МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ

КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

**«Московский Технический Университет Связи И Информатики»**

Кафедра «Математическая кибернетика и информационные технологии»

**Курсовая работа**

по дисциплине

**«Структуры и алгоритмы обработки данных»**

Выполнил:

студент группы БВТ2201 Шамсутдинов Р. Ф.

Проверил:

преподаватель Симонов С. Е.

Москва, 2024 год

**Оглавление**

[**Введение 3**](#_gjdgxs)

[Цель работы 3](#_30j0zll)

[Задачи 3](#_1fob9te)

[**Ход работы 3**](#_3znysh7)

[Алгоритм в лоб 4](#_2et92p0)

[Генетический алгоритм 16](#_tyjcwt)

[Сравнение алгоритмов 26](#_3dy6vkm)

[**Заключение 26**](#_1t3h5sf)

[Вывод 27](#_4d34og8)

[**Список источников 28**](#_dpk8f8y6k14g)

[**Приложения 29**](#_vi86lxsyhl0o)

[GitHub 29](#_3rdcrjn)

[Код 29](#_9gqz8arnwcma)

## 

## 

## 

## 

# Введение

## Цель работы

Создать два алгоритма (в лоб и генетический) для составления оптимального графика работы водителей и маршрутов автобусов. Необходимо учесть все условия и сравнить подходы между собой.

## Задачи

Исходные данные:

1) Автопарк: количество доступных автобусов.

2) Водители: количество доступных водителей.

3) Целевая функция: обеспечить круглосуточный поток автобусов, задействовав как можно меньше водителей и автобусов.

График работы водителей:

1) 8 часов в сутки, выходные – суббота и воскресенье, без ночных смен, начинают работать с 6:00 до 10:00 утра. Перерыв 1 час на конечной станции.

2) 12 часов в сутки, работа через 2 дня (1 день работает, 2 дня отдыхает), могут работать в любое время суток и любой день недели. Могут брать перерыв 10 минут каждые 2-4 часа на конечной станции.

Остальные условия:

1) Автобусы должны круглосуточно курсировать.

2) Кольцевой маршрут.

3) Пересменка водителей длится 15 минут и происходит на конечной станции.

Выходные данные:

Итоговые таблицы, содержащие графики работы водителей.

# Ход работы

## Алгоритм в лоб

Перед написание кода, был проведен анализ задачи, в ходе которого был получен вывод, о том, что три группы водителей по 4 человека с режимом работы 12 часов будут являться самым оптимальным решением. Каждый день нужно минимум 2 водителя с режимом работы 12 часов, так как только они могут работать ночью, и должны заменять друг друга на перерывах, чтобы обеспечить круглосуточное курсирование автобусов. Также в выходные дни работать могут только 12-часовые водители. Чтобы закрыть половину дня и поддерживать непрерывное курсирование автобусов нужно также 2 водителя с 12-часовым режимом работы, таким образом группа из 4 водителей с режимом работы 12 часов сможет полностью закрыть целые сутки. Взяв три таких группы мы сможем обеспечить круглосуточное курсирование автобусов каждый день. 1-ая группа берет понедельник, четверг и воскресенье, 2-ая: вторник, пятницу и понедельник следующей недели, 3-тья: среду, субботу и вторник следующей недели.

Переходим к реализации алгоритма.

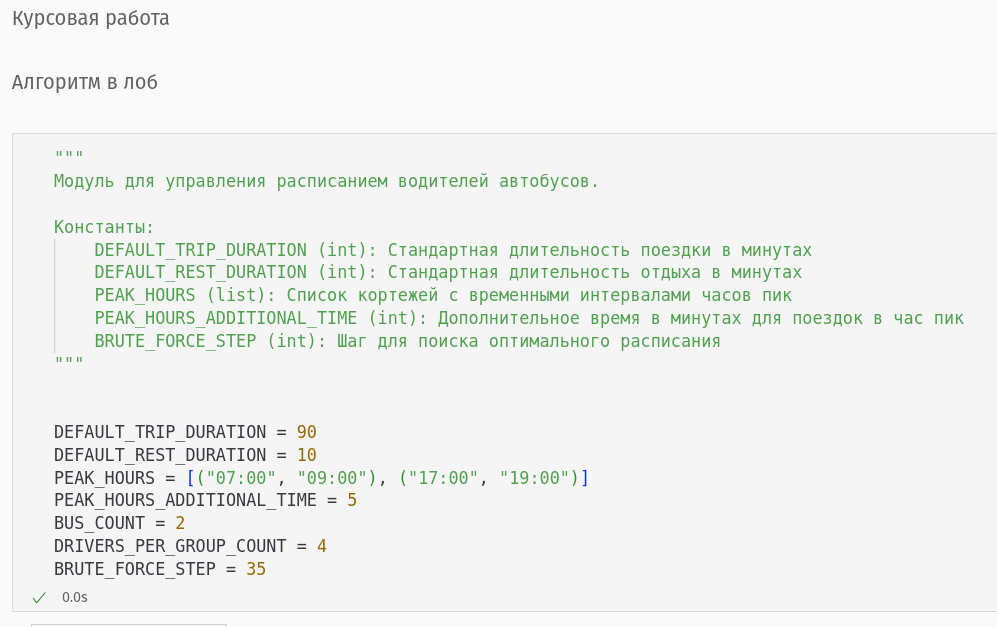


Рисунок 1. Задаем глобальные константы.

Импортируем нужные модули и библиотеки, создадим вспомогательные функции. is\_peak\_hour проверяет попадает ли время в часпик, get\_trip\_duration рассчитывает время поездки в зависимости от часа пик.

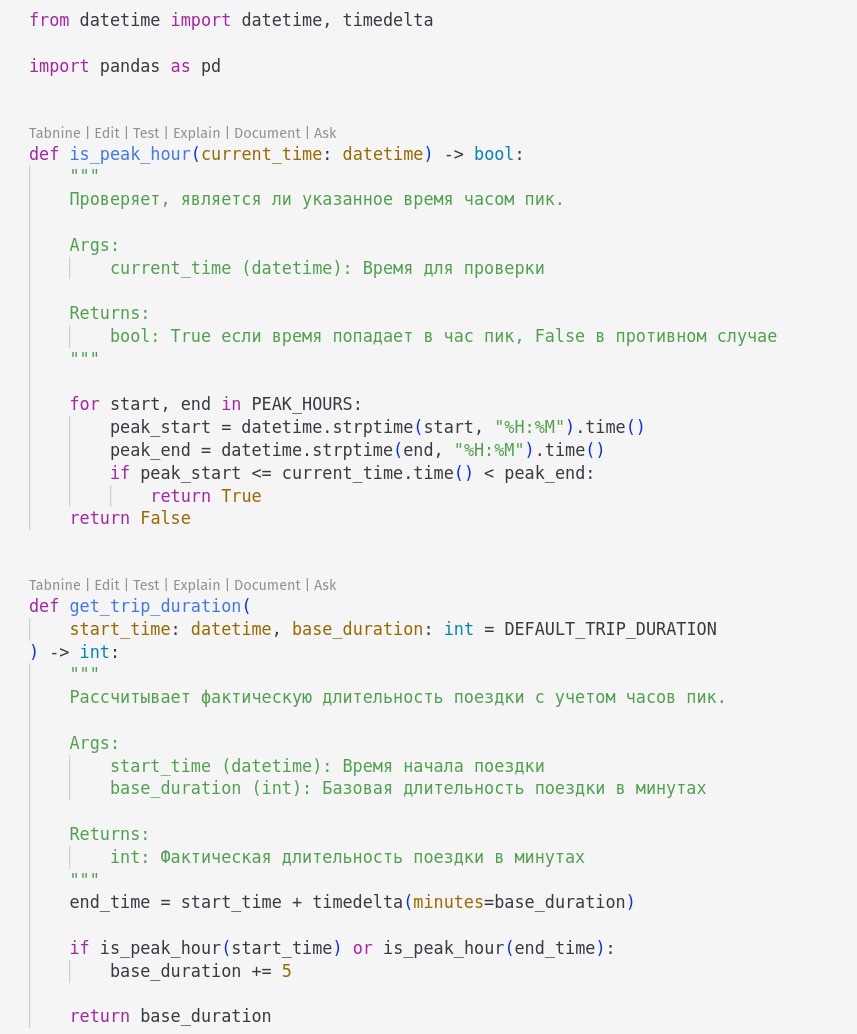


Рисунок 2. Функции is\_peak\_hour и get\_trip\_duration

Создадим вспомогательный класс TimeInterval для представления временного промежутка, добавим вспомогательные методы для удобного вывода интервала и проверки пересечения с другим интервалом.

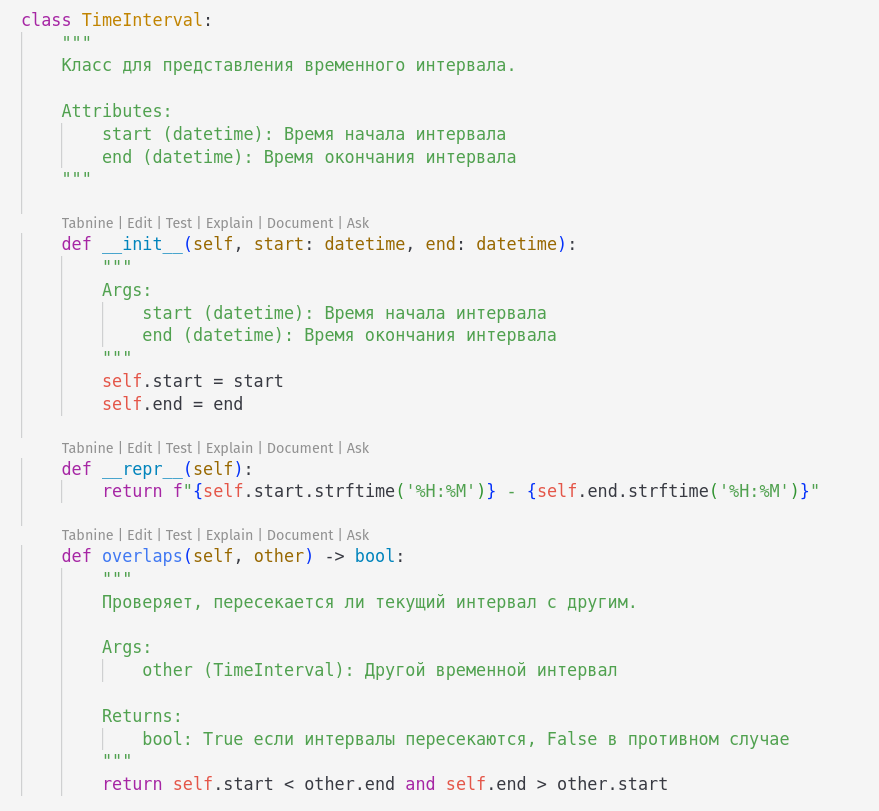


Рисунок 3. Класс TimeInterval

Создадим класс Bus, в котором будем хранить информацию об автобусах

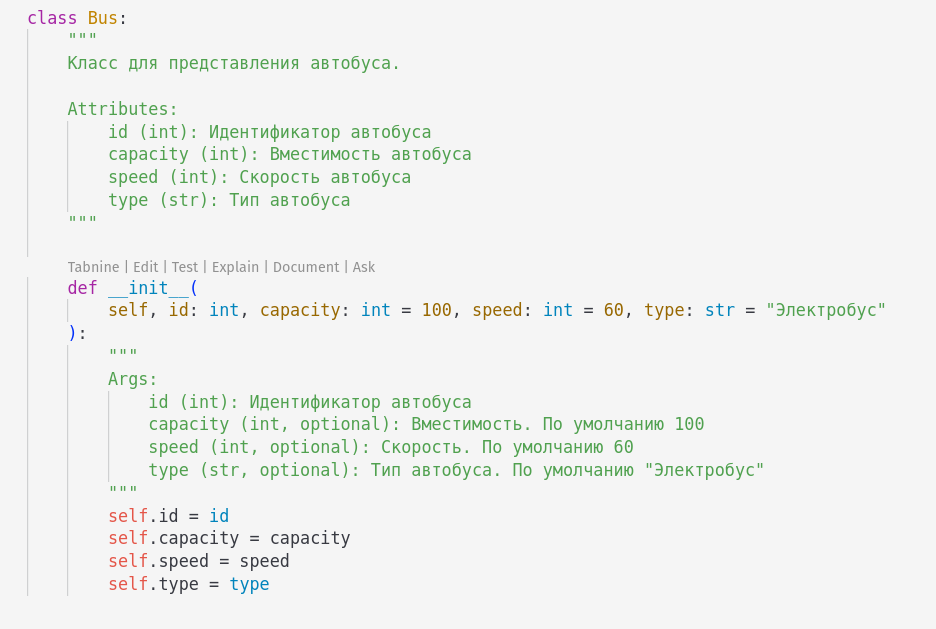


Рисунок 4. Класс Bus

Переходим к описанию самого масштабного класса в данном проекте, класс водителя Driver. Он содержит в себе информацию о водителе, номере автобуса, время работы и отдыха, а также тип работы. Также были добавлены методы, get\_trip\_intevals возвращает список интервалов работы, get\_last\_trip\_inteval возвращает период последней поездки (рис. 5). Метод get\_work\_hours возвращает количество часов работы, в зависимости от типа водителя, get\_rest\_minutes возвращает время, отведенное на перерыв (рис. 6)



