|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** |
| Институт информационных технологий |
| Кафедра корпоративных информационных систем  **КУРСОВАЯ РАБОТА**  по дисциплине  Системы синхронизированного планирования ресурсов предприятия  **Тема курсовой работы**: Решение задачи о назначении с применением генетического алгоритма   |  |  |  | | --- | --- | --- | | Студент группы | ИКМО-03-23  Монаков Андрей Вячеславович | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись студента) | |  |  |  | | Руководитель курсовой работы | профессор  Демидова Лилия Анатольевна | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись руководителя) | |  |  |  | | Работа представлена к защите | «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_ 2025 г. |  | |  |  |  | | Допущен к защите | «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_ 2025 г. |  | |

Москва 2025

****

**Оглавление**

[Введение 4](#_Toc193076300)

[1. Теоретическая часть 5](#_Toc193076301)

[1.1. Генетический алгоритм 5](#_Toc193076302)

[1.2. Сущность задачи о назначении 7](#_Toc193076303)

[2. Реализация задачи о назначении 9](#_Toc193076304)

[2.1. Реализации генетического алгоритма 10](#_Toc193076305)

[2.2. Полная реализация решения задачи о назначении 12](#_Toc193076306)

[Заключение 22](#_Toc193076307)

[Список источников 23](#_Toc193076308)

[Приложение А 24](#_Toc193076309)

# Введение

В текущую эпоху развития информационных технологий приходится сталкиваться со сбором, хранением и передачей огромных объемов данных, которые необходимо изучать, обрабатывать и применять выводы, полученные на их основе для дальнейшего развития. Чтобы данные задачи были выполнимы существует целый пласт алгоритмов по работе с данными, каждый из которых лучше подходит для разных случаев.

Одним из наиболее востребованных типов алгоритмов являются алгоритмы оптимизации данных. К ним относятся генетические алгоритмы и алгоритмы, основанные на поведении различных животных.

Целью данной работы является изучение задачи о назначении с применением генетического алгоритма. В рамках работы будут рассмотрены основные подходы к реализации алгоритма, будет исследована специфика предметной области и действующие в ней ограничения, а также представлена реализация алгоритма с решением поставленной в рамках темы задачи.

# 1. Теоретическая часть

## 1.1. Генетический алгоритм

Генетический алгоритм (GA) был предложен Джоном Холландом (J. Holland) в 1975 году. Он основан на принципах естественного отбора и генетики. В алгоритме используется популяция потенциальных решений, представленных в виде хромосом, которые эволюционируют по мере выполнения алгоритма. Каждая хромосома состоит из генов, представляющих различные параметры задачи [6].

Процесс работы генетического алгоритма начинается с инициализации начальной популяции случайным образом. Каждая хромосома оценивается с помощью фитнес-функции, которая измеряет качество решения. Алгоритм повторяет цикл выбора, скрещивания и мутации до достижения заданного критерия остановки [7].

Этапы генетического алгоритма:

1. Инициализация: Создание начальной популяции хромосом случайным образом.

2. Оценка: Вычисление фитнес-функции для каждой хромосомы.

3. Выбор: Отбор хромосом для создания новой популяции на основе их фитнеса. Часто используются методы рулеточного колеса или турнирного отбора.

4. Скрещивание (кроссинговер): Комбинирование пар хромосом для создания потомков. Обычно применяются одноточечный, двуточечный или униформный кроссинговеры.

5. Мутация: Внесение случайных изменений в гены хромосом для поддержания генетического разнообразия.

6. Замещение: Замена старой популяции новым поколением хромосом.

Генетический алгоритм формально описывается следующими уравнениями [6]:

Формула фитнес-функции:

(1.1)  
где x ‒ текущая хромосома, представляющая решение задачи.

Формула отбора:

(1.2)  
где P*i* ‒ вероятность отбора хромосомы x*i*, n ‒ размер популяции.

Формула кроссинговера:

*новыйродитель1родитель2*,(1.3)  
где alpha ‒ случайная величина в диапазоне [0, 1].

Формула мутации:

*новый* (1.4)  
где delta ‒ случайное отклонение.

Преимущества генетических алгоритмов [7]:

1. Эффективность в больших пространствах поиска: ГА способны исследовать сложные и многомерные пространства решений, что делает их подходящими для задач с высокой вычислительной сложностью.
2. Адаптивность: Алгоритм может адаптироваться к изменяющимся условиям задачи, что особенно полезно в динамических средах.
3. Робастность: ГА устойчивы к шуму и локальным оптимумам благодаря использованию операторов мутации и кроссовера.
4. Универсальность: Применимы к широкому кругу задач, включая дискретные, непрерывные и комбинаторные оптимизации.

Генетические алгоритмы находят широкое применение в различных областях, таких как [7]:

1. Обучение нейронных сетей.
2. Тестирование программного обеспечения
3. Криптография
4. Обработка сигналов
5. Обучение нечетких систем

Существуют различные модели генетического алгоритма, которые отличаются стратегиями отбора, операторами и способами кодирования генов. в данной практической работе используется не только базовая, но и островная модель [8].

Островная модель (Island Model) — это параллельная версия генети, где популяция делится на несколько подпопуляций ("островов"). Особенности:

* Изолированная эволюция: Каждая подпопуляция развивается независимо, используя свой генетический алгоритма.
* Миграция: периодически острова обмениваются лучшими особями. Это позволяет избежать преждевременной сходимости и улучшить качество решений.
* Частота миграции: важно правильно настроить частоту и количество мигрирующих особей. Слишком частая миграция приводит к смешению подпопуляций, а слишком редкая — к преждевременной сходимости.

Генетические алгоритмы представляют собой мощный инструмент для решения сложных задач оптимизации и поиска. Их способность исследовать большие пространства решений, адаптироваться к изменяющимся условиям и находить глобальные оптимумы делает их популярным выбором в различных областях. Различные модели, такие как CHC и островная модель [8], позволяют адаптировать алгоритм под конкретные задачи и улучшать качество решений. В будущем развитие генетических алгоритмов может быть связано с их интеграцией с методами искусственного интеллекта и машинного обучения для решения еще более сложных задач.

## 1.2. Сущность задачи о назначении

Задача о назначении представляет собой сложную задачу планирования времени и распределения ресурсов с учётом множества ограничений. Эта задача широко применяется в различных областях, таких как производство, логистика, образование и управление персоналом, где важно эффективно организовать выполнение работ для минимизации конфликтов и максимизации использования ресурсов.

Дано:

* Набор работ: Каждая работа имеет свои временные рамки (время начала и окончания), длительность, а также требования к ресурсам. Некоторые работы должны выполняться последовательно, что накладывает дополнительные ограничения на порядок их выполнения.
* Набор работников: Работники обладают определёнными навыками, которые позволяют им выполнять только некоторые работы качественно. Каждый работник имеет ограничения по рабочему времени (не более 8 часов в день) и должен иметь обеденный перерыв продолжительностью 30 минут.
* Ограничения:
  + Работы должны выполняться в заданные временные рамки.
  + Работник не может быть назначен на работу, которую он не может выполнить качественно.
  + Работы, зависящие друг от друга, должны выполняться в определённом порядке.
  + Рабочее время работника не должно превышать 8 часов в день, включая обеденный перерыв.

