for operate, charge in zip(individual[i], operations):  
 charge\_operate = charge\_operate + operate \* charge  
  
 load\_charge = load\_charge + charge\_operate - load\_present\_schedule[i] # Заряд в конце часа  
  
 if load\_charge + initCharge > capacity or load\_charge + initCharge < 0:  
 flag\_possibility = False # Такое состояние батареи невозможно  
  
 income = income - charge\_operate \* priceSchedule[i] # Обновление дохода  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Расчет приспособленности (учет дохода и учет конечной энергии батареи)  
 # Базисные величины соответствуют максимальному убытку (только покупке электроэнергии)  
 # ------------------------------------------------------------------------------------------------------------------  
 delta = (load\_charge + initCharge - targetCharge) / basis\_charge # Может быть и отрицательной и положительной  
 fitness = abs(delta) \* (basis\_income - income) / basis\_income \* (-1)  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Намеренное ухудшение приспособленности индивида, который содержит недопустимое решение задачи  
 # ------------------------------------------------------------------------------------------------------------------  
 if not flag\_possibility:  
 fitness = fitness + 5 \* fitness  
  
 return fitness  
  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Создание класса индивида  
# ----------------------------------------------------------------------------------------------------------------------  
creator.create("Fitness", base.Fitness, weights=(1.0,)) # Использование стандартной функции Fitness  
creator.create("Individual", list, fitness=creator.Fitness) # Создаем самого класс индивидуума  
  
toolbox = base.Toolbox() # Создание экземпляра класса toolbox для регистрации функций  
  
# ----------------------------------------------------------------------------------------------------------------------  
# ---------------------------------------Генерация одного гена для индивида---------------------------------------------  
# ----------------------------------------------------------------------------------------------------------------------  
toolbox.register("zeroOrOne", gen\_Initial)  
  
# ----------------------------------------------------------------------------------------------------------------------  
# -------------------------------------Создаем начальную популяцию хромосом---------------------------------------------  
# ----------------------------------------------------------------------------------------------------------------------  
# Создаем индивида, используя функцию zeroOrOne, то есть индивид состоит из 24 списков, которые содержат набор  
# 0 и 1, которые обозначают торговую операцию (за основу взята задача OneMax)  
# ----------------------------------------------------------------------------------------------------------------------  
toolbox.register("initial\_population", tools.initRepeat, creator.Individual, toolbox.zeroOrOne, NUMBER\_OF\_HOURS)  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Создаем функцию для генерации популяции, но не указываем количество индивидов в популяции  
# это значение указывается позже  
# ----------------------------------------------------------------------------------------------------------------------  
toolbox.register("population\_creator", tools.initRepeat, list, toolbox.initial\_population)  
  
# Создание первой популяции размером POPULATION\_SIZE = 200 и инициализация счетчика поколений  
population = toolbox.population\_creator(n=POPULATION\_SIZE)  
generation\_counter = 0 # Счетчик поколений  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Псевдоним функции отбора (стандартный)  
# ----------------------------------------------------------------------------------------------------------------------  
toolbox.register("select", tools.selTournament, tournsize=3)  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Псевдоним функции скрещивания(стандартный)  
# ----------------------------------------------------------------------------------------------------------------------  
toolbox.register("mate", tools.cxOnePoint)  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Псевдоним функции мутации (моя реализация)  
# ----------------------------------------------------------------------------------------------------------------------  
toolbox.register("mutate", mutation, indpb=1.0 / NUMBER\_OF\_HOURS)  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Псевдоним функция расчета приспособленности (моя реализация)  
# ----------------------------------------------------------------------------------------------------------------------  
toolbox.register("evaluate", funct\_Fitness)  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Вычисление приспособленности групп индивидуумов первого поколения  
# ----------------------------------------------------------------------------------------------------------------------  
fitness\_values = list(map(toolbox.evaluate, population))  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Присваиваем конкретному свойству классу Individual значения fitness\_values  
# ----------------------------------------------------------------------------------------------------------------------  
for individual\_group, fitness\_value in zip(population, fitness\_values):  
 individual\_group.fitness.values = (fitness\_value,)  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Создаем коллекцию значений приспособленности данной популяции  
# ----------------------------------------------------------------------------------------------------------------------  
fitness\_values = [individual.fitness.values[0] for individual in population]  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Для хранения статистики  
# ----------------------------------------------------------------------------------------------------------------------  
max\_fitness\_values = [] # Максимальная приспособленность особей в текущей популяции  
mean\_fitness\_values = [] # Средняя приспособленность всех особей в текущей популяции  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Генетический алгоритм  
# Цикл продолжается пока не пройдет указанное число поколений, либо пока не будет найдено лучшее решение  
# ----------------------------------------------------------------------------------------------------------------------  
while max(fitness\_values) < -0.3399 and generation\_counter < MAX\_GENERATION:  
 generation\_counter += 1 # Номер поколения  
  