Рисунок 5. Класс Driver

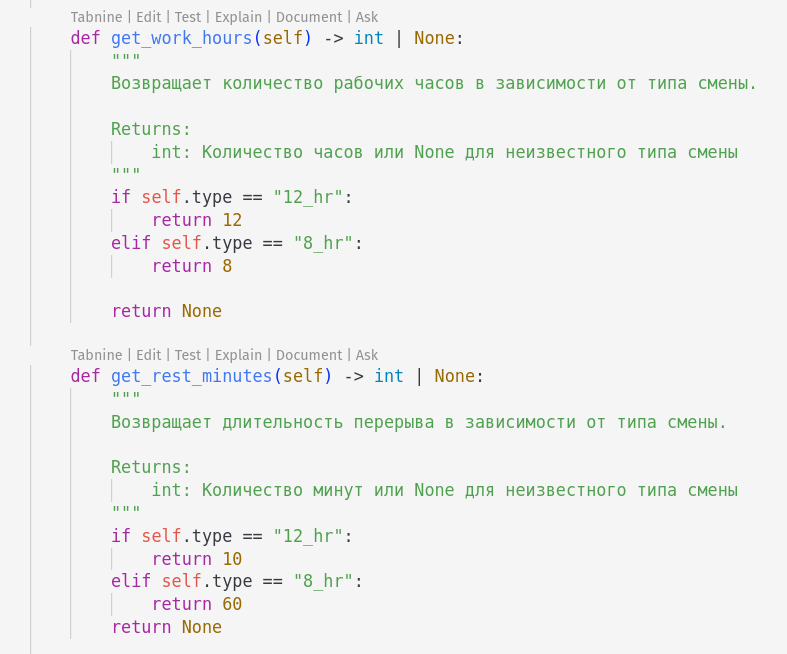


Рисунок 6. Методы класса Driver

Метод generate\_schedule класса Driver генерирует время работы и время отдыха для водителей.

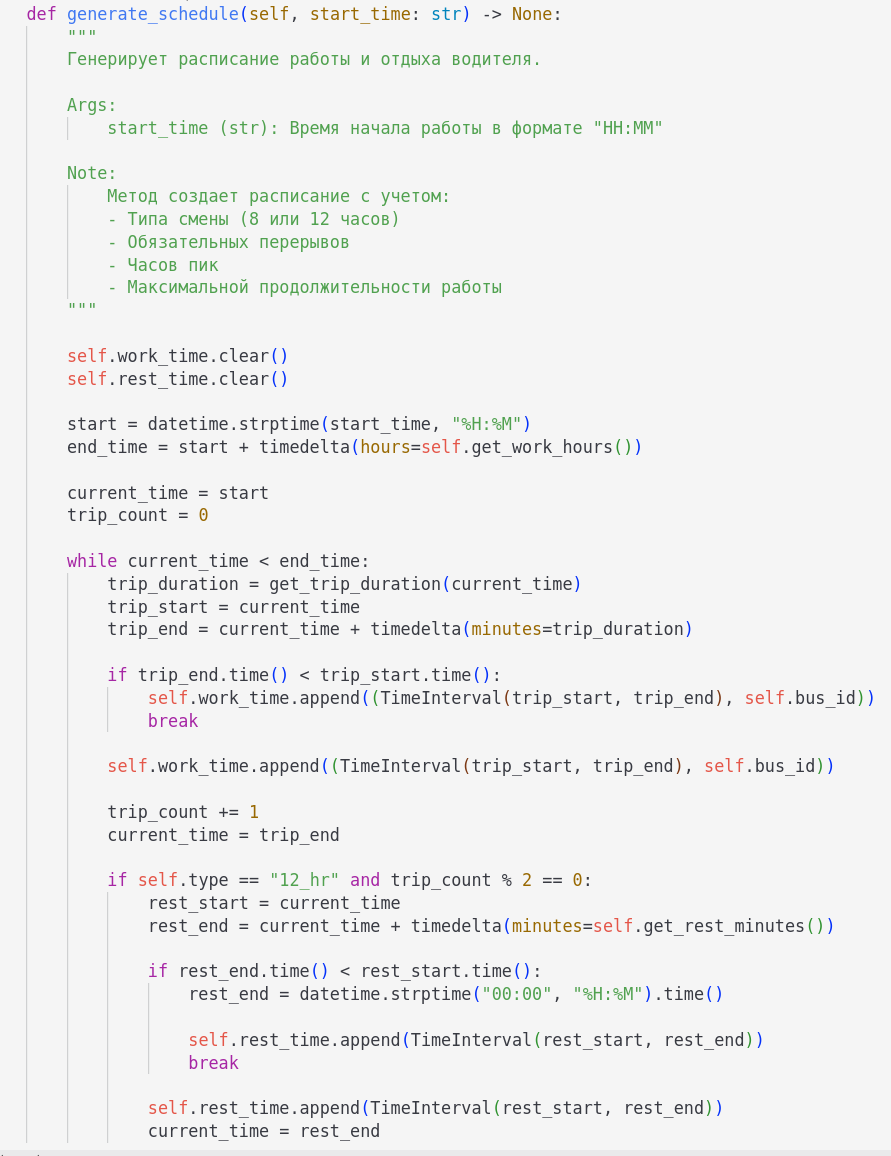
****

Рисунок 7. Метод generate\_schedule класса Driver

Функция get\_all\_trips\_intervals создает список из интервалов работ всех водителей. Функция check\_continuous\_coverage проверяет все интервалы и ищет есть ли такой промежуток, который не покрывает расписание водителей



Рисунок 8. Вспомогательные функции check\_continuous\_coverage и get\_all\_trips\_intervals

Функция find\_valid\_schedule является основной в данном алгоритме, она ищет оптимальное время начала работы для водителей путем перебора всевозможных вариантов. Данный перебор оптимизирован на основе того, что два водителя должны начинать в начале дня, а два в середине, чтобы они смогли удовлетворить условиям задачи, поэтому перебор начинается с 4, а потом и 3 водителя, в то время как 1 всегда начинает в 00:00, а второй +- в 03:00. Также можно задать шаг перебора в константах (рис. 1).

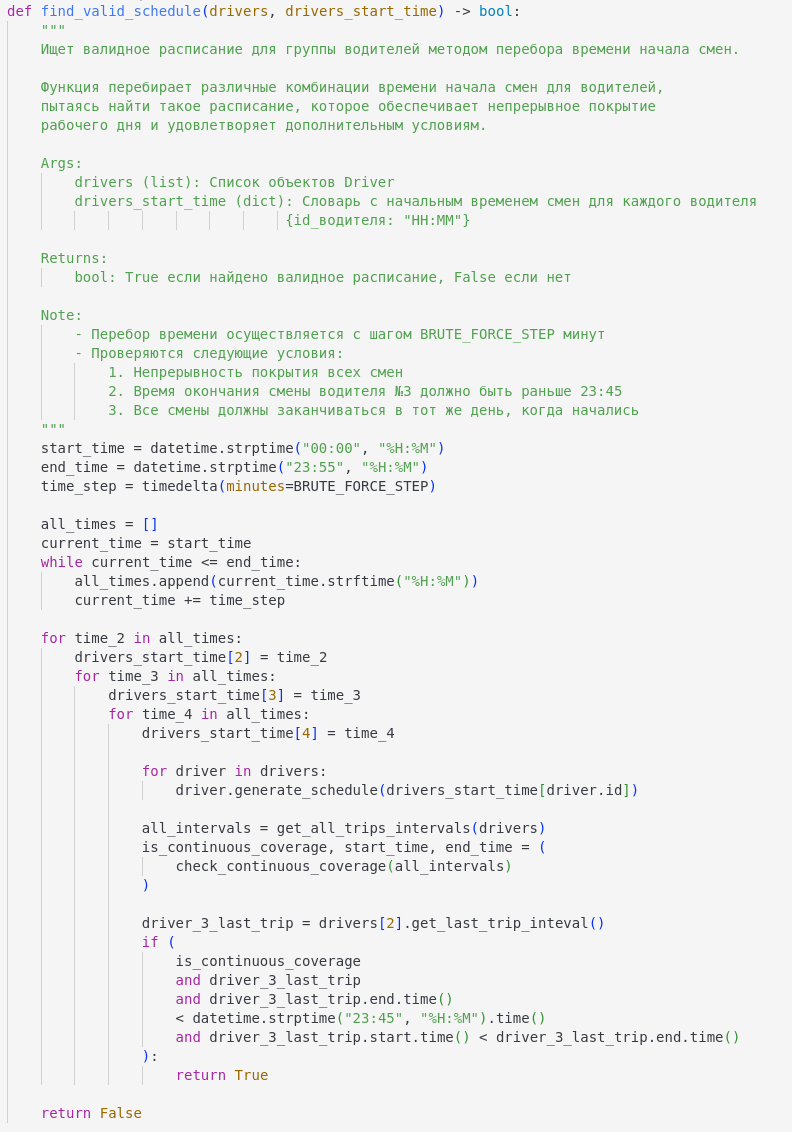


Рисунок 9. Функция find\_valid\_schedule, перебор начала работы.

Инициализация программы и запуск происходит на рисунке 10.