Необходимо:

* Составить оптимальное назначение для выполнения работ с учётом заданных ограничений.
* Минимизировать штрафы за нарушения ограничений, такие как:
  + Нарушение порядка выполнения работ.
  + Назначение работника на работу, которую он не может выполнить качественно.
  + Превышение рабочего времени.
  + Нарушение обеденного перерыва.
  + Повторение одной и той же работы одним и тем же работником
* Максимизировать эффективность использования рабочего времени и ресурсов.

Учёт времени, ресурсов, приоритетов и зависимостей:

* Время выполнения: Каждая работа имеет определённое время начала и окончания, которое должно быть строго соблюдено.
* Ресурсы: Каждая работа требует определённых навыков, которые должны быть доступны у назначенного работника.
* Зависимости: Некоторые работы зависят от выполнения других работ, что требует соблюдения порядка выполнения.

Таким образом, задача о назначении заключается в нахождении оптимального распределения времени и ресурсов для выполнения всех работ с учётом всех заданных ограничений и приоритетов. Решение этой задачи позволяет минимизировать штрафы за нарушения и обеспечить эффективное использование рабочего времени и ресурсов.

# 2. Реализация задачи о назначении

Чтобы реализовать решение данной задачи, необходимо применить генетический алгоритм и использовать систему штрафов из таблицы 1.

Таблица 1 – Система штрафов

|  |  |  |
| --- | --- | --- |
| **№** | Критерий | Штрафы, баллы |
| 1. | За нарушение порядка выполнения 2 последовательных работ. | 100 |
| 2. | За обратный порядок выполнения последовательных работ. | Число последовательных работ\*100 |
| 3. | За повторное выполнение работ, тем же работником. | 80 |
| 4. | За одновременное выполнение нескольких работ одним работником | 100 |
| 5. | За назначение работника на работу, которую он может выполнить некачественно | 80 |
| 6. | За рабочий день более 8 часов | 100 |
| 7. | За обеденный перерыв менее одного часа и не в рамках интервала [12-00, 14-00]. | 100 |

## **2.1. Реализации генетического алгоритма**

1. Инициализация популяции

Функция create\_schedule создает начальное назначение, назначая задачи работникам случайным образом.

* Для каждого работника задачи назначаются с учетом их навыков.
* Если задача не может быть назначена на работника с соответствующими навыками, она назначается случайному работнику.
* Возвращает назначение в виде списка задач для каждого работника.

Входные данные:

* + Список работников (workers) с их навыками.
  + Список работ (works) с временными рамками и зависимостями.

Выходные данные:

* + Список назначений для каждого работника.

2. Оценка приспособленности

Функция evaluate\_schedule рассчитывает штрафы для назначения на основе нарушений ограничений:

* Нарушение порядка выполнения работ.
* Назначение на работу, которую работник не может выполнить качественно.
* Превышение рабочего времени (более 8 часов).
* Нарушение обеденного перерыва (менее часа и вне интервала [12:00, 14:00]).
* Повторное выполнение работ, тем же работником.
* Одновременное выполнение нескольких работ

Входные данные:

* + Назначение работников (schedule).

Выходные данные:

* + Суммарный штраф для назначения.

3. Селекция

Функция select\_best\_schedules выбирает лучшие назначения из популяции на основе штрафов. Используется сортировка по возрастанию штрафов.

Входные данные:

* + Популяция назначений (population).
  + Количество лучших назначений для выбора (num\_best).

Выходные данные:

* + Список лучших назначений.

4. Мутация

Функция mutate\_schedule вносит случайные изменения в назначение:

* Выбирает случайного работника и случайную задачу в его назначении.
* Может изменить время выполнения задачи или передать её другому работнику.
* Возвращает измененное назначение.

Входные данные:

* + Назначение работников (schedule).

Выходные данные:

* + Мутированное назначение.

5. Кроссовер

Функция crossover выполняет скрещивание двух родительских назначений, создавая два новых назначения (потомка). В данном случае используется одноточечный кроссовер, где задачи от родителей случайным образом распределяются между потомками.

Входные данные:

* + Родители.

Выходные данные:

* + Потомки.

6. Создание новой популяции

Функция genetic\_algorithm реализует основной цикл ГА:

* Создание начальной популяции.
* Оценка приспособленности.
* Селекция лучших решений.
* Мутация, кроссовер и создание новой популяции.
* Повторение процесса для заданного числа поколений.

Входные данные:

* + Количество поколений (num\_generations).
  + Размер популяции (population\_size).

Выходные данные:

* + Лучшее назначение.
  + Список штрафов для каждого поколения.

7. Островная модель

Функция island\_model реализует островную модель, где популяции разделены на несколько "островов". Между островами происходит миграция лучших решений, что позволяет улучшить качество оптимизации.

Входные данные:

* + Количество поколений (num\_generations).
  + Размер популяции (population\_size).
  + Количество островов (num\_islands).
  + Частота миграции (migration\_rate).

Выходные данные:

* + Лучшее назначение.
  + Список штрафов для каждого поколения.

## 2.2. Полная реализация решения задачи о назначении

Работы и работники задаются в виде списка словарей. Каждая работа и работник описываются своими характеристиками.

Листинг 1 - Инициализация данных

works = [

{"name": "Подготовка деталей", "start\_time": 10, "end\_time": 15, "duration": 1, "dependencies": []},

{"name": "Сборка лампы", "start\_time": 10, "end\_time": 15, "duration": 2, "dependencies": ["Покраска корпуса"]},

{"name": "Покраска корпуса", "start\_time": 10, "end\_time": 15, "duration": 1, "dependencies": ["Подготовка деталей"]},

{"name": "Тестирование", "start\_time": 10, "end\_time": 16, "duration": 2, "dependencies": []},

{"name": "Упаковка", "start\_time": 14, "end\_time": 17, "duration": 1, "dependencies": ["Тестирование"]},

{"name": "Изготовление плафона", "start\_time": 14, "end\_time": 17, "duration": 1, "dependencies": ["Сборка лампы"]},

{"name": "Проверка электроники", "start\_time": 12, "end\_time": 15, "duration": 1, "dependencies": []},

{"name": "Настройка светильника", "start\_time": 14, "end\_time": 16, "duration": 1, "dependencies": ["Сборка лампы"]},

{"name": "Закупка материалов", "start\_time": 9, "end\_time": 13, "duration": 1, "dependencies": []},

{"name": "Доставка готовой продукции", "start\_time": 15, "end\_time": 17, "duration": 1, "dependencies": ["Упаковка"]},

{"name": "Контроль качества", "start\_time": 14, "end\_time": 17, "duration": 1, "dependencies": ["Тестирование"]},

{"name": "Установка лампы", "start\_time": 14, "end\_time": 17, "duration": 2, "dependencies": ["Сборка лампы"]},

{"name": "Ремонт оборудования", "start\_time": 10, "end\_time": 14, "duration": 2, "dependencies": ["Подготовка деталей"]},

{"name": "Обучение персонала", "start\_time": 12, "end\_time": 17, "duration": 2, "dependencies": []},

{"name": "Уборка рабочего места", "start\_time": 16, "end\_time": 17, "duration": 1, "dependencies": []},

]

# Работник

workers = [

{"name": "Мария", "skills": ["Покраска корпуса", "Тестирование", "Проверка электроники", "Контроль качества"], "max\_hours": 8},

{"name": "Петр", "skills": ["Упаковка", "Тестирование", "Настройка светильника", "Установка лампы"], "max\_hours": 8},

{"name": "Алексей", "skills": ["Закупка материалов", "Подготовка деталей", "Доставка готовой продукции", "Обучение персонала"], "max\_hours": 8},

{"name": "Дмитрий", "skills": ["Сборка лампы", "Проверка электроники", "Ремонт оборудования", "Обучение персонала", "Настройка светильника"], "max\_hours": 8},

]

Штрафы задаются как константы. Они используются для оценки качества назначения.