 offspring = toolbox.select(population, len(population)) # Отбор лучших особей  
 offspring = list(map(toolbox.clone, offspring)) # Клонирование отобранных особей для нового списка  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Скрещивание  
 # Берем четные и нечетные по порядку элементы и производим одноточечное скрещивание  
 # ------------------------------------------------------------------------------------------------------------------  
 for child1, child2 in zip(offspring[::2], offspring[1::2]):  
 if random.random() < P\_CROSSOVER: # Проверка вероятности скрещивания  
 toolbox.mate(child1, child2)  
 # ----------------------------------------------------------------------------------------------------------  
 # Удаление значений приспособленности скрещенных индивидуумов  
 # ----------------------------------------------------------------------------------------------------------  
 del child1.fitness.values  
 del child2.fitness.values  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Мутация  
 # ------------------------------------------------------------------------------------------------------------------  
 for mutant in offspring:  
 if random.random() < P\_MUTATION: # проверка вероятности мутации индивида  
 toolbox.mutate(mutant)  
 # ----------------------------------------------------------------------------------------------------------  
 # Удаление значения приспособленности мутировавшего индивидуума  
 # ----------------------------------------------------------------------------------------------------------  
 del mutant.fitness.values  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Повторная оценка скрещенных или мутировавших индивидов  
 # ------------------------------------------------------------------------------------------------------------------  
 fresh\_individuals = [ind for ind in offspring if not ind.fitness.valid]  
 fresh\_fitness\_values = list(map(toolbox.evaluate, fresh\_individuals)) # Новый лист приспособленностей  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Запись значений приспособленности индивида  
 # ------------------------------------------------------------------------------------------------------------------  
 for individual, fitnessValue in zip(fresh\_individuals, fresh\_fitness\_values):  
 individual.fitness.values = (fitnessValue,)  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Обновляем список популяции  
 # ------------------------------------------------------------------------------------------------------------------  
 population[:] = offspring  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Обновляем список приспособленности популяции  
 # ------------------------------------------------------------------------------------------------------------------  
 fitness\_values = [ind.fitness.values[0] for ind in population]  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Определяем максимальную и среднюю приспособленности  
 # ------------------------------------------------------------------------------------------------------------------  
 max\_fitness = max(fitness\_values)  
 mean\_fitness = sum(fitness\_values) / len(population)  
 max\_fitness\_values.append(max\_fitness)  
 mean\_fitness\_values.append(mean\_fitness)  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Вывод информации ы консоль  
 # ------------------------------------------------------------------------------------------------------------------  
 print(f"Поколение {generation\_counter}: Макс. приспособ. = {max\_fitness}, Средняя приспособ. = {mean\_fitness}")  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Цикл проверки итоговых результатов  
# Если среди конечной популяции все решения приводят к несоответствию действительной работы батареи, то  
# выведется соответствующее сообщение об отсутствии допустимых решений. Если в конечной популяции, есть индивид,  
# который удовлетворяет работе батареи, его результат будет выведен н графике  
# ----------------------------------------------------------------------------------------------------------------------  
energy = [] # Хранит массив энергии в конце каждого часа  
battery\_damage = False # Флаг для определения допустимых решений (False - решения есть, True - решений нет)  
count\_ind = 0 # Счетчик  
while count\_ind < POPULATION\_SIZE:  
 count\_ind += 1  
 # ------------------------------------------------------------------------------------------------------------------  
 # Обновляемые параметры  
 # ------------------------------------------------------------------------------------------------------------------  
 best\_index = fitness\_values.index(max(fitness\_values)) # Индекс лучшего решения (по приспособленности)  
 best\_individual = population[best\_index] # Индивид с лучшей приспособленностью  
 battery\_energy = [initCharge] # Начальный массив значений энергии(заряда батареи) в конце каждого часа  
 best\_income = 0 # Доход от торговых операций  
 energy\_end\_hour = initCharge # Переменная для хранения энергии батареи в конце часа  
 flag\_batt = False # Флаг допустимости текущего решения  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Определяем самого приспособленного (лучшего) индивида и проверяем его на допустимость решения  
 # поставленной задачи. Если он не удовлетворяет требованиям, его стирают из списка популяции и определяют  
 # следующего по приспособленности индивида и повторяют операцию допустимости решения.  
 # ------------------------------------------------------------------------------------------------------------------  
 for i in range(len(best\_individual)):  
 charge\_operate = 0 # Изменение заряд после проведенной торговой операции  
 # --------------------------------------------------------------------------------------------------------------  
 # Индивид состоит из 24 листов, которые представляют собой набор 0 и 1 соответствующие торговой  
 # операции. То есть, 1 час (й такой лист) - это геном, который по факту надо расшифровать.  
 # Итерируемся по всему геному, в поисках заветной 1, чтобы перемножить ее на соответсвующую  
 # торговую операцию и получить значений дохода.  
 # --------------------------------------------------------------------------------------------------------------  
 for operate, charge in zip(best\_individual[i], operations):  
 charge\_operate = charge\_operate + operate \* charge  
  
 energy\_end\_hour = energy\_end\_hour + charge\_operate - load\_present\_schedule[i] # Заряд в конце часа  
  