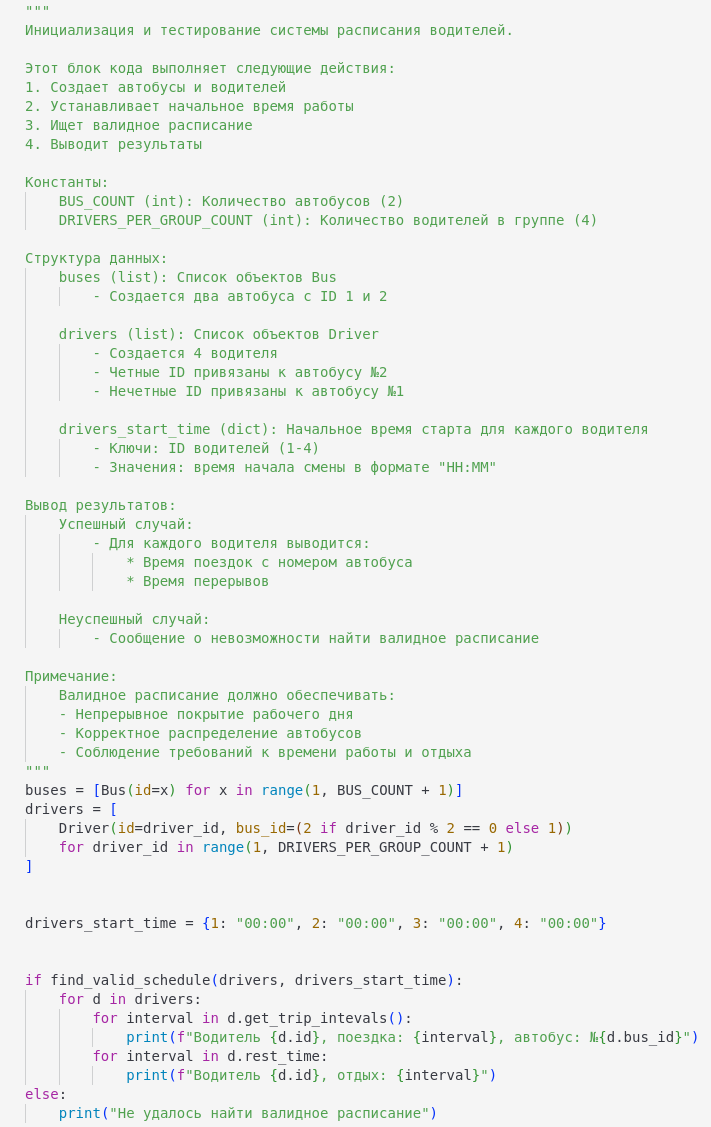


Рисунок 10. Запуск программы

Результат работы программы

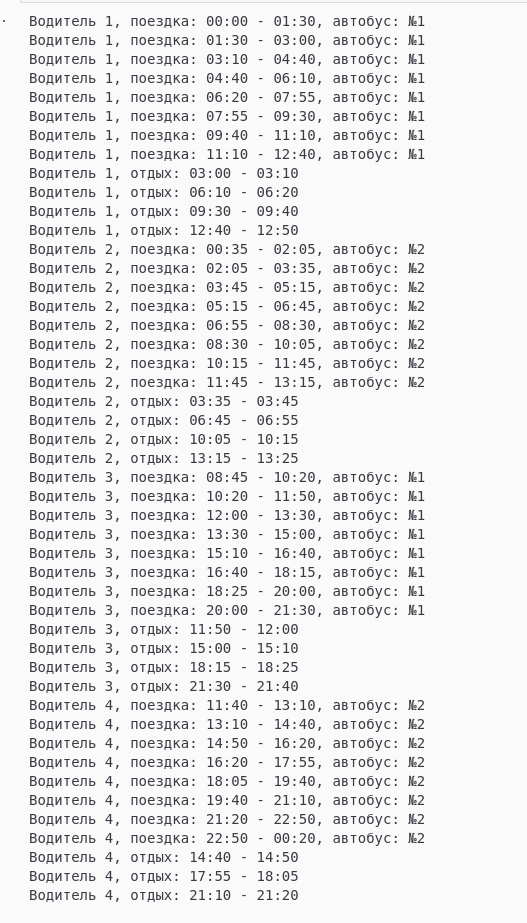


Рисунок 11. Расписание первой группы водителей

Продублируем это расписание на две другие группы и выведем в красивой таблице.

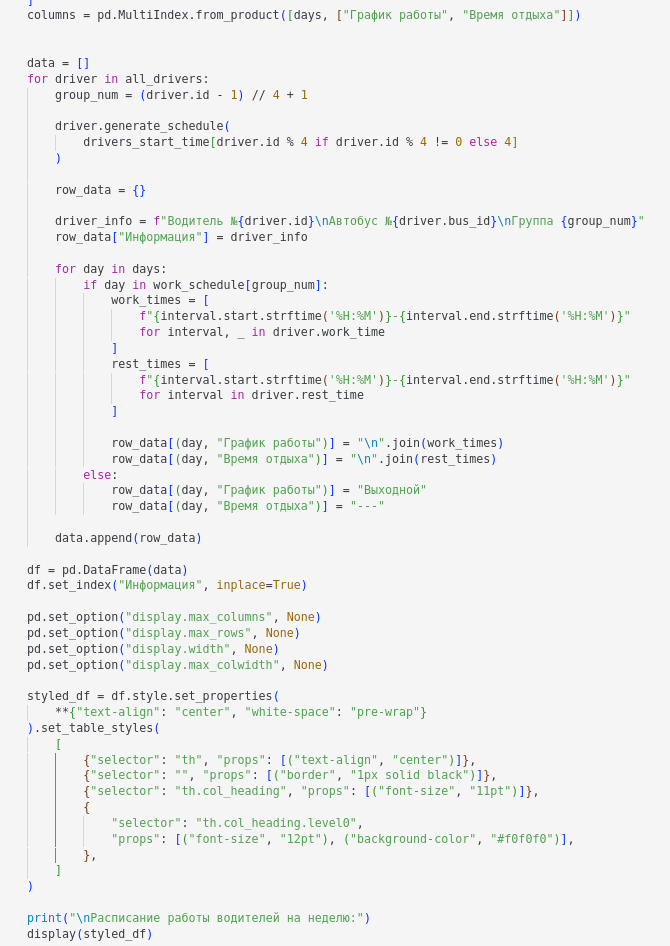


Рисунок 12. Код для таблицы

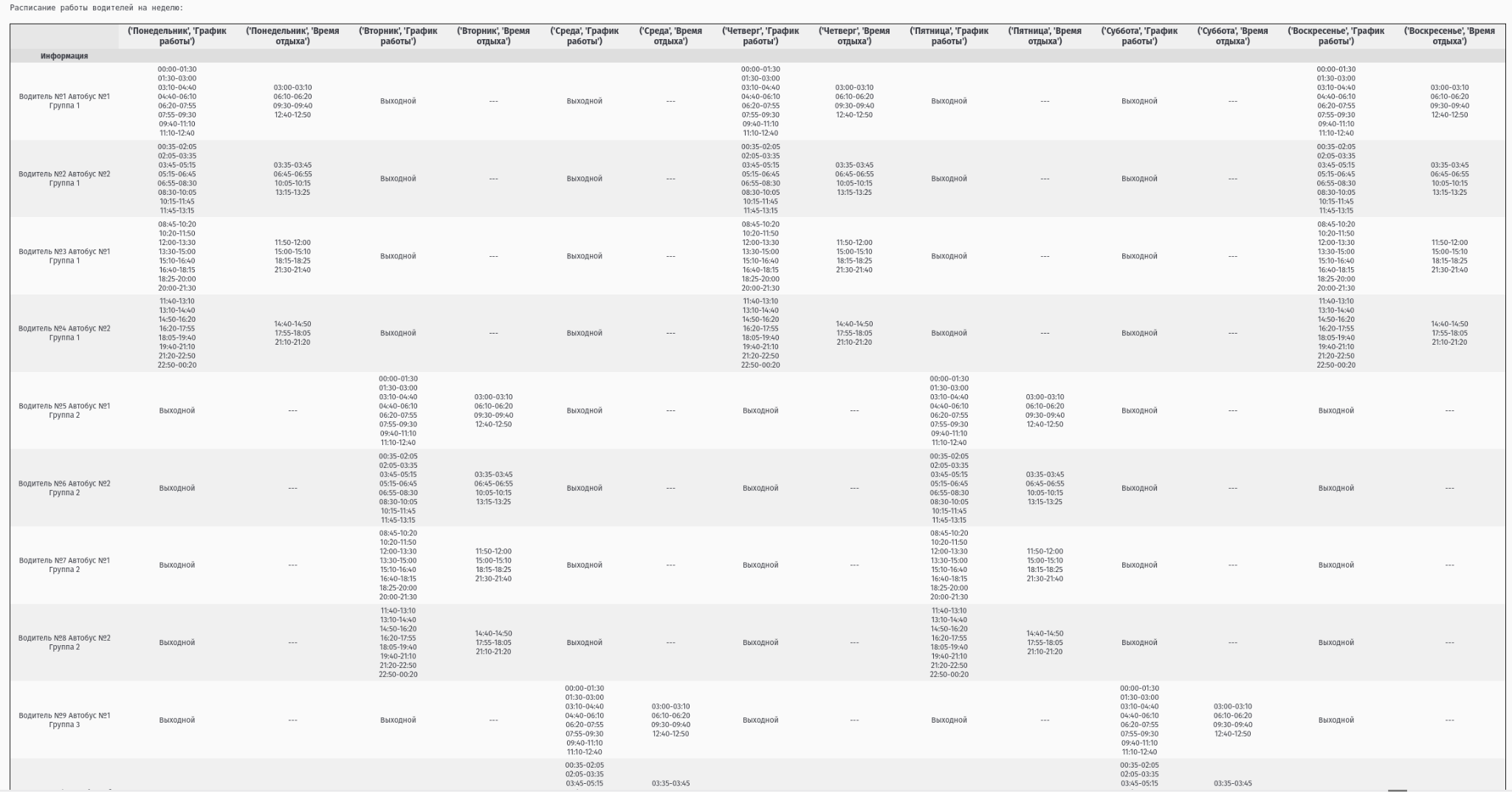


Рисунок 13. Таблица с расписание

Сохраним расписание в эксель файл



Рисунок 14. Сохранение расписания в эксель

## Генетический алгоритм

Генетический алгоритм — это эвристический метод поиска, применяемый для решения задач оптимизации и моделирования. Он основывается на механизмах естественного отбора, таких как наследование, мутации, отбор и скрещивание, имитируя процессы биологической эволюции.

**Основные этапы работы генетического алгоритма:**

1. **Создание начальной популяции:** Генерируется множество потенциальных решений задачи, обычно случайным образом.
2. **Оценка приспособленности:** Каждое решение оценивается с помощью функции приспособленности, которая определяет, насколько хорошо оно решает поставленную задачу.
3. **Отбор (селекция):** Выбираются наиболее приспособленные решения, которые будут использоваться для создания потомков.
4. **Скрещивание (кроссовер):** Избранные решения комбинируются между собой, обмениваясь частями своих данных, чтобы сформировать новые решения.
5. **Мутация:** В новые решения вносятся небольшие случайные изменения, что способствует разнообразию популяции и предотвращает преждевременную сходимость к локальным оптимумам.
6. **Формирование нового поколения:** На основе отобранных и модифицированных решений формируется новое поколение, и процесс повторяется с шага 2.

Алгоритм повторяет эти шаги до достижения заданного критерия остановки, например, определенного числа поколений или удовлетворительного уровня приспособленности.

Напишем небольшую документацию к реализации генетического алгоритма



Рисунок 15. Документация генетического алгоритма

Далее импортируем библиотеки и модули, необходимые для реализации

алгоритма. Также зададим глобальные константы для дальнейшего использования.

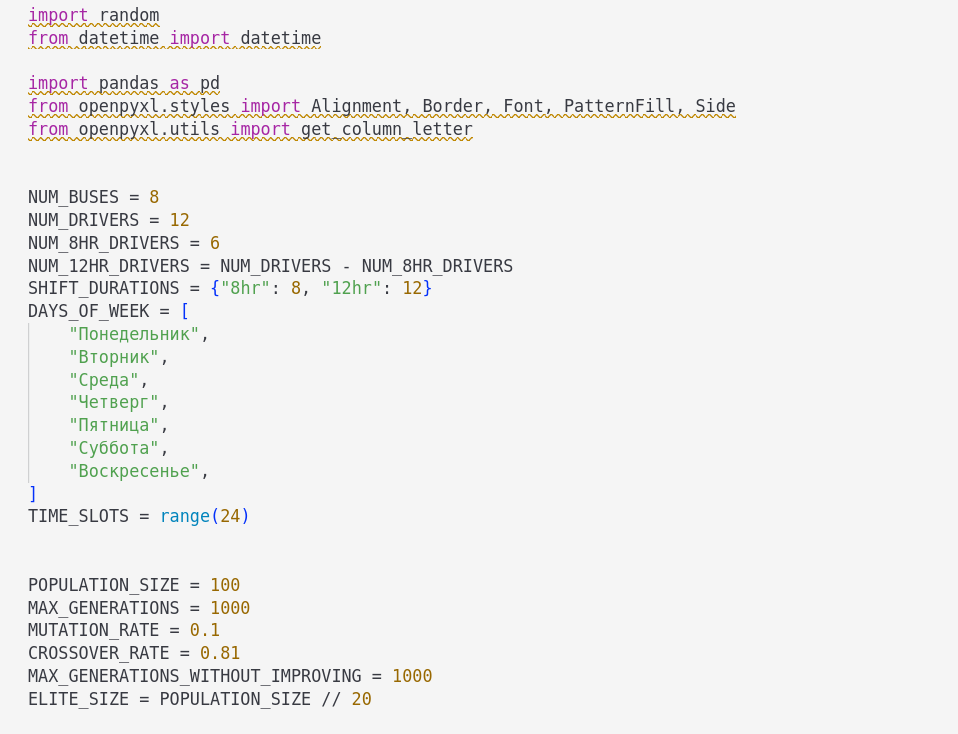


Рисунок 16. Модули и константы

Переходим к созданию вспомогательных функций. Функция is\_rest\_day определяет, является ли день выходным для конкретного водителя. Функция adaptive\_mutation\_rate вычисляет адаптивную вероятность мутации на основе стагнации алгоритма. Благодаря адаптивной вероятности мутации алгоритм может найти выход из ситуации, при которой не появляется новых потомков, которые были бы лучше предыдущих результатов.