Листинг 2 - Штрафы

PENALTY\_ORDER\_VIOLATION = 100 # Нарушение порядка выполнения

PENALTY\_LOW\_QUALITY = 80 # Назначение на работу, которую работник не может выполнить качественно

PENALTY\_OVERTIME = 100 # Рабочий день более 8 часов

PENALTY\_SHORT\_LUNCH = 100 # Обеденный перерыв менее часа и не в 12-14

PENALTY\_REPEATED\_WORK = 80 # Штраф за повторное назначение одной и той же работы одному сотруднику

PENALTY\_OVERLAP = 100 # Штраф за одновременное выполнение нескольких работ одним работником

PENALTY\_UNASSIGNED\_WORK = 1000 # Штраф за невыполненную работу

PENALTY\_INCOMPLETE\_WORK = 1000 # Штраф за неполное выполнение задачи

PENALTY\_TIME\_VIOLATION = 1000 # Штраф за нарушение временного интервала

Функция create\_schedule создает начальное назначение для каждого работника. Для каждой работы, которую работник может выполнить, случайным образом выбирается время начала в пределах допустимого интервала.

Листинг 3 – Создание начального назначения

def create\_schedule():

schedule = [[] for \_ in workers] # Назначение для каждого работника

assigned\_works = set() # Множество для отслеживания назначенных работ

for work in works:

# Выбираем случайного работника, который может выполнить задачу

possible\_workers = [worker\_idx for worker\_idx, worker in enumerate(workers) if work["name"] in worker["skills"]]

if not possible\_workers:

# Если никто не может выполнить задачу, выбираем любого работника

possible\_workers = range(len(workers))

worker\_idx = random.choice(possible\_workers)

start\_time = random.randint(work["start\_time"], work["end\_time"] - work["duration"])

schedule[worker\_idx].append((work["name"], start\_time, start\_time + work["duration"]))

assigned\_works.add(work["name"])

return schedule

Функция evaluate\_schedule оценивает назначение, рассчитывая штрафы за нарушения ограничений. Каждый тип нарушения проверяется отдельно.

Листинг 4 – Оценка назначения

ef evaluate\_schedule(schedule):

penalty = 0

penalty\_details = [] # Список для хранения информации о штрафах

# Создаем словарь для хранения времени выполнения задач

task\_times = {}

for worker\_schedule in schedule:

for task in worker\_schedule:

task\_name, start\_time, end\_time = task

task\_times[task\_name] = (start\_time, end\_time)

# Проверяем порядок выполнения задач

for work in works:

for dependency in work["dependencies"]:

if dependency in task\_times and work["name"] in task\_times:

dependency\_start, dependency\_end = task\_times[dependency]

work\_start, work\_end = task\_times[work["name"]]

# Проверяем, выполняется ли задача до завершения зависимости

if work\_start < dependency\_end:

penalty += PENALTY\_ORDER\_VIOLATION

penalty\_details.append(f"Нарушение порядка выполнения: {work['name']} начинается до завершения {dependency} (штраф: {PENALTY\_ORDER\_VIOLATION})")

    return penalty, penalty\_details

Функция mutate\_schedule вносит случайные изменения в назначение.

Листинг 5 – Мутация назначения

def mutate\_schedule(schedule):

mutated\_schedule = [worker\_schedule.copy() for worker\_schedule in schedule]

# Выбираем случайного работника

worker\_idx = random.randint(0, len(mutated\_schedule) - 1)

worker\_schedule = mutated\_schedule[worker\_idx]

if len(worker\_schedule) > 0:

# Выбираем случайную задачу у этого работника

task\_idx = random.randint(0, len(worker\_schedule) - 1)

task = worker\_schedule[task\_idx]

task\_name, start\_time, end\_time = task

# Варианты мутации:

if random.random() < 0.5:

# Мутация 1: Изменяем время выполнения задачи

new\_start\_time = random.randint(works[task\_idx]["start\_time"], works[task\_idx]["end\_time"] - works[task\_idx]["duration"])

worker\_schedule[task\_idx] = (task\_name, new\_start\_time, new\_start\_time + works[task\_idx]["duration"])

else:

# Мутация 2: Перемещаем задачу другому работнику

new\_worker\_idx = random.randint(0, len(mutated\_schedule) - 1)

if new\_worker\_idx != worker\_idx:

mutated\_schedule[worker\_idx].pop(task\_idx)

mutated\_schedule[new\_worker\_idx].append((task\_name, start\_time, end\_time))

return mutated\_schedule

Функция select\_best\_schedules выбирает лучшие назначения из популяции на основе штрафов

Листинг 6 – Селекция лучших назначений

def select\_best\_schedules(population, num\_best=5):

evaluated\_population = [(schedule, evaluate\_schedule(schedule)) for schedule in population]

evaluated\_population.sort(key=lambda x: x[1])

return [schedule for schedule, \_ in evaluated\_population[:num\_best]]

Функция genetic\_algorithm реализует основной цикл генетического алгоритма.

Листинг 7 – Генетический алгоритм

def genetic\_algorithm(num\_generations=500, population\_size=20):

population = [create\_schedule() for \_ in range(population\_size)]

penalties = []

for generation in range(num\_generations):

# Оценка популяции

evaluated\_population = [(schedule, evaluate\_schedule(schedule)[0]) for schedule in population]

evaluated\_population.sort(key=lambda x: x[1])

# Отбор лучших назначений

best\_schedules = [schedule for schedule, \_ in evaluated\_population[:population\_size // 2]]

# Создание нового поколения

new\_population = best\_schedules.copy()

while len(new\_population) < population\_size:

# Выбор двух родителей

parent1, parent2 = random.choices(best\_schedules, k=2)

# Скрещивание (кроссовер)

child1, child2 = crossover(parent1, parent2)

# Мутация

if random.random() < 0.1: # Вероятность мутации 10%

child1 = mutate\_schedule(child1)

if random.random() < 0.1:

child2 = mutate\_schedule(child2)

new\_population.extend([child1, child2])

population = new\_population

penalties.append(evaluated\_population[0][1])

return population[0], penalties

Функция island\_model реализует островную модель генетического алгоритма, где популяции разделены на несколько "островов".

