**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 «Математической кибернетики и информационных технологий»

Структуры и алгоритмы обработки данных

Курсовая работа на тему:

“Построение модели расписания автобусов и его подбор двумя методами. Их сравнение.”

Выполнил:

студент группы БВТ2203

Дворянчиков Даниил Денисович

Проверил:  
Симонов Сергей Евгеньевич

Оглавление

[**ВВЕДЕНИЕ** 3](#_Toc183639955)

[**Ход работы** 6](#_Toc183639956)

[**Создание симуляции.** 6](#_Toc183639957)

[**Ручное решение** 11](#_Toc183639958)

[**Запуск** 19](#_Toc183639959)

[**Вывод** 21](#_Toc183639960)

[**Полный код** 22](#_Toc183639961)

[**Список литературы** 32](#_Toc183639962)

# **ВВЕДЕНИЕ**

Организация эффективного расписания общественного транспорта является важной задачей, от которой зависит не только удобство пассажиров, но и экономическая эффективность транспортных компаний. Современные методы составления расписаний требуют учета множества факторов: времени поездок, частоты рейсов, загруженности маршрутов, доступности транспортных средств и других параметров. Кроме того, расписание должно быть гибким, чтобы адаптироваться к изменяющимся условиям, таким как рост пассажиропотока или изменение дорожной ситуации.

Целью данной курсовой работы является разработка модели расписания автобусов двумя способами: ручным подбором параметров и использованием генетического алгоритма. Первый подход позволяет понять принципы построения расписания и выявить основные ограничения и зависимости, а второй демонстрирует применение современных алгоритмов оптимизации для