Рисунок 17. Функции is\_rest\_day и adaptive\_mutation\_rate

Создадим функцию скрещивания two\_point\_crossover, которая комбинирует параметры двух родителей на основе разделения на три части. И функцию check\_shift\_overlap, которая поможет рассчитать количество очков, необходимое отнять от решения за пересечение смен водителей.

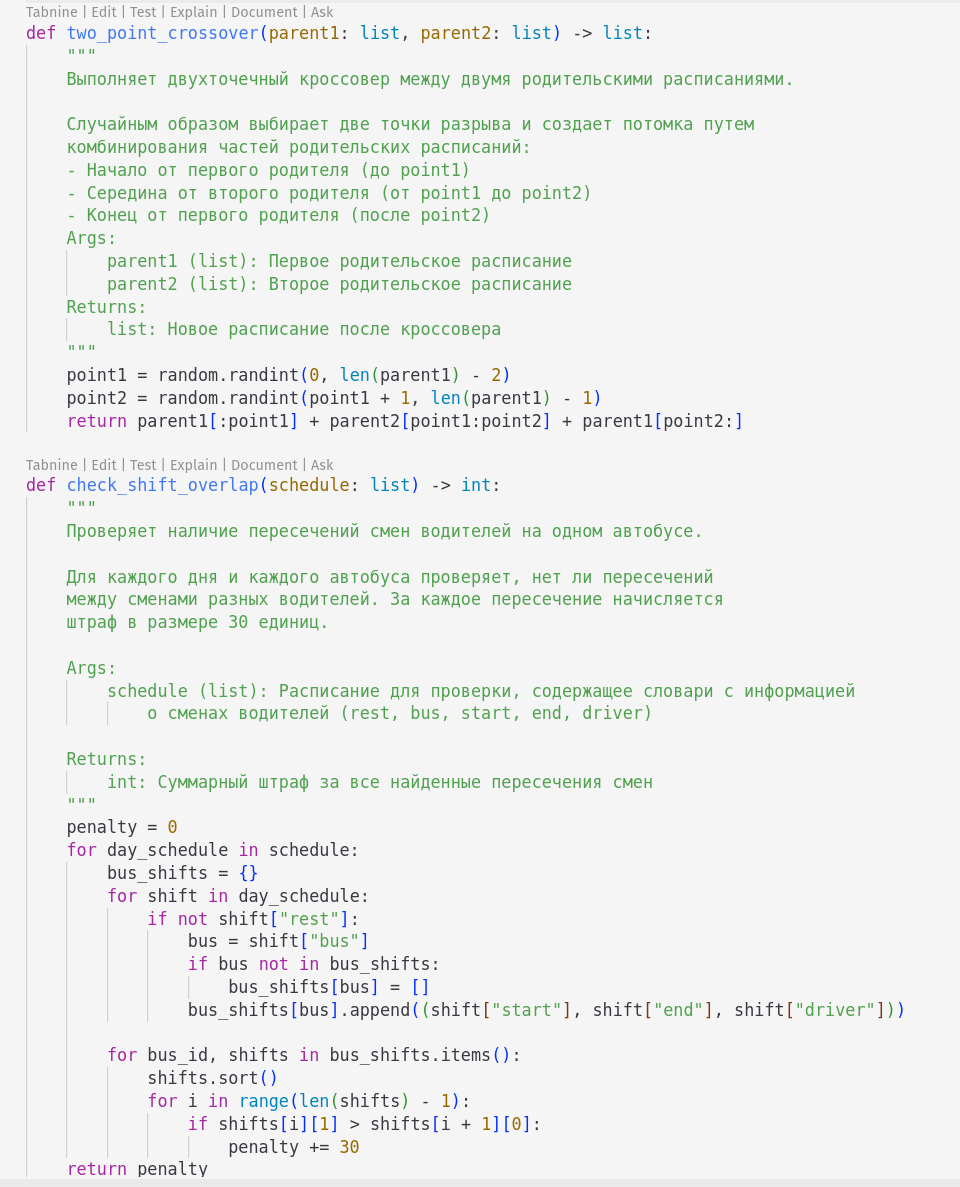


Рисунок 18. Функции two\_point\_crossover и check\_shift\_overlap

Далее создадим одну из самых важных функций, для генетического алгоритма. Функция fitness вычисляет насколько хорошо расписание удовлетворяет заданным критериям. В нашем случае чем больше результат функции fitness, тем лучше расписание удовлетворяет условию задачи.

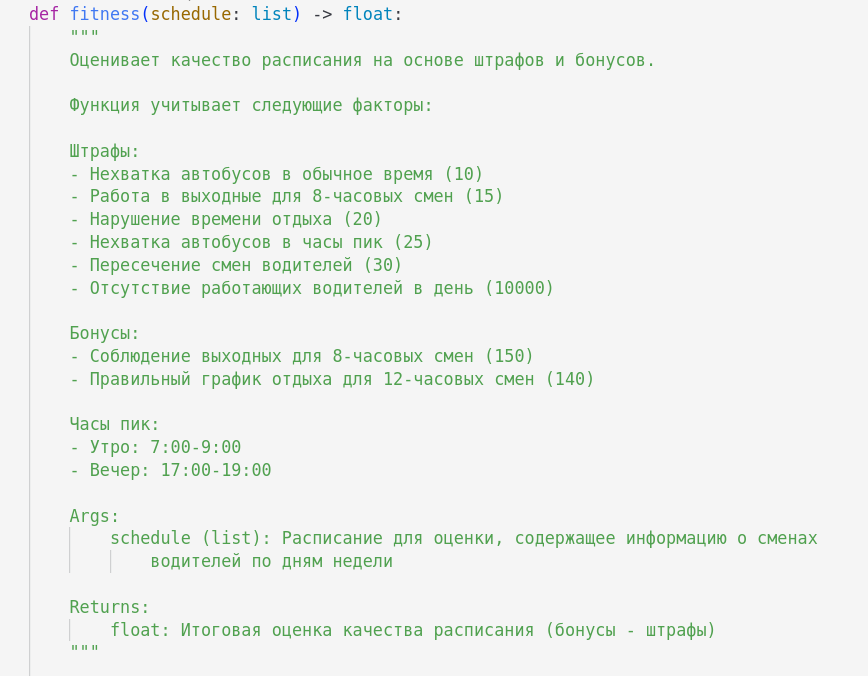


Рисунок 19. Документация функции fitness

Также необходимо реализовать функции создания случайного расписания, то есть начальной популяции.

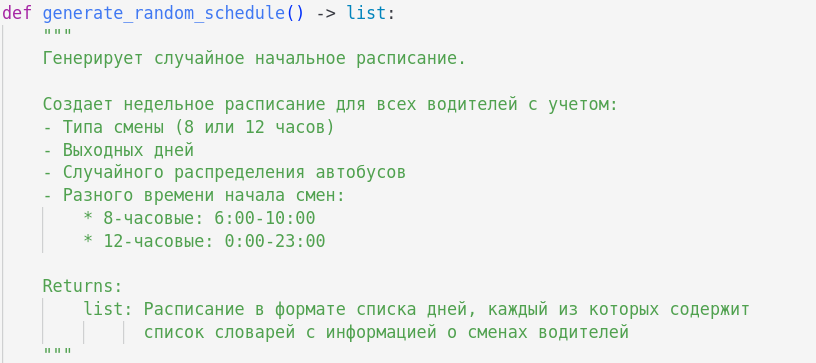


Рисунок 20. Документация функции generate\_random\_schedule для создания случайного расписания

Еще одним обязательным элементом генетического алгоритма является создание мутаций. Мутации обычно происходят не так часто и нужны для появления новых наборов параметров, которые могут быть необходимы для создания наилучшего решения.

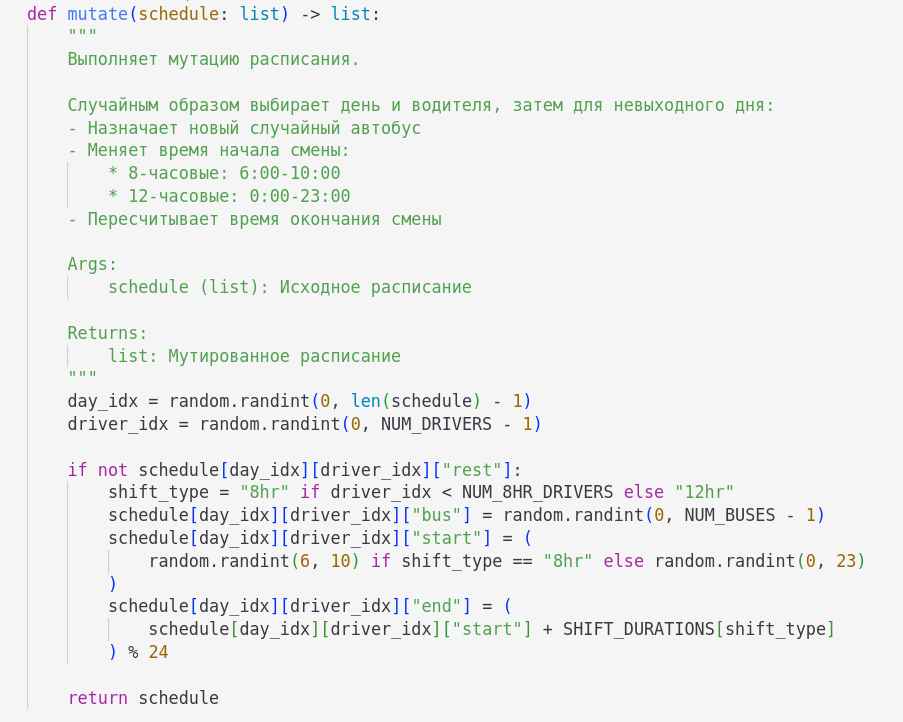


Рисунок 21. Функция mutate для создания мутаций

Теперь главную функцию, а именно функцию генетического алгоритма, которая будет использовать вышеперечисленные функции для создания и отбора особей.

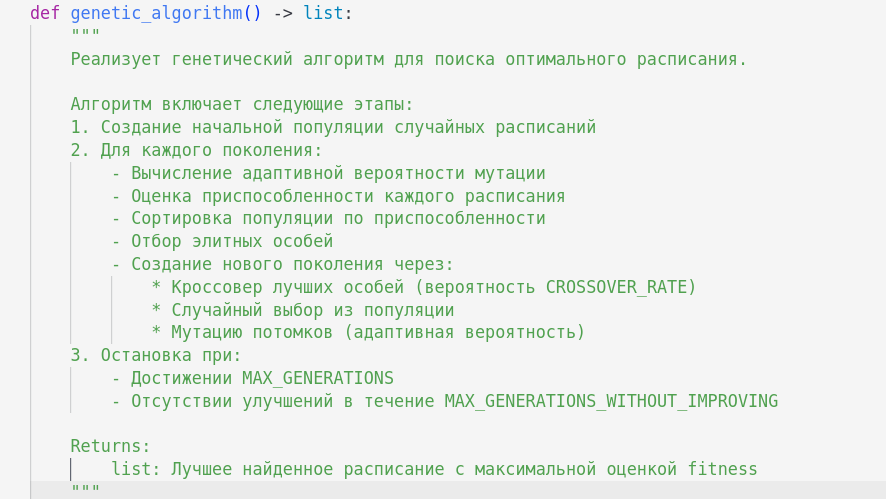


Рисунок 22. Документация функции genetic\_algorithm

Запустим алгоритм для нахождения оптимального расписания. Запишем результат в DataFrame, с помощью функции generate\_schedule\_dataframe (см рис. 24) для удобного отображения расписания.