Листинг 8 – Островная моедель.

def island\_model(num\_generations=500, population\_size=40, num\_islands=5, migration\_rate=20):

islands = [[create\_schedule() for \_ in range(population\_size)] for \_ in range(num\_islands)]

penalties = []

for generation in range(num\_generations):

for i in range(num\_islands):

# Оценка и отбор на каждом острове

evaluated\_population = [(schedule, evaluate\_schedule(schedule)[0]) for schedule in islands[i]]

evaluated\_population.sort(key=lambda x: x[1])

best\_schedules = [schedule for schedule, \_ in evaluated\_population[:population\_size // 2]]

# Создание нового поколения на острове

new\_population = best\_schedules.copy()

while len(new\_population) < population\_size:

parent1, parent2 = random.choices(best\_schedules, k=2)

child1, child2 = crossover(parent1, parent2)

if random.random() < 0.1:

child1 = mutate\_schedule(child1)

if random.random() < 0.1:

child2 = mutate\_schedule(child2)

new\_population.extend([child1, child2])

islands[i] = new\_population

# Миграция между островами

if generation % migration\_rate == 0:

best\_solutions = [select\_best\_schedules(island, num\_best=1)[0] for island in islands]

for i in range(num\_islands):

next\_island = (i + 1) % num\_islands

islands[next\_island].append(best\_solutions[i])

if len(islands[next\_island]) > population\_size:

islands[next\_island] = select\_best\_schedules(islands[next\_island], num\_best=population\_size)

penalties.append(evaluate\_schedule(select\_best\_schedules(islands[0], num\_best=1)[0])[0])

best\_schedules = [select\_best\_schedules(island, num\_best=1)[0] for island in islands]

best\_schedule = min(best\_schedules, key=lambda s: evaluate\_schedule(s)[0])

return best\_schedule, penalties, best\_schedules

Функция crossover выполняет скрещивание двух родительских назначений, создавая два новых назначения (потомка).

Листинг 9 – Кроссовер.

def crossover(parent1, parent2):

    child1 = []

    child2 = []

    for i in range(len(parent1)):

        if random.random() < 0.5:

            child1.append(parent1[i])

            child2.append(parent2[i])

        else:

            child1.append(parent2[i])

            child2.append(parent1[i])

    return child1, child2

Критерий Уилкоксона используется для сравнения двух связанных выборок (в данном случае — штрафов базовой и островной моделей). Он проверяет, есть ли статистически значимые различия между двумя наборами данных.

Листинг 10 – Критерий Уилкоксона

statistic, p\_value = Wilcoxon (penalties\_base, penalties\_island)

* Нулевая гипотеза (H₀): Нет различий между двумя моделями.
* Альтернативная гипотеза (H₁): Есть различия между двумя моделями.

Если p-value < 0.05, нулевая гипотеза отвергается, и различия считаются статистически значимыми.

Функции visualize\_schedule, run\_experiments и plot\_island\_penalties используются для визуализации назначения, сравнения моделей и анализа влияния числа островов на качество оптимизации.

Таким образом, решение задачи оптимизации функции посредством применения генетического алгоритма найдёт оптимальный набор входных параметров, при котором значение функции общего штрафа будет минимальным. Оптимальным набором, полученным в результате работы алгоритма, будет являться назначение работ при минимуме штрафов.

Для оценки эффективности работы генетического алгоритма в задаче поиска оптимального решения были замерены различные метрики, отражающие качество работы алгоритма в задаче. На рисунке 1 продемонстрировано время выполнения решения задачи двумя методами, количество штрафов, статистика Уилкоксона и p-value [9].

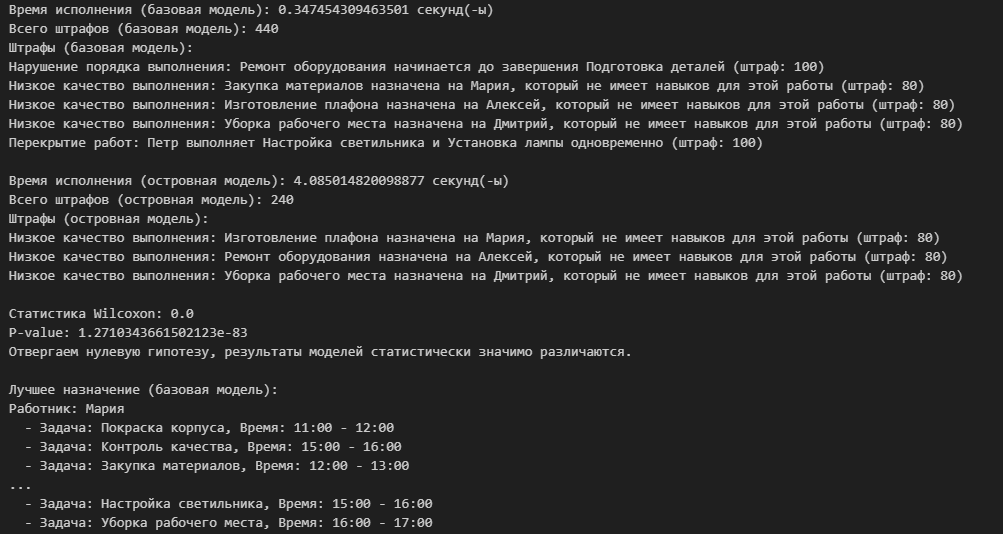


Рисунок 1 ‒ Метрики

Критерий Уилкоксона ‒ это непараметрический статистический тест, используемый для сравнения двух связанных выборок. Он проверяет гипотезу о том, что различия между парами выборок имеют симметричное распределение вокруг нуля.

Если вычисленное p-value меньше выбранного уровня значимости alpha, то нулевую гипотезу следует отвергнуть. Это означает, что существует статистически значимая разница между сравниваемыми выборками.

В приведенном примере, при alpha = 0.05, вычисленное p-value меньше 0.05. Поэтому нулевая гипотеза отвергается, и можно сделать вывод о наличии статистически значимой разницы между базовой моделью и островной моделью с 5 островами.

На рисунке 2 и 3 можно увидеть фрагмент результата решения задачи о назначении. В нем выведено лучшее назначение для базовой и островной модели с 5ю островами.

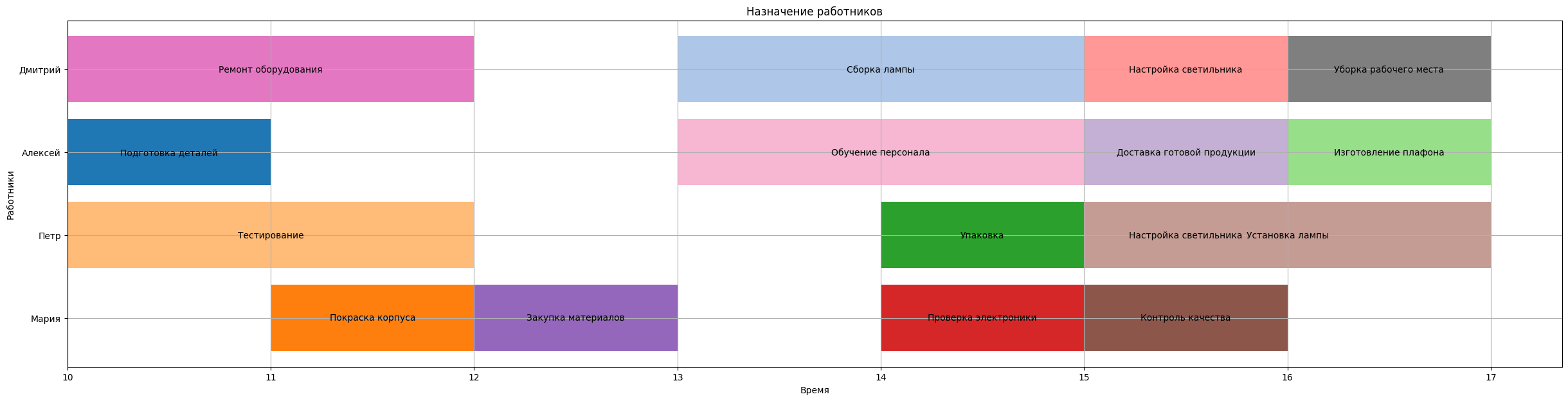


Рисунок 2 ‒ Лучшее назначение работников на рабочий день (базовая модель)