 # --------------------------------------------------------------------------------------------------------------  
 # Проверяем условие допустимости текущего решения по каждому часу  
 # Если заряд батареи в конце часа опускается ниже 0 или становится выше максимальной емкости (capacity)  
 # в этом случае решение считается недопустимым  
 # --------------------------------------------------------------------------------------------------------------  
 if energy\_end\_hour > capacity or energy\_end\_hour < 0:  
 flag\_batt = True # Такое состояние батареи невозможно  
  
 battery\_energy.append(energy\_end\_hour) # Добавляем значение энергии(заряда) батареи в конце текущего часа  
  
 best\_income = best\_income - charge\_operate \* priceSchedule[i] # Обновляем переменную дохода  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Обновляем параметры  
 # ------------------------------------------------------------------------------------------------------------------  
 battery\_damage = flag\_batt  
 energy[:] = battery\_energy  
  
 # ------------------------------------------------------------------------------------------------------------------  
 # Проверка условия допустимости решения:  
 # [False] - С батарее все нормально, такое решение возможно, вывод успеха в консоль и построение графиков  
 # [True] - переход к другому решению  
 # ------------------------------------------------------------------------------------------------------------------  
 if flag\_batt:  
 del fitness\_values[best\_index] # Удаляем провальное решение из популяции  
 continue  
 else:  
 print("статистика лучшего индивидуума")  
 print(f"\nРезультат торгов: {best\_income} \n"  
 f"Остаточный заряд: {energy\_end\_hour} \n")  
 break  
  
if battery\_damage:  
 print("Допустимые решения отсутствуют")  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Расчет изменения заряда ПЭБ в течение суток без проведения плановых торговых операций  
# Учитываем допустимое состояние батареи, энергия батареи не должна упасть ниже 0  
# ----------------------------------------------------------------------------------------------------------------------  
res\_level\_energy\_mass = [initCharge]  
res\_energy = initCharge  
  
for i in range(len(load\_present\_schedule)):  
 res\_energy = res\_energy - load\_present\_schedule[i]  
 if res\_energy > 0:  
 res\_level\_energy\_mass.append(res\_energy)  
 else:  
 delta\_energy = (load\_present\_schedule[i] - res\_energy)  
  
 if delta\_energy <= 1000:  
 res\_energy += 1000  
 elif delta\_energy <= 2000:  
 res\_energy += 2000  
 elif delta\_energy <= 3000:  
 res\_energy += 3000  
 else:  
 res\_energy += 4000  
  
 res\_level\_energy\_mass.append(res\_energy)  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Построение графиков  
# ----------------------------------------------------------------------------------------------------------------------  
fig, ax = plt.subplots(nrows=1, ncols=2)  
plt.subplots\_adjust(wspace=10, hspace=0.2, left=0.06, right=0.98, top=0.96, bottom=0.1) # Отступы по краям  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Распределение приспособленностей  
# ----------------------------------------------------------------------------------------------------------------------  
plt.subplot(2, 1, 1)  
plt.grid(True, color="grey", linewidth="1.4", linestyle="-.")  
plt.plot(max\_fitness\_values, "r")  
plt.plot(mean\_fitness\_values, "g")  
plt.title("Распределение максимальной и средней приспособленности поколений", fontsize=9)  
plt.legend(("max", "mean"))  
  
# ----------------------------------------------------------------------------------------------------------------------  
# Графики изменения энергии  
# ----------------------------------------------------------------------------------------------------------------------  
plt.subplot(2, 1, 2)  
plt.grid(True, color="grey", linewidth="1.4", linestyle="-.")  
plt.plot(energy, "k")  
plt.plot(res\_level\_energy\_mass, "r")  
plt.title("Изменение заряда ПЭБ", fontsize=9)  
plt.legend(("с торгами", "без торгов"))  
plt.show()

**Вывод:**

В ходе лабораторной работы были получены практические навыки работы с генетическим алгоритмом для решения задач оптимизации. Был написан сам генетический алгоритм для решения задачи часовой загрузки ПЭБ с получением максимальной выгоды от произведенных торговых операций. За основу структуры алгоритма была взята реализация генетического алгоритма для задачи oneMax.

В ходе проведения опытов выяснилось, что с увеличением числа поколений, возрастает вероятность нахождения наилучшего решения данной задачи.

В ходе проведения опытов было выяснено, что скрещивание особей значительно ускоряет процесс нахождения лучшего решения, а мутация помогает поддерживать разнообразие особей в текущей популяции и возвращать время от времени более оптимальные гены, которые могли исчезнуть в процессе скрещивания.

Отбор особей проводился в виде турнира с использованием стандартной функции selTournament пакета Deap.