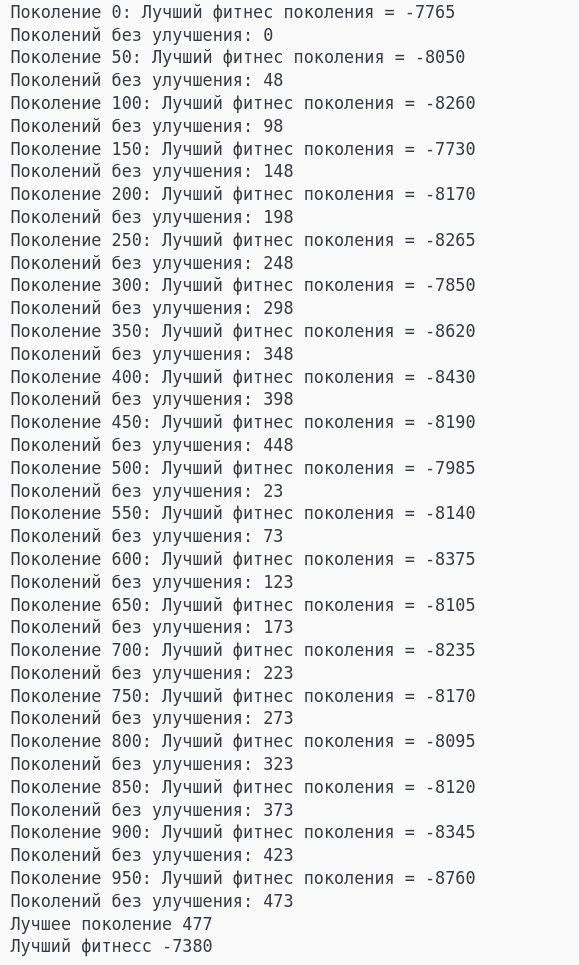


Рисунок 23. Результат работы генетического алгоритма

Отобразим полученное расписание в табличном виде

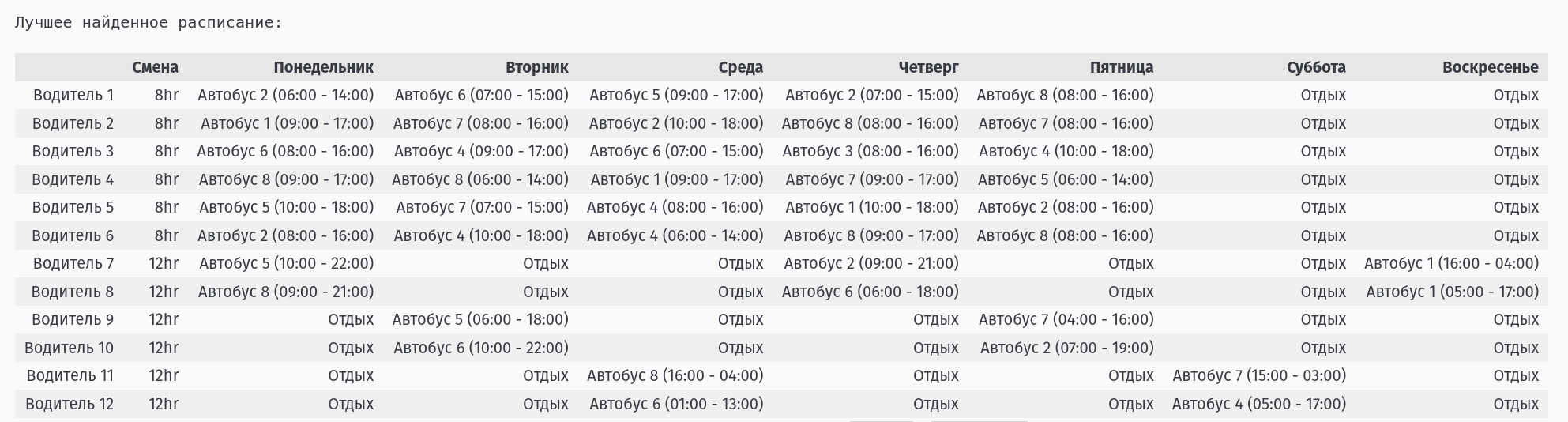


Рисунок 24. Расписание, полученное с помощью генетического алгоритма

Сохраним полученный результат в эксель файл

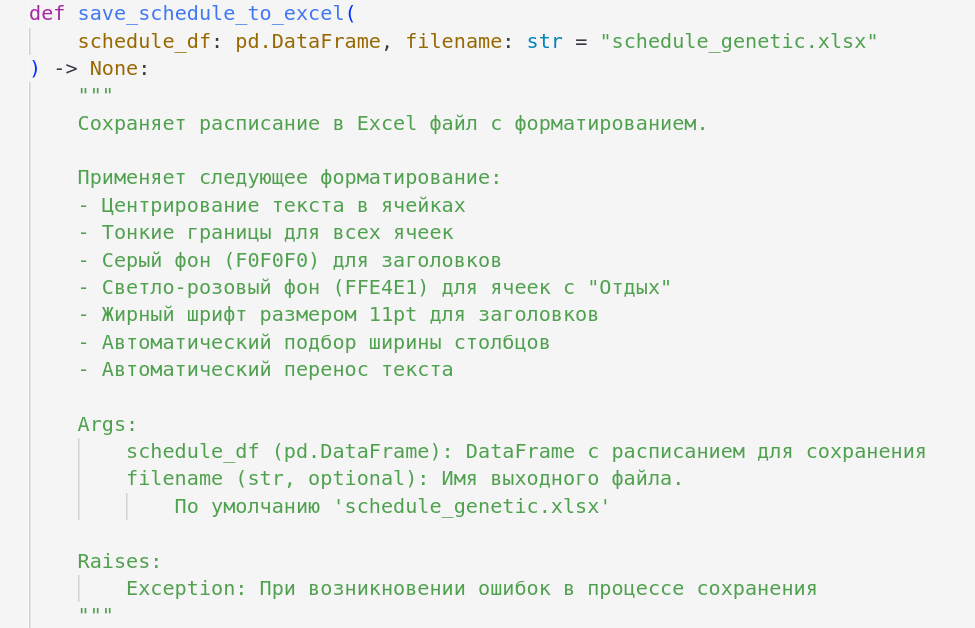


Рисунок 25. Документация функции save\_schedule\_to\_excel

## 

## Сравнение алгоритмов

|  | Алгоритм в лоб | Генетический алгоритм |
| --- | --- | --- |
| Временная сложность | O(D \* T / S)  где D – кол-во водителей, T - комбинации времени начала смены  S – шаг перебора. | O(G \* P \* T)  где G – кол-во поколений,  P – размер популяции,  T – время оценивания особи |
| Ресурсоемкость | Требует большое количество вычислительных ресурсов | Требует меньше ресурсов по сравнению с алгоритмом в лоб, так как использует эвристический подход для поиска решения, избегая проверки всех возможных комбинаций |
| Гибкость | не гибкий, так как жестко следует заданным условиям и не может адаптироваться к изменениям в данных или условиях | Высокая гибкость, так как алгоритм может адаптироваться к изменениям в условиях и данных, изменяя параметры, такие как размер популяции, вероятность мутации и кроссовера |
| Точность | Алгоритм гарантирует нахождение оптимального решения, если таковое существует, поскольку он проверяет все возможные варианты | Генетический алгоритм не гарантирует нахождение оптимального решения, но способен находить достаточно хорошие решения за разумное время |
| Область применения | Подходит для небольших задач, где количество возможных комбинаций ограничено и вычислительные ресурсы не являются критическим фактором | Подходит для сложных задач оптимизации, где точное решение не обязательно, но требуется быстрое нахождение приемлемого решения |
| Практичность | В реальных условиях алгоритм в лоб может быть непрактичным из-за высокой временной и вычислительной стоимости | Практичен для использования в реальных условиях, особенно когда требуется быстрое решение и ресурсы ограничены |
| Масштабируемость | Плохо масштабируется, так как увеличение количества водителей приводит к росту количества комбинаций | Хорошо масштабируется, так как может работать с большими наборами данных |

# Заключение

## Вывод

Для выполнения данной работы мною были изучены такие алгоритмы, как перебор всех вариантов (brute force) и генетический алгоритм. Решая задачу с расписанием автобусов, я смог понять разницу в двух методах решения, выделил их плюсы и минусы, а также получил знания и навыки для решения подобных задач.

# Список источников

1. Документация языка программирования Python, раздел классы.

<https://docs.python.org/3/tutorial/classes.html>

1. Объяснение brute force алгоритма.

<https://www.freecodecamp.org/news/brute-force-algorithms-explained/>

1. Объяснение генетического алгоритма

<https://www.datacamp.com/tutorial/genetic-algorithm-python>

# Приложения

## GitHub

<https://github.com/Sypoo1/mtuci-DSA-2/blob/main/course_work.ipynb>

## Код

*# %% [markdown]*

*# Курсовая работа*

*#*

*# %% [markdown]*

*# Алгоритм в лоб*

*# %%*

"""

Модуль для управления расписанием водителей автобусов.

Константы:

DEFAULT\_TRIP\_DURATION (int): Стандартная длительность поездки в минутах

DEFAULT\_REST\_DURATION (int): Стандартная длительность отдыха в минутах

PEAK\_HOURS (list): Список кортежей с временными интервалами часов пик

PEAK\_HOURS\_ADDITIONAL\_TIME (int): Дополнительное время в минутах для поездок в час пик

BRUTE\_FORCE\_STEP (int): Шаг для поиска оптимального расписания

"""

DEFAULT\_TRIP\_DURATION = 90

DEFAULT\_REST\_DURATION = 10

PEAK\_HOURS = [("07:00", "09:00"), ("17:00", "19:00")]

PEAK\_HOURS\_ADDITIONAL\_TIME = 5

BUS\_COUNT = 2

DRIVERS\_PER\_GROUP\_COUNT = 4

BRUTE\_FORCE\_STEP = 35

*# %%*

from datetime import datetime, timedelta

import pandas as pd

def is\_peak\_hour(current\_time: datetime) -> bool:

"""

Проверяет, является ли указанное время часом пик.

Args:

current\_time (datetime): Время для проверки

Returns:

bool: True если время попадает в час пик, False в противном случае

"""

for start, end in PEAK\_HOURS:

peak\_start = datetime.strptime(start, "%H:%M").time()

peak\_end = datetime.strptime(end, "%H:%M").time()

if peak\_start <= current\_time.time() < peak\_end:

return True

return False

def get\_trip\_duration(

start\_time: datetime, base\_duration: int = DEFAULT\_TRIP\_DURATION

) -> int:

"""

Рассчитывает фактическую длительность поездки с учетом часов пик.

Args:

start\_time (datetime): Время начала поездки

base\_duration (int): Базовая длительность поездки в минутах

Returns:

int: Фактическая длительность поездки в минутах

"""

end\_time = start\_time + timedelta(minutes=base\_duration)

if is\_peak\_hour(start\_time) or is\_peak\_hour(end\_time):

base\_duration += 5

return base\_duration

class TimeInterval:

"""

Класс для представления временного интервала.

Attributes:

start (datetime): Время начала интервала

end (datetime): Время окончания интервала

"""

def \_\_init\_\_(self, start: datetime, end: datetime):

"""

Args:

start (datetime): Время начала интервала

end (datetime): Время окончания интервала

"""

self.start = start

self.end = end

def \_\_repr\_\_(self):

return f"{self.start.strftime('%H:%M')} - {self.end.strftime('%H:%M')}"

def overlaps(self, other) -> bool:

"""

Проверяет, пересекается ли текущий интервал с другим.