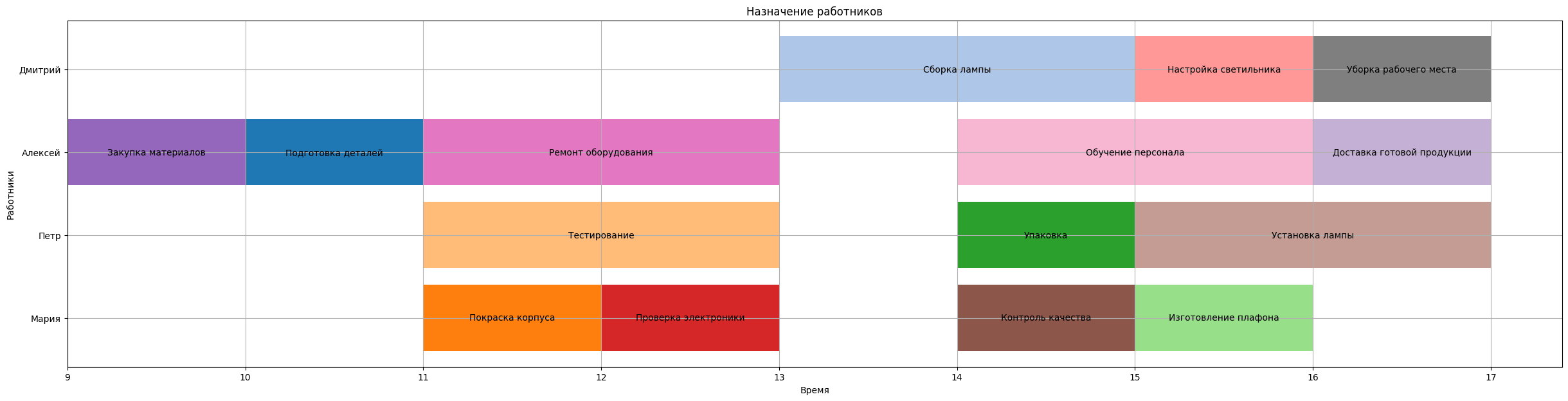


Рисунок 3 ‒ Лучшее назначение работников на рабочий день (островная модель)

Также были построены графики для более наглядного сравнения результатов работы базовой и островной моделей (Рисунок 4).

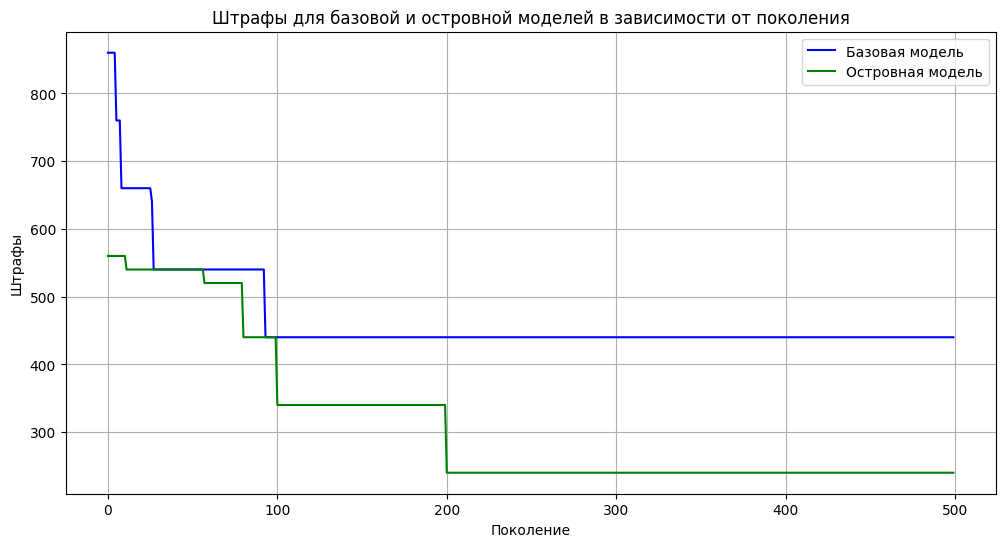


Рисунок 4 ‒ График штрафов для базовой и островной моделей

При анализе графика можно заметить, что в то время, как базовая модель показывает «усредненный» результат ‒ островная модель позволяет значительно сократить штрафы при решении задачи.

Также были выведены отдельные показатели для каждого из островов (Рисунок 5).

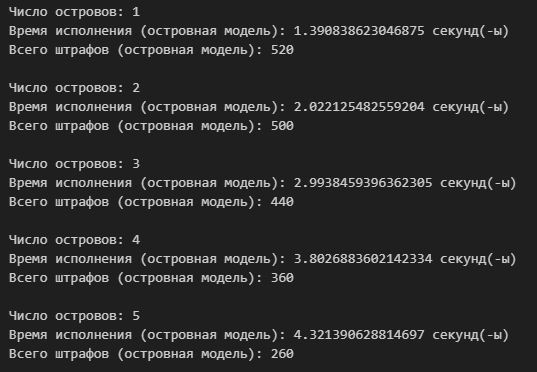


Рисунок 5 ‒ Метрики времени и штрафов для каждого острова

Более наглядно динамику количества островов и количества штрафов можно увидеть на следующих графике (Рисунок 6).



Рисунок 6 ‒ График штрафов островных моделей

При рассмотрении графиков и логов можно заметить, что с большим числом островов, используемых при вычислениях, растёт сходимость к нулю и равномерное возрастание времени требуемого на выполнение кода в зависимости от количества островов.

# Заключение

Во время выполнения данной работы была детально рассмотрена задача о назначении и её решение с применением генетического алгоритма. Реализация осуществлялась с помощью языка программирования Python. При реализации генетического алгоритма была использована базовая островная модель, которая позволяет получать более точные результаты за счёт обмена лучшими решениями между островами. Это особенно полезно для задач с большим количеством ограничений и сложной структурой зависимостей.

Были заранее заданы входные данные, включающие список работ с их временными рамками, длительностью и зависимостями, а также список работников с их навыками и ограничениями по рабочему времени. Эти данные были обработаны с помощью базовой модели (с одним островом) и островной модели (с пятью островами). Результатом работы алгоритмов стали скомпонованные назначения с учётом штрафов, которые являются метрикой «правильности» распределения работ по времени и работникам.

Следовательно, для сложных задач с различными комбинациями параметров наиболее эффективно использовать генетический алгоритм с островной моделью. Этот подход обеспечивает более высокую точность и стабильность в нахождении оптимального назначения.