Args:

other (TimeInterval): Другой временной интервал

Returns:

bool: True если интервалы пересекаются, False в противном случае

"""

return self.start < other.end and self.end > other.start

class Bus:

"""

Класс для представления автобуса.

Attributes:

id (int): Идентификатор автобуса

capacity (int): Вместимость автобуса

speed (int): Скорость автобуса

type (str): Тип автобуса

"""

def \_\_init\_\_(

self, id: int, capacity: int = 100, speed: int = 60, type: str = "Электробус"

):

"""

Args:

id (int): Идентификатор автобуса

capacity (int, optional): Вместимость. По умолчанию 100

speed (int, optional): Скорость. По умолчанию 60

type (str, optional): Тип автобуса. По умолчанию "Электробус"

"""

self.id = id

self.capacity = capacity

self.speed = speed

self.type = type

class Driver:

"""

Класс для представления водителя.

Attributes:

id (int): Идентификатор водителя

bus\_id (int): Идентификатор закрепленного автобуса

work\_time (list): Список рабочих интервалов

rest\_time (list): Список интервалов отдыха

type (str): Тип смены ("12\_hr" или "8\_hr")

experience (int): Опыт работы

"""

def \_\_init\_\_(

self,

id: int,

bus\_id: int,

work\_time: [(TimeInterval, int)] = None,

rest\_time: [TimeInterval] = None,

type: str = "12\_hr",

expirience: int = 0,

):

self.id = id

self.bus\_id = bus\_id

self.work\_time = work\_time if work\_time is not None else []

self.rest\_time = rest\_time if rest\_time is not None else []

self.type = type

self.expirience = expirience

def get\_trip\_intevals(self) -> [TimeInterval]:

"""

Возвращает список всех рабочих интервалов водителя.

Returns:

list: Список объектов TimeInterval

"""

return [interval for interval, bus\_id in self.work\_time]

def get\_last\_trip\_inteval(self) -> TimeInterval | None:

"""

Возвращает последний рабочий интервал водителя.

Returns:

TimeInterval: Последний интервал или None, если интервалов нет

"""

return self.get\_trip\_intevals()[-1] if self.get\_trip\_intevals() else None

def get\_work\_hours(self) -> int | None:

"""

Возвращает количество рабочих часов в зависимости от типа смены.

Returns:

int: Количество часов или None для неизвестного типа смены

"""

if self.type == "12\_hr":

return 12

elif self.type == "8\_hr":

return 8

return None

def get\_rest\_minutes(self) -> int | None:

"""

Возвращает длительность перерыва в зависимости от типа смены.

Returns:

int: Количество минут или None для неизвестного типа смены

"""

if self.type == "12\_hr":

return 10

elif self.type == "8\_hr":

return 60

return None

def generate\_schedule(self, start\_time: str) -> None:

"""

Генерирует расписание работы и отдыха водителя.

Args:

start\_time (str): Время начала работы в формате "HH:MM"

Note:

Метод создает расписание с учетом:

- Типа смены (8 или 12 часов)

- Обязательных перерывов

- Часов пик

- Максимальной продолжительности работы

"""

self.work\_time.clear()

self.rest\_time.clear()

start = datetime.strptime(start\_time, "%H:%M")

end\_time = start + timedelta(hours=self.get\_work\_hours())

current\_time = start

trip\_count = 0

while current\_time < end\_time:

trip\_duration = get\_trip\_duration(current\_time)

trip\_start = current\_time

trip\_end = current\_time + timedelta(minutes=trip\_duration)

if trip\_end.time() < trip\_start.time():

self.work\_time.append((TimeInterval(trip\_start, trip\_end), self.bus\_id))

break

self.work\_time.append((TimeInterval(trip\_start, trip\_end), self.bus\_id))

trip\_count += 1

current\_time = trip\_end

if self.type == "12\_hr" and trip\_count % 2 == 0:

rest\_start = current\_time

rest\_end = current\_time + timedelta(minutes=self.get\_rest\_minutes())

if rest\_end.time() < rest\_start.time():

rest\_end = datetime.strptime("00:00", "%H:%M").time()

self.rest\_time.append(TimeInterval(rest\_start, rest\_end))

break

self.rest\_time.append(TimeInterval(rest\_start, rest\_end))

current\_time = rest\_end

*# %%*

def check\_continuous\_coverage(intervals) -> tuple:

"""

Проверяет непрерывность покрытия рабочих смен в течение суток.

Функция сортирует интервалы по времени начала и проверяет наличие

промежутков между последовательными интервалами. Также проверяется,

что последний интервал достигает конца суток.

Args:

intervals (list): Список объектов TimeInterval, представляющих рабочие смены

Returns:

tuple: Кортеж из трех элементов:

- bool: True если покрытие непрерывно, False если есть разрывы

- str|None: Время окончания интервала, где найден разрыв (формат "HH:MM")

- str|None: Время начала следующего интервала (формат "HH:MM")

"""

intervals.sort(key=lambda x: x.start)

for i in range(len(intervals) - 1):

if (

intervals[i].end < intervals[i + 1].start

and intervals[i].start < intervals[i].end

):

return (

False,

intervals[i].end.strftime("%H:%M"),

intervals[i + 1].start.strftime("%H:%M"),

)

if (

intervals[-1].end.time() < datetime.strptime("23:59", "%H:%M").time()

and intervals[-1].start.time() < intervals[-1].end.time()

):

return False, intervals[-1].end.strftime("%H:%M"), "23:59"

return True, None, None

def get\_all\_trips\_intervals(drivers) -> [TimeInterval]:

"""

Собирает все рабочие интервалы всех водителей в единый список.

Args:

drivers (list): Список объектов Driver

Returns:

list: Список всех рабочих интервалов (объекты TimeInterval)

"""

intervals = []

for driver in drivers:

intervals.extend(driver.get\_trip\_intevals())

return intervals

def find\_valid\_schedule(drivers, drivers\_start\_time) -> bool:

"""

Ищет валидное расписание для группы водителей методом перебора времени начала смен.

Функция перебирает различные комбинации времени начала смен для водителей,

пытаясь найти такое расписание, которое обеспечивает непрерывное покрытие

рабочего дня и удовлетворяет дополнительным условиям.

Args:

drivers (list): Список объектов Driver

drivers\_start\_time (dict): Словарь с начальным временем смен для каждого водителя

{id\_водителя: "HH:MM"}

Returns:

bool: True если найдено валидное расписание, False если нет

Note:

- Перебор времени осуществляется с шагом BRUTE\_FORCE\_STEP минут

- Проверяются следующие условия:

1. Непрерывность покрытия всех смен

2. Время окончания смены водителя №3 должно быть раньше 23:45

3. Все смены должны заканчиваться в тот же день, когда начались

"""

start\_time = datetime.strptime("00:00", "%H:%M")

end\_time = datetime.strptime("23:55", "%H:%M")

time\_step = timedelta(minutes=BRUTE\_FORCE\_STEP)

all\_times = []

current\_time = start\_time

while current\_time <= end\_time:

all\_times.append(current\_time.strftime("%H:%M"))

current\_time += time\_step

for time\_2 in all\_times:

drivers\_start\_time[2] = time\_2

for time\_3 in all\_times:

drivers\_start\_time[3] = time\_3

for time\_4 in all\_times:

drivers\_start\_time[4] = time\_4

for driver in drivers:

driver.generate\_schedule(drivers\_start\_time[driver.id])

all\_intervals = get\_all\_trips\_intervals(drivers)

is\_continuous\_coverage, start\_time, end\_time = (

check\_continuous\_coverage(all\_intervals)

)

driver\_3\_last\_trip = drivers[2].get\_last\_trip\_inteval()

if (

is\_continuous\_coverage

and driver\_3\_last\_trip

and driver\_3\_last\_trip.end.time()

< datetime.strptime("23:45", "%H:%M").time()

and driver\_3\_last\_trip.start.time() < driver\_3\_last\_trip.end.time()

):

return True

return False

*# %%*

"""

Инициализация и тестирование системы расписания водителей.

Этот блок кода выполняет следующие действия:

1. Создает автобусы и водителей

2. Устанавливает начальное время работы

3. Ищет валидное расписание

4. Выводит результаты

Константы:

BUS\_COUNT (int): Количество автобусов (2)

DRIVERS\_PER\_GROUP\_COUNT (int): Количество водителей в группе (4)

Структура данных:

buses (list): Список объектов Bus

- Создается два автобуса с ID 1 и 2

drivers (list): Список объектов Driver

- Создается 4 водителя

- Четные ID привязаны к автобусу №2

- Нечетные ID привязаны к автобусу №1

drivers\_start\_time (dict): Начальное время старта для каждого водителя

- Ключи: ID водителей (1-4)

- Значения: время начала смены в формате "HH:MM"

Вывод результатов:

Успешный случай:

- Для каждого водителя выводится:

\* Время поездок с номером автобуса

\* Время перерывов

Неуспешный случай:

- Сообщение о невозможности найти валидное расписание

Примечание:

Валидное расписание должно обеспечивать:

- Непрерывное покрытие рабочего дня

- Корректное распределение автобусов

- Соблюдение требований к времени работы и отдыха

"""

buses = [Bus(id=x) for x in range(1, BUS\_COUNT + 1)]

drivers = [

Driver(id=driver\_id, bus\_id=(2 if driver\_id % 2 == 0 else 1))

for driver\_id in range(1, DRIVERS\_PER\_GROUP\_COUNT + 1)

]

drivers\_start\_time = {1: "00:00", 2: "00:00", 3: "00:00", 4: "00:00"}

if find\_valid\_schedule(drivers, drivers\_start\_time):

for d in drivers:

for interval in d.get\_trip\_intevals():

print(f"Водитель {d.id}, поездка: {interval}, автобус: №{d.bus\_id}")

for interval in d.rest\_time:

print(f"Водитель {d.id}, отдых: {interval}")

else:

print("Не удалось найти валидное расписание")

*# %%*

"""

Создание и форматирование недельного расписания водителей в виде таблицы pandas DataFrame.

Основные этапы:

1. Инициализация водителей

2. Определение графика работы групп

3. Создание структуры таблицы

4. Заполнение данных

5. Форматирование и стилизация

Структура данных:

all\_drivers (list): Список из 12 водителей

- 3 группы по 4 водителя

- Четные ID привязаны к автобусу №2

- Нечетные ID привязаны к автобусу №1

work\_schedule (dict): График работы для каждой группы

- Группа 1: Понедельник, Четверг, Воскресенье

- Группа 2: Вторник, Пятница

- Группа 3: Среда, Суббота

days (list): Список дней недели для колонок таблицы

columns (MultiIndex): Двухуровневый индекс колонок

- Первый уровень: дни недели

- Второй уровень: "График работы" и "Время отдыха"

Формат таблицы:

Индекс:

- Информация о водителе (номер, автобус, группа)

Колонки:

- Основные: дни недели

- Подколонки: график работы и время отдыха

Содержимое ячеек:

- Для рабочих дней: времена поездок и перерывов

- Для выходных: "Выходной" и "---"

Стилизация:

- Центрирование текста

- Границы ячеек

- Разные размеры шрифта для заголовков

- Фоновый цвет для основных заголовков

- Автоматический перенос текста

Примечание:

Расписание генерируется на основе начального времени из drivers\_start\_time

с учетом принадлежности водителя к определенной группе.

"""

all\_drivers = []

for group in range(3):

for driver\_id in range(1, 5):

current\_id = driver\_id + (group \* 4)

bus\_id = 2 if driver\_id % 2 == 0 else 1

all\_drivers.append(Driver(id=current\_id, bus\_id=bus\_id))

work\_schedule = {

1: ["Понедельник", "Четверг", "Воскресенье"],

2: ["Вторник", "Пятница"],

3: ["Среда", "Суббота"],

}

days = [

"Понедельник",

"Вторник",

"Среда",

"Четверг",

"Пятница",

"Суббота",

"Воскресенье",

]

columns = pd.MultiIndex.from\_product([days, ["График работы", "Время отдыха"]])

data = []

for driver in all\_drivers:

group\_num = (driver.id - 1) // 4 + 1

driver.generate\_schedule(

drivers\_start\_time[driver.id % 4 if driver.id % 4 != 0 else 4]

)

row\_data = {}

driver\_info = f"Водитель №{driver.id}\nАвтобус №{driver.bus\_id}\nГруппа {group\_num}"

row\_data["Информация"] = driver\_info

for day in days:

if day in work\_schedule[group\_num]:

work\_times = [

f"{interval.start.strftime('%H:%M')}-{interval.end.strftime('%H:%M')}"

for interval, \_ in driver.work\_time

]

rest\_times = [

f"{interval.start.strftime('%H:%M')}-{interval.end.strftime('%H:%M')}"

for interval in driver.rest\_time

]

row\_data[(day, "График работы")] = "\n".join(work\_times)

row\_data[(day, "Время отдыха")] = "\n".join(rest\_times)

else:

row\_data[(day, "График работы")] = "Выходной"

row\_data[(day, "Время отдыха")] = "---"

data.append(row\_data)

df = pd.DataFrame(data)

df.set\_index("Информация", inplace=True)

pd.set\_option("display.max\_columns", None)

pd.set\_option("display.max\_rows", None)

pd.set\_option("display.width", None)

pd.set\_option("display.max\_colwidth", None)

styled\_df = df.style.set\_properties(

\*\*{"text-align": "center", "white-space": "pre-wrap"}

).set\_table\_styles(

[

{"selector": "th", "props": [("text-align", "center")]},

{"selector": "", "props": [("border", "1px solid black")]},

{"selector": "th.col\_heading", "props": [("font-size", "11pt")]},

{

"selector": "th.col\_heading.level0",

"props": [("font-size", "12pt"), ("background-color", "#f0f0f0")],

},

]

)

print("\nРасписание работы водителей на неделю:")

display(styled\_df)

*# %%*

from openpyxl.styles import Alignment, Border, Font, PatternFill, Side

from openpyxl.utils import get\_column\_letter

"""

Сохранение и форматирование таблицы расписания в Excel-файл.

Импорты:

openpyxl.styles: Классы для стилизации Excel

openpyxl.utils: Утилиты для работы с Excel

Основные этапы:

1. Создание Excel-файла

2. Запись данных из DataFrame

3. Применение стилей форматирования

4. Настройка размеров колонок

Форматирование:

border: Тонкие границы для всех ячеек

header\_fill: Серый фон для заголовков

alignment: Центрирование текста и автоперенос

font: Размер шрифта 11pt для заголовков

Особенности:

- Используется openpyxl для расширенного форматирования

- Автоматический расчет ширины колонок

- Двухуровневые заголовки (строки 1 и 2)

- Сохранение в файл 'schedule.xlsx'

Примечание:

Форматирование применяется после записи данных

и сохраняет все стили из исходного DataFrame

"""

with pd.ExcelWriter("schedule.xlsx", engine="openpyxl") as writer:

df.to\_excel(writer, sheet\_name="Расписание")

worksheet = writer.sheets["Расписание"]

border = Border(

left=Side(style="thin"),

right=Side(style="thin"),

top=Side(style="thin"),

bottom=Side(style="thin"),

)

header\_fill = PatternFill(

start\_color="F0F0F0", end\_color="F0F0F0", fill\_type="solid"

)

for row in worksheet.iter\_rows():

for cell in row:

cell.alignment = Alignment(

horizontal="center", vertical="center", wrap\_text=True

)

cell.border = border

if cell.row in [1, 2]:

cell.font = Font(size=11)

cell.fill = header\_fill

for column in worksheet.columns:

max\_length = 0

column\_letter = get\_column\_letter(column[0].column)

for cell in column:

try:

if len(str(cell.value)) > max\_length:

max\_length = len(str(cell.value))

except Exception as e:

print(e)

adjusted\_width = max\_length + 2

worksheet.column\_dimensions[column\_letter].width = adjusted\_width

print("Расписание сохранено в файл 'schedule.xlsx'")

*# %% [markdown]*

*# Генетический алгоритм*

*# %%*

"""

Конфигурация системы составления расписания автобусного парка.

Этот модуль содержит основные константы и настройки для генетического алгоритма,

используемого при составлении оптимального расписания работы водителей автобусного парка.

Constants:

Параметры автобусного парка:

NUM\_BUSES (int): Общее количество автобусов (8)

NUM\_DRIVERS (int): Общее количество водителей (12)

NUM\_8HR\_DRIVERS (int): Количество водителей с 8-часовой сменой (6)

NUM\_12HR\_DRIVERS (int): Количество водителей с 12-часовой сменой (6)

Временные параметры:

SHIFT\_DURATIONS (dict): Словарь длительностей смен

Ключи: "8hr", "12hr"

Значения: 8, 12 (часов соответственно)

DAYS\_OF\_WEEK (list): Список дней недели на русском языке

TIME\_SLOTS (range): Диапазон часов в сутках (0-23)

Параметры генетического алгоритма:

POPULATION\_SIZE (int): Размер популяции в каждом поколении (100)

MAX\_GENERATIONS (int): Максимальное количество поколений (1000)

MUTATION\_RATE (float): Базовая вероятность мутации (0.1)

CROSSOVER\_RATE (float): Вероятность кроссовера (0.8)

MAX\_GENERATIONS\_WITHOUT\_IMPROVING (int): Максимум поколений без улучшения (1000)

ELITE\_SIZE (int): Количество элитных особей (POPULATION\_SIZE // 20)

Dependencies:

random: Для генерации случайных чисел

datetime: Для работы с датами и временем

pandas: Для создания и обработки таблиц данных

openpyxl.styles: Для стилизации Excel-файлов

openpyxl.utils: Для работы с Excel-файлами

Notes:

Система поддерживает два типа смен:

- 8-часовые смены для первых 6 водителей

- 12-часовые смены для остальных 6 водителей

Расписание составляется на неделю с учетом:

- Выходных дней

- Часов пик

- Требований к отдыху водителей

- Оптимального распределения автобусов

Генетический алгоритм использует:

- Элитарную стратегию отбора

- Адаптивную мутацию

- Двухточечный кроссовер

"""

import random

from datetime import datetime

import pandas as pd

from openpyxl.styles import Alignment, Border, Font, PatternFill, Side

from openpyxl.utils import get\_column\_letter

NUM\_BUSES = 8

NUM\_DRIVERS = 12

NUM\_8HR\_DRIVERS = 6

NUM\_12HR\_DRIVERS = NUM\_DRIVERS - NUM\_8HR\_DRIVERS

SHIFT\_DURATIONS = {"8hr": 8, "12hr": 12}

DAYS\_OF\_WEEK = [

"Понедельник",

"Вторник",

"Среда",

"Четверг",

"Пятница",

"Суббота",

"Воскресенье",

]

TIME\_SLOTS = range(24)

POPULATION\_SIZE = 100

MAX\_GENERATIONS = 1000

MUTATION\_RATE = 0.1

CROSSOVER\_RATE = 0.8

MAX\_GENERATIONS\_WITHOUT\_IMPROVING = 1000

ELITE\_SIZE = POPULATION\_SIZE // 20

def is\_rest\_day(driver\_id: int, day\_idx: int) -> bool:

"""

Определяет, является ли день выходным для конкретного водителя.

Распределяет выходные дни в зависимости от типа смены водителя:

- Водители с 8-часовыми сменами: выходные в субботу и воскресенье

- Три группы 12-часовых водителей с разными графиками работы:

1 группа: работает в пн, чт, вс

2 группа: работает во вт, пт

3 группа: работает в ср, сб

Args:

driver\_id (int): Идентификатор водителя

day\_idx (int): Индекс дня недели

Returns:

bool: True если день является выходным для водителя

"""

if driver\_id < NUM\_8HR\_DRIVERS:

return DAYS\_OF\_WEEK[day\_idx] in ["Суббота", "Воскресенье"]

elif driver\_id < NUM\_8HR\_DRIVERS + 2:

return DAYS\_OF\_WEEK[day\_idx] not in ["Понедельник", "Четверг", "Воскресенье"]

elif driver\_id < NUM\_8HR\_DRIVERS + 4:

return DAYS\_OF\_WEEK[day\_idx] not in ["Вторник", "Пятница"]

else:

return DAYS\_OF\_WEEK[day\_idx] not in ["Среда", "Суббота"]

def adaptive\_mutation\_rate(generations\_without\_improvement: int) -> float:

"""

Вычисляет адаптивную вероятность мутации на основе стагнации алгоритма.

Увеличивает вероятность мутации при отсутствии улучшений, чтобы помочь

алгоритму выйти из локального оптимума. Максимальная вероятность

ограничена значением 0.5.

Args:

generations\_without\_improvement (int): Количество поколений без улучшения

Returns:

float: Вероятность мутации в диапазоне [0.1, 0.5]

"""

base\_rate = 0.1

return min(base\_rate \* (1 + generations\_without\_improvement / 50), 0.5)

def two\_point\_crossover(parent1: list, parent2: list) -> list:

"""

Выполняет двухточечный кроссовер между двумя родительскими расписаниями.

Случайным образом выбирает две точки разрыва и создает потомка путем

комбинирования частей родительских расписаний:

- Начало от первого родителя (до point1)

- Середина от второго родителя (от point1 до point2)

- Конец от первого родителя (после point2)

Args:

parent1 (list): Первое родительское расписание

parent2 (list): Второе родительское расписание

Returns:

list: Новое расписание после кроссовера

"""

point1 = random.randint(0, len(parent1) - 2)

point2 = random.randint(point1 + 1, len(parent1) - 1)

return parent1[:point1] + parent2[point1:point2] + parent1[point2:]

def check\_shift\_overlap(schedule: list) -> int:

"""

Проверяет наличие пересечений смен водителей на одном автобусе.

Для каждого дня и каждого автобуса проверяет, нет ли пересечений

между сменами разных водителей. За каждое пересечение начисляется

штраф в размере 30 единиц.

Args:

schedule (list): Расписание для проверки, содержащее словари с информацией

о сменах водителей (rest, bus, start, end, driver)

Returns:

int: Суммарный штраф за все найденные пересечения смен

"""

penalty = 0

for day\_schedule in schedule:

bus\_shifts = {}

for shift in day\_schedule:

if not shift["rest"]:

bus = shift["bus"]

if bus not in bus\_shifts:

bus\_shifts[bus] = []

bus\_shifts[bus].append((shift["start"], shift["end"], shift["driver"]))

for bus\_id, shifts in bus\_shifts.items():

shifts.sort()

for i in range(len(shifts) - 1):

if shifts[i][1] > shifts[i + 1][0]:

penalty += 30

return penalty

def fitness(schedule: list) -> float:

"""

Оценивает качество расписания на основе штрафов и бонусов.