# Список источников

1. Тарасов И.Е. Системы синхронизированного планирования ресурсов предприятия [Электронный ресурс]: учебное пособие / И.Е. Тарасов. – М.: МИРЭА – Российский технологический университет, 2021. – 96 с. (Дата обращения: 22.02.2025)
2. ГОСТ Р ИСО/МЭК 15288-2005. Информационная технология. Системная инженерия. Процессы жизненного цикла систем. (Дата обращения: 22.02.2025)
3. ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств. (Дата обращения: 22.02.2025)
4. ГОСТ 19.201-78 Техническое задание. Требования к содержанию и оформлению. (Дата обращения: 22.02.2025)
5. ГОСТ 34.602-89 Техническое задание на создание автоматизированной системы. (Дата обращения: 22.02.2025)
6. Генетические алгоритмы. Лекция [Электронный ресурс]: https://sdo.nsuem.ru/pluginfile.php/310628/mod\_resource/content/1/ЛЕКЦИЯ%20генетические%20алгоритмы\_с.pdf (Дата обращения: 15.03.2025)
7. Genetic Algorithms [Электронный ресурс]: https://www.geeksforgeeks.org/genetic-algorithms/ (дата обращения 15.03.2025)
8. Виды генетических алгоритмов [Электронный ресурс]: https://studfile.net/preview/7192285/page:15/ (Дата обращения: 15.03.2025)
9. Исследование метрик оценки кода при формировании наборов данных с использованием генетического алгоритма [Электронный ресурс]: https://cyberleninka.ru/article/n/issledovanie-metrik-otsenki-koda-pri-formirovanii-naborov-dannyh-s-ispolzovaniem-geneticheskogo-algoritma/viewer (Дата обращения: 22.02.2025)

# Приложение А

import numpy as np

import time

import random

from scipy.stats import wilcoxon

import matplotlib.pyplot as plt

# Работы

works = [

    {"name": "Подготовка деталей", "start\_time": 10, "end\_time": 15, "duration": 1, "dependencies": []},

    {"name": "Сборка лампы", "start\_time": 10, "end\_time": 15, "duration": 2, "dependencies": ["Покраска корпуса"]},

    {"name": "Покраска корпуса", "start\_time": 10, "end\_time": 15, "duration": 1, "dependencies": ["Подготовка деталей"]},

    {"name": "Тестирование", "start\_time": 10, "end\_time": 16, "duration": 2, "dependencies": []},

    {"name": "Упаковка", "start\_time": 14, "end\_time": 17, "duration": 1, "dependencies": ["Тестирование"]},

    {"name": "Изготовление плафона", "start\_time": 14, "end\_time": 17, "duration": 1, "dependencies": ["Сборка лампы"]},

    {"name": "Проверка электроники", "start\_time": 12, "end\_time": 15, "duration": 1, "dependencies": []},

    {"name": "Настройка светильника", "start\_time": 14, "end\_time": 16, "duration": 1, "dependencies": ["Сборка лампы"]},

    {"name": "Закупка материалов", "start\_time": 9, "end\_time": 13, "duration": 1, "dependencies": []},

    {"name": "Доставка готовой продукции", "start\_time": 15, "end\_time": 17, "duration": 1, "dependencies": ["Упаковка"]},

    {"name": "Контроль качества", "start\_time": 14, "end\_time": 17, "duration": 1, "dependencies": ["Тестирование"]},

    {"name": "Установка лампы", "start\_time": 14, "end\_time": 17, "duration": 2, "dependencies": ["Сборка лампы"]},

    {"name": "Ремонт оборудования", "start\_time": 10, "end\_time": 14, "duration": 2, "dependencies": ["Подготовка деталей"]},

    {"name": "Обучение персонала", "start\_time": 12, "end\_time": 17, "duration": 2, "dependencies": []},

    {"name": "Уборка рабочего места", "start\_time": 16, "end\_time": 17, "duration": 1, "dependencies": []},

]

# Работник

workers = [

    {"name": "Мария", "skills": ["Покраска корпуса", "Тестирование", "Проверка электроники", "Контроль качества"], "max\_hours": 8},

    {"name": "Петр", "skills": ["Упаковка", "Тестирование", "Настройка светильника", "Установка лампы"], "max\_hours": 8},

    {"name": "Алексей", "skills": ["Закупка материалов", "Подготовка деталей", "Доставка готовой продукции", "Обучение персонала"], "max\_hours": 8},

    {"name": "Дмитрий", "skills": ["Сборка лампы", "Проверка электроники", "Ремонт оборудования", "Обучение персонала", "Настройка светильника"], "max\_hours": 8},

]

# Штрафы

PENALTY\_ORDER\_VIOLATION = 100  # Нарушение порядка выполнения

PENALTY\_LOW\_QUALITY = 80       # Назначение на работу, которую работник не может выполнить качественно

PENALTY\_OVERTIME = 100         # Рабочий день более 8 часов

PENALTY\_SHORT\_LUNCH = 100      # Обеденный перерыв менее часа и не в 12-14

PENALTY\_REPEATED\_WORK = 80     # Штраф за повторное назначение одной и той же работы одному сотруднику

PENALTY\_OVERLAP = 100          # Штраф за одновременное выполнение нескольких работ одним работником

PENALTY\_UNASSIGNED\_WORK = 1000 # Штраф за невыполненную работу

PENALTY\_INCOMPLETE\_WORK = 1000  # Штраф за неполное выполнение задачи

PENALTY\_TIME\_VIOLATION = 1000  # Штраф за нарушение временного интервала

# Функция для создания начальной популяции

def create\_schedule():

    schedule = [[] for \_ in workers]  # Назначение для каждого работника

    assigned\_works = set()  # Множество для отслеживания назначенных работ

    for work in works:

        # Выбираем случайного работника, который может выполнить задачу

        possible\_workers = [worker\_idx for worker\_idx, worker in enumerate(workers) if work["name"] in worker["skills"]]

        if not possible\_workers:

            # Если никто не может выполнить задачу, выбираем любого работника

            possible\_workers = range(len(workers))

        worker\_idx = random.choice(possible\_workers)

        start\_time = random.randint(work["start\_time"], work["end\_time"] - work["duration"])

        schedule[worker\_idx].append((work["name"], start\_time, start\_time + work["duration"]))

        assigned\_works.add(work["name"])

    return schedule

def evaluate\_schedule(schedule):

    penalty = 0

    penalty\_details = []  # Список для хранения информации о штрафах

    # Создаем словарь для хранения времени выполнения задач

    task\_times = {}

    for worker\_schedule in schedule:

        for task in worker\_schedule:

            task\_name, start\_time, end\_time = task

            task\_times[task\_name] = (start\_time, end\_time)

    # Проверяем порядок выполнения задач

    for work in works:

        for dependency in work["dependencies"]:

            if dependency in task\_times and work["name"] in task\_times:

                dependency\_start, dependency\_end = task\_times[dependency]

                work\_start, work\_end = task\_times[work["name"]]

                # Проверяем, выполняется ли задача до завершения зависимости

                if work\_start < dependency\_end:

                    penalty += PENALTY\_ORDER\_VIOLATION

                    penalty\_details.append(f"Нарушение порядка выполнения: {work['name']} начинается до завершения {dependency} (штраф: {PENALTY\_ORDER\_VIOLATION})")

    # Проверка качества выполнения работ

    for worker\_schedule in schedule:

        worker = workers[schedule.index(worker\_schedule)]

        for task in worker\_schedule:

            if task[0] not in worker["skills"]:

                penalty += PENALTY\_LOW\_QUALITY

                penalty\_details.append(f"Низкое качество выполнения: {task[0]} назначена на {worker['name']}, который не имеет навыков для этой работы (штраф: {PENALTY\_LOW\_QUALITY})")

    # Проверка рабочего времени и обеденного перерыва

    for worker\_schedule in schedule:

        total\_hours = sum(task[2] - task[1] for task in worker\_schedule)

        if total\_hours > 8:

            penalty += PENALTY\_OVERTIME

            penalty\_details.append(f"Переработка: {workers[schedule.index(worker\_schedule)]['name']} работает более 8 часов (штраф: {PENALTY\_OVERTIME})")

        # Проверка обеденного перерыва

        lunch\_found = False

        # Проверяем интервал 12:00 - 13:00

        slot\_free = True

        for task in worker\_schedule:

            task\_start, task\_end = task[1], task[2]

            if not (task\_end <= 12 or task\_start >= 13):  # Задача пересекается с 12:00 - 13:00

                slot\_free = False

                break

        if slot\_free:

            lunch\_found = True

        # Проверяем интервал 13:00 - 14:00

        if not lunch\_found:

            slot\_free = True

            for task in worker\_schedule:

                task\_start, task\_end = task[1], task[2]

                if not (task\_end <= 13 or task\_start >= 14):  # Задача пересекается с 13:00 - 14:00

                    slot\_free = False

                    break

            if slot\_free:

                lunch\_found = True

        # Если обеденный перерыв не найден, начисляем штраф

        if not lunch\_found:

            penalty += PENALTY\_SHORT\_LUNCH

            penalty\_details.append(f"Отсутствие обеденного перерыва: {workers[schedule.index(worker\_schedule)]['name']} не имеет обеда (штраф: {PENALTY\_SHORT\_LUNCH})")

    # Проверка повторного назначения одной и той же работы одному сотруднику

    for worker\_schedule in schedule:

        tasks = [task[0] for task in worker\_schedule]

        if len(tasks) != len(set(tasks)):

            penalty += PENALTY\_REPEATED\_WORK

            penalty\_details.append(f"Повторное назначение работы: {workers[schedule.index(worker\_schedule)]['name']} назначен на одну и ту же работу несколько раз (штраф: {PENALTY\_REPEATED\_WORK})")

    # Проверка одновременного выполнения нескольких работ одним работником

    for worker\_schedule in schedule:

        for i in range(len(worker\_schedule)):

            for j in range(i + 1, len(worker\_schedule)):

                task1 = worker\_schedule[i]

                task2 = worker\_schedule[j]

                if task1[1] < task2[2] and task2[1] < task1[2]:

                    penalty += PENALTY\_OVERLAP

                    penalty\_details.append(f"Перекрытие работ: {workers[schedule.index(worker\_schedule)]['name']} выполняет {task1[0]} и {task2[0]} одновременно (штраф: {PENALTY\_OVERLAP})")

    # Проверка, что все работы выполнены

    assigned\_works = set()

    for worker\_schedule in schedule:

        for task in worker\_schedule:

            assigned\_works.add(task[0])

    for work in works:

        if work["name"] not in assigned\_works:

            penalty += PENALTY\_UNASSIGNED\_WORK

            penalty\_details.append(f"Невыполненная работа: {work['name']} не назначена ни одному работнику (штраф: {PENALTY\_UNASSIGNED\_WORK})")

    # Проверка, что задачи выполнены полностью (продолжительность равна duration)

    for worker\_schedule in schedule:

        for task in worker\_schedule:

            task\_name, start\_time, end\_time = task

            work = next(w for w in works if w["name"] == task\_name)

            if (end\_time - start\_time) != work["duration"]:

                penalty += PENALTY\_INCOMPLETE\_WORK

                penalty\_details.append(f"Неполное выполнение задачи: {task\_name} выполнена за {end\_time - start\_time} часа(ов) вместо {work['duration']} (штраф: {PENALTY\_INCOMPLETE\_WORK})")

    # Проверка, что задачи выполняются в пределах start\_time и end\_time

    for worker\_schedule in schedule:

        for task in worker\_schedule:

            task\_name, start\_time, end\_time = task

            work = next(w for w in works if w["name"] == task\_name)

            if start\_time < work["start\_time"] or end\_time > work["end\_time"]:

                penalty += PENALTY\_TIME\_VIOLATION

                penalty\_details.append(f"Нарушение временного интервала: {task\_name} выполняется вне допустимого времени ({work['start\_time']}:00 - {work['end\_time']}:00) (штраф: {PENALTY\_TIME\_VIOLATION})")

    return penalty, penalty\_details

# Функция мутации

def mutate\_schedule(schedule):

    mutated\_schedule = [worker\_schedule.copy() for worker\_schedule in schedule]

    # Выбираем случайного работника

    worker\_idx = random.randint(0, len(mutated\_schedule) - 1)

    worker\_schedule = mutated\_schedule[worker\_idx]

    if len(worker\_schedule) > 0:

        # Выбираем случайную задачу у этого работника

        task\_idx = random.randint(0, len(worker\_schedule) - 1)

        task = worker\_schedule[task\_idx]

        task\_name, start\_time, end\_time = task

        # Варианты мутации:

        if random.random() < 0.5:

            # Мутация 1: Изменяем время выполнения задачи

            new\_start\_time = random.randint(works[task\_idx]["start\_time"], works[task\_idx]["end\_time"] - works[task\_idx]["duration"])

            worker\_schedule[task\_idx] = (task\_name, new\_start\_time, new\_start\_time + works[task\_idx]["duration"])

        else:

            # Мутация 2: Перемещаем задачу другому работнику

            new\_worker\_idx = random.randint(0, len(mutated\_schedule) - 1)

            if new\_worker\_idx != worker\_idx:

                mutated\_schedule[worker\_idx].pop(task\_idx)

                mutated\_schedule[new\_worker\_idx].append((task\_name, start\_time, end\_time))

    return mutated\_schedule

# Функция кроссовера

def crossover(parent1, parent2):

    child1 = []

    child2 = []

    for i in range(len(parent1)):

        if random.random() < 0.5:

            child1.append(parent1[i])

            child2.append(parent2[i])

        else:

            child1.append(parent2[i])

            child2.append(parent1[i])

    return child1, child2

# Функция отбора лучших назначений

def select\_best\_schedules(population, num\_best=5):

    evaluated\_population = [(schedule, evaluate\_schedule(schedule)[0]) for schedule in population]

    evaluated\_population.sort(key=lambda x: x[1])

    return [schedule for schedule, \_ in evaluated\_population[:num\_best]]

# Основной алгоритм

def genetic\_algorithm(num\_generations=500, population\_size=20):

    population = [create\_schedule() for \_ in range(population\_size)]

    penalties = []

    for generation in range(num\_generations):

        # Оценка популяции

        evaluated\_population = [(schedule, evaluate\_schedule(schedule)[0]) for schedule in population]

        evaluated\_population.sort(key=lambda x: x[1])

        # Отбор лучших назначений

        best\_schedules = [schedule for schedule, \_ in evaluated\_population[:population\_size // 2]]

        # Создание нового поколения

        new\_population = best\_schedules.copy()

        while len(new\_population) < population\_size:

            # Выбор двух родителей

            parent1, parent2 = random.choices(best\_schedules, k=2)

            # Скрещивание (кроссовер)

            child1, child2 = crossover(parent1, parent2)

            # Мутация

            if random.random() < 0.1:  # Вероятность мутации 10%

                child1 = mutate\_schedule(child1)

            if random.random() < 0.1:

                child2 = mutate\_schedule(child2)

            new\_population.extend([child1, child2])

        population = new\_population

        penalties.append(evaluated\_population[0][1])

    return population[0], penalties

# Островная модель

def island\_model(num\_generations=500, population\_size=40, num\_islands=5, migration\_rate=20):

    islands = [[create\_schedule() for \_ in range(population\_size)] for \_ in range(num\_islands)]

    penalties = []

    for generation in range(num\_generations):

        for i in range(num\_islands):

            # Оценка и отбор на каждом острове

            evaluated\_population = [(schedule, evaluate\_schedule(schedule)[0]) for schedule in islands[i]]

            evaluated\_population.sort(key=lambda x: x[1])

            best\_schedules = [schedule for schedule, \_ in evaluated\_population[:population\_size // 2]]

            # Создание нового поколения на острове

            new\_population = best\_schedules.copy()

            while len(new\_population) < population\_size:

                parent1, parent2 = random.choices(best\_schedules, k=2)

                child1, child2 = crossover(parent1, parent2)

                if random.random() < 0.1:

                    child1 = mutate\_schedule(child1)

                if random.random() < 0.1:

                    child2 = mutate\_schedule(child2)

                new\_population.extend([child1, child2])

            islands[i] = new\_population

        # Миграция между островами

        if generation % migration\_rate == 0:

            best\_solutions = [select\_best\_schedules(island, num\_best=1)[0] for island in islands]

            for i in range(num\_islands):

                next\_island = (i + 1) % num\_islands

                islands[next\_island].append(best\_solutions[i])

                if len(islands[next\_island]) > population\_size:

                    islands[next\_island] = select\_best\_schedules(islands[next\_island], num\_best=population\_size)

        penalties.append(evaluate\_schedule(select\_best\_schedules(islands[0], num\_best=1)[0])[0])

    best\_schedules = [select\_best\_schedules(island, num\_best=1)[0] for island in islands]

    best\_schedule = min(best\_schedules, key=lambda s: evaluate\_schedule(s)[0])

    return best\_schedule, penalties, best\_schedules

# Визуализация назначения

def visualize\_schedule(schedule):

    fig, ax = plt.subplots(figsize=(30, 7))

    colors = plt.cm.tab20.colors  # Цвета для задач

    # Создаем словарь для хранения цветов задач

    task\_colors = {work["name"]: colors[i % len(colors)] for i, work in enumerate(works)}

    for i, worker\_schedule in enumerate(schedule):

        for task in worker\_schedule:

            # Используем duration из works, а не из task

            work = next(w for w in works if w["name"] == task[0])

            duration = work["duration"]

            # Позиция по Y: i (целое число, соответствующее индексу работника)

            ax.barh(i, duration, left=task[1], color=task\_colors[task[0]], height=0.8)  # height регулирует высоту столбца

            # Текст задачи по центру столбца

            ax.text((task[1] + task[1] + duration) / 2, i, task[0], ha='center', va='center', color='black')

    # Настройка оси Y

    ax.set\_yticks(range(len(workers)))  # Метки по оси Y соответствуют индексам работников

    ax.set\_yticklabels([worker["name"] for worker in workers])  # Подписи — имена работников

    ax.set\_xlabel("Время")

    ax.set\_ylabel("Работники")

    ax.set\_title("Назначение работников")

    plt.grid(True)

    plt.show()

# Функция для текстового вывода назначения

def print\_schedule(schedule):

    for i, worker\_schedule in enumerate(schedule):

        worker = workers[i]

        print(f"Работник: {worker['name']}")

        for task in worker\_schedule:

            task\_name, start\_time, end\_time = task

            print(f"  - Задача: {task\_name}, Время: {start\_time}:00 - {end\_time}:00")

        print()  # Пустая строка для разделения работников

# Сравнение моделей

def run\_experiments():

    # Базовая модель

    start\_time = time.time()

    best\_schedule\_base, penalties\_base = genetic\_algorithm()

    end\_time = time.time()

    execution\_time\_base = end\_time - start\_time

    total\_penalty\_base, penalty\_details\_base = evaluate\_schedule(best\_schedule\_base)

    # Островная модель

    start\_time = time.time()

    best\_schedule\_island, penalties\_island, best\_schedules\_island = island\_model()

    end\_time = time.time()

    execution\_time\_island = end\_time - start\_time

    total\_penalty\_island, penalty\_details\_island = evaluate\_schedule(best\_schedule\_island)

    # Wilcoxon signed-rank test

    statistic, p\_value = wilcoxon(penalties\_base, penalties\_island)

    # Вывод результатов

    print(f"Время исполнения (базовая модель): {execution\_time\_base} секунд(-ы)")

    print(f"Всего штрафов (базовая модель): {total\_penalty\_base}")

    print("Штрафы (базовая модель):")

    for detail in penalty\_details\_base:

        print(detail)

    print()

    print(f"Время исполнения (островная модель): {execution\_time\_island} секунд(-ы)")

    print(f"Всего штрафов (островная модель): {total\_penalty\_island}")

    print("Штрафы (островная модель):")

    for detail in penalty\_details\_island:

        print(detail)

    print()

    print(f"Статистика Wilcoxon: {statistic}")

    print(f"P-value: {p\_value}")

    if p\_value < 0.05:

        print("Отвергаем нулевую гипотезу, результаты моделей статистически значимо различаются.")

    else:

        print("Не можем отвергнуть нулевую гипотезу, результаты моделей не статистически значимо различаются.")

    print()

    # Вывод лучшего назначения в текстовом формате

    print("Лучшее назначение (базовая модель):")

    print\_schedule(best\_schedule\_base)

    print("Лучшее назначение (островная модель):")

    print\_schedule(best\_schedule\_island)

    # Визуализация назначения

    print("Лучшее назначение (базовая модель):")

    visualize\_schedule(best\_schedule\_base)

    print("Лучшее назначение (островная модель):")

    visualize\_schedule(best\_schedule\_island)

    # График штрафов для базовой и островной моделей

    plt.figure(figsize=(12, 6))

    plt.plot(penalties\_base, label="Базовая модель", color="blue")

    plt.plot(penalties\_island, label="Островная модель", color="green")

    plt.xlabel("Поколение")

    plt.ylabel("Штрафы")

    plt.title("Штрафы для базовой и островной моделей в зависимости от поколения")

    plt.legend()

    plt.grid(True)

    plt.show()

# График: Штрафы для островной модели в зависимости от числа островов и поколения

def plot\_island\_penalties():

    num\_islands\_list = [1, 2, 3, 4, 5]

    num\_generations = 500

    plt.figure(figsize=(12, 6))

    for num\_islands in num\_islands\_list:

        start\_time = time.time()

        \_, penalties, \_ = island\_model(num\_generations=num\_generations, num\_islands=num\_islands)

        end\_time = time.time()

        execution\_time = end\_time - start\_time

        total\_penalty = penalties[-1]

        print(f"Число островов: {num\_islands}")

        print(f"Время исполнения (островная модель): {execution\_time} секунд(-ы)")

        print(f"Всего штрафов (островная модель): {total\_penalty}")

        print()

        plt.plot(penalties, label=f"{num\_islands} островов")

    plt.xlabel("Поколение")

    plt.ylabel("Штрафы")

    plt.title("Штрафы для островной модели в зависимости от числа островов и поколения")

    plt.legend()

    plt.grid(True)

    plt.show()

# Запуск экспериментов

run\_experiments()

plot\_island\_penalties()