Функция учитывает следующие факторы:

Штрафы:

- Нехватка автобусов в обычное время (10)

- Работа в выходные для 8-часовых смен (15)

- Нарушение времени отдыха (20)

- Нехватка автобусов в часы пик (25)

- Пересечение смен водителей (30)

- Отсутствие работающих водителей в день (10000)

Бонусы:

- Соблюдение выходных для 8-часовых смен (150)

- Правильный график отдыха для 12-часовых смен (140)

Часы пик:

- Утро: 7:00-9:00

- Вечер: 17:00-19:00

Args:

schedule (list): Расписание для оценки, содержащее информацию о сменах

водителей по дням недели

Returns:

float: Итоговая оценка качества расписания (бонусы - штрафы)

"""

PENALTIES = {

"bus\_shortage": 10,

"weekend\_work": 15,

"rest\_violation": 20,

"peak\_hours\_shortage": 25,

"shift\_overlap": 30,

"no\_workers\_day": 10000,

}

BONUSES = {

"weekend\_rest": 150,

"proper\_rest\_12hr": 140,

}

PEAK\_HOURS = {"morning": range(7, 9), "evening": range(17, 19)}

penalty = 0

bonus = 0

bus\_coverage = {day: {hour: [] for hour in TIME\_SLOTS} for day in DAYS\_OF\_WEEK}

for day\_idx, daily\_schedule in enumerate(schedule):

working\_drivers = sum(

1 for driver\_schedule in daily\_schedule if not driver\_schedule["rest"]

)

if working\_drivers == 0:

penalty += PENALTIES["no\_workers\_day"]

weekend\_rest\_violation = False

for day\_idx, day in enumerate(DAYS\_OF\_WEEK):

if day in ["Суббота", "Воскресенье"]:

for driver\_schedule in schedule[day\_idx]:

if (

driver\_schedule["driver"] < NUM\_8HR\_DRIVERS

and not driver\_schedule["rest"]

):

weekend\_rest\_violation = True

break

if not weekend\_rest\_violation:

bonus += BONUSES["weekend\_rest"]

for driver\_id in range(NUM\_8HR\_DRIVERS, NUM\_DRIVERS):

proper\_rest = True

for day\_idx in range(1, len(DAYS\_OF\_WEEK)):

if not schedule[day\_idx][driver\_id]["rest"]:

for prev\_day in range(max(0, day\_idx - 2), day\_idx):

if not schedule[prev\_day][driver\_id]["rest"]:

proper\_rest = False

break

if proper\_rest:

bonus += BONUSES["proper\_rest\_12hr"]

penalty += check\_shift\_overlap(schedule)

for day\_idx, daily\_schedule in enumerate(schedule):

for driver\_schedule in daily\_schedule:

if driver\_schedule["rest"]:

continue

bus = driver\_schedule["bus"]

start, end = driver\_schedule["start"], driver\_schedule["end"]

if bus is not None:

hours = range(start, end + 1 if end >= start else end + 24)

for hour in hours:

hour = hour % 24

bus\_coverage[DAYS\_OF\_WEEK[day\_idx]][hour].append(bus)

if driver\_schedule["driver"] < NUM\_8HR\_DRIVERS and DAYS\_OF\_WEEK[

day\_idx

] in ["Суббота", "Воскресенье"]:

penalty += PENALTIES["weekend\_work"]

if driver\_schedule["driver"] >= NUM\_8HR\_DRIVERS:

if day\_idx > 0:

prev\_day = schedule[day\_idx - 1]

for prev\_driver in prev\_day:

if (

prev\_driver["driver"] == driver\_schedule["driver"]

and not prev\_driver["rest"]

):

penalty += PENALTIES["rest\_violation"]

for day, coverage in bus\_coverage.items():

for hour, buses in coverage.items():

buses\_working = len(set(buses))

shortage = max(0, NUM\_BUSES - buses\_working)

if hour in PEAK\_HOURS["morning"] or hour in PEAK\_HOURS["evening"]:

penalty += shortage \* PENALTIES["peak\_hours\_shortage"]

else:

if 6 <= hour <= 22:

penalty += shortage \* PENALTIES["bus\_shortage"]

return bonus - penalty

def generate\_random\_schedule() -> list:

"""

Генерирует случайное начальное расписание.

Создает недельное расписание для всех водителей с учетом:

- Типа смены (8 или 12 часов)

- Выходных дней

- Случайного распределения автобусов

- Разного времени начала смен:

\* 8-часовые: 6:00-10:00

\* 12-часовые: 0:00-23:00

Returns:

list: Расписание в формате списка дней, каждый из которых содержит

список словарей с информацией о сменах водителей

"""

schedule = []

for day\_idx, day in enumerate(DAYS\_OF\_WEEK):

daily\_schedule = []

for driver\_id in range(NUM\_DRIVERS):

if is\_rest\_day(driver\_id, day\_idx):

daily\_schedule.append(

{

"driver": driver\_id,

"bus": None,

"start": None,

"end": None,

"rest": True,

}

)

else:

bus = random.randint(0, NUM\_BUSES - 1)

shift\_type = "8hr" if driver\_id < NUM\_8HR\_DRIVERS else "12hr"

start\_time = (

random.randint(6, 10)

if shift\_type == "8hr"

else random.randint(0, 23)

)

end\_time = (start\_time + SHIFT\_DURATIONS[shift\_type]) % 24

daily\_schedule.append(

{

"driver": driver\_id,

"bus": bus,

"start": start\_time,

"end": end\_time,

"rest": False,

}

)

schedule.append(daily\_schedule)

return schedule

def mutate(schedule: list) -> list:

"""

Выполняет мутацию расписания.

Случайным образом выбирает день и водителя, затем для невыходного дня:

- Назначает новый случайный автобус

- Меняет время начала смены:

\* 8-часовые: 6:00-10:00

\* 12-часовые: 0:00-23:00

- Пересчитывает время окончания смены

Args:

schedule (list): Исходное расписание

Returns:

list: Мутированное расписание

"""

day\_idx = random.randint(0, len(schedule) - 1)

driver\_idx = random.randint(0, NUM\_DRIVERS - 1)

if not schedule[day\_idx][driver\_idx]["rest"]:

shift\_type = "8hr" if driver\_idx < NUM\_8HR\_DRIVERS else "12hr"

schedule[day\_idx][driver\_idx]["bus"] = random.randint(0, NUM\_BUSES - 1)

schedule[day\_idx][driver\_idx]["start"] = (

random.randint(6, 10) if shift\_type == "8hr" else random.randint(0, 23)

)

schedule[day\_idx][driver\_idx]["end"] = (

schedule[day\_idx][driver\_idx]["start"] + SHIFT\_DURATIONS[shift\_type]

) % 24

return schedule

def genetic\_algorithm() -> list:

"""

Реализует генетический алгоритм для поиска оптимального расписания.

Алгоритм включает следующие этапы:

1. Создание начальной популяции случайных расписаний

2. Для каждого поколения:

- Вычисление адаптивной вероятности мутации

- Оценка приспособленности каждого расписания

- Сортировка популяции по приспособленности

- Отбор элитных особей

- Создание нового поколения через:

\* Кроссовер лучших особей (вероятность CROSSOVER\_RATE)

\* Случайный выбор из популяции

\* Мутацию потомков (адаптивная вероятность)

3. Остановка при:

- Достижении MAX\_GENERATIONS

- Отсутствии улучшений в течение MAX\_GENERATIONS\_WITHOUT\_IMPROVING

Returns:

list: Лучшее найденное расписание с максимальной оценкой fitness

"""

population = [generate\_random\_schedule() for \_ in range(POPULATION\_SIZE)]

best\_solution = None

best\_fitness = float("-inf")

generations\_without\_improvement = 0

best\_generation = -1

for generation in range(MAX\_GENERATIONS):

current\_mutation\_rate = adaptive\_mutation\_rate(generations\_without\_improvement)

fitness\_values = [fitness(schedule) for schedule in population]

population\_with\_fitness = list(zip(population, fitness\_values))

population\_with\_fitness.sort(key=lambda x: x[1], reverse=True)

current\_best, current\_fitness = population\_with\_fitness[0]

if current\_fitness > best\_fitness:

best\_solution = current\_best

best\_fitness = current\_fitness

generations\_without\_improvement = 0

best\_generation = generation

else:

generations\_without\_improvement += 1

next\_generation = [ind for ind, \_ in population\_with\_fitness[:ELITE\_SIZE]]

while len(next\_generation) < POPULATION\_SIZE:

if random.random() < CROSSOVER\_RATE:

parents = random.sample(population\_with\_fitness[:ELITE\_SIZE], 2)

child = two\_point\_crossover(parents[0][0], parents[1][0])

else:

child = random.choice(population\_with\_fitness)[0]

if random.random() < current\_mutation\_rate:

child = mutate(child)

next\_generation.append(child)

population = next\_generation

if generation % 50 == 0:

print(

f"Поколение {generation}: Лучший фитнес поколения = {current\_fitness}"

)

print(f"Поколений без улучшения: {generations\_without\_improvement}")

if generations\_without\_improvement >= MAX\_GENERATIONS\_WITHOUT\_IMPROVING:

print(

f"Досрочная остановка: нет улучшений в течение {MAX\_GENERATIONS\_WITHOUT\_IMPROVING} поколений"

)

break

print(f"Лучшее поколение {best\_generation}")

print(f"Лучший фитнесс {best\_fitness}")

return best\_solution

def generate\_schedule\_dataframe(schedule: list) -> pd.DataFrame:

"""

Создает DataFrame с расписанием в читаемом формате.

Формирует таблицу со следующей структурой:

- Индекс: номера водителей

- Колонки: тип смены и дни недели

- Ячейки содержат:

\* "Отдых" для выходных дней

\* "Автобус N (HH:00 - HH:00)" для рабочих смен

Args:

schedule (list): Расписание для отображения, содержащее информацию

о сменах водителей по дням недели

Returns:

pd.DataFrame: Отформатированная таблица с расписанием,

отсортированная по типу смены

"""

driver\_labels = [f"Водитель {d + 1}" for d in range(NUM\_DRIVERS)]

shifts = ["8hr" if d < NUM\_8HR\_DRIVERS else "12hr" for d in range(NUM\_DRIVERS)]

schedule\_df = pd.DataFrame(index=driver\_labels, columns=["Смена"] + DAYS\_OF\_WEEK)

schedule\_df["Смена"] = shifts

for day\_idx, daily\_schedule in enumerate(schedule):

for driver\_schedule in daily\_schedule:

driver = f"Водитель {driver\_schedule['driver'] + 1}"

if driver\_schedule["rest"]:

schedule\_df.loc[driver, DAYS\_OF\_WEEK[day\_idx]] = "Отдых"

else:

bus = driver\_schedule["bus"]

start = driver\_schedule["start"]

end = driver\_schedule["end"]

schedule\_df.loc[driver, DAYS\_OF\_WEEK[day\_idx]] = (

f"Автобус {bus + 1} ({start:02d}:00 - {end:02d}:00)"

)

return schedule\_df.sort\_values(by="Смена", ascending=False)

best\_schedule = genetic\_algorithm()

schedule\_df = generate\_schedule\_dataframe(best\_schedule)

print("\nЛучшее найденное расписание:")

schedule\_df

*# %%*

def save\_schedule\_to\_excel(

schedule\_df: pd.DataFrame, filename: str = "schedule\_genetic.xlsx"

) -> None:

"""

Сохраняет расписание в Excel файл с форматированием.

Применяет следующее форматирование:

- Центрирование текста в ячейках

- Тонкие границы для всех ячеек

- Серый фон (F0F0F0) для заголовков

- Светло-розовый фон (FFE4E1) для ячеек с "Отдых"

- Жирный шрифт размером 11pt для заголовков

- Автоматический подбор ширины столбцов

- Автоматический перенос текста

Args:

schedule\_df (pd.DataFrame): DataFrame с расписанием для сохранения

filename (str, optional): Имя выходного файла.

По умолчанию 'schedule\_genetic.xlsx'

Raises:

Exception: При возникновении ошибок в процессе сохранения

"""

try:

with pd.ExcelWriter(filename, engine="openpyxl") as writer:

schedule\_df.to\_excel(writer, sheet\_name="Расписание")

worksheet = writer.sheets["Расписание"]

border = Border(

left=Side(style="thin"),

right=Side(style="thin"),

top=Side(style="thin"),

bottom=Side(style="thin"),

)

header\_fill = PatternFill(

start\_color="F0F0F0", end\_color="F0F0F0", fill\_type="solid"

)

rest\_fill = PatternFill(

start\_color="FFE4E1", end\_color="FFE4E1", fill\_type="solid"

)

for row in worksheet.iter\_rows():

for cell in row:

cell.alignment = Alignment(

horizontal="center", vertical="center", wrap\_text=True

)

cell.border = border

if cell.row == 1:

cell.font = Font(size=11, bold=True)

cell.fill = header\_fill

elif cell.value == "Отдых":

cell.fill = rest\_fill

for column in worksheet.columns:

max\_length = 0

column\_letter = get\_column\_letter(column[0].column)

for cell in column:

try:

if len(str(cell.value)) > max\_length:

max\_length = len(str(cell.value))

except Exception as e:

print(e)

adjusted\_width = max\_length + 2

worksheet.column\_dimensions[column\_letter].width = adjusted\_width

print(f"\nРасписание успешно сохранено в файл: {filename}")

except Exception as e:

print(f"\nОшибка при сохранении файла: {str(e)}")

save\_schedule\_to\_excel(schedule\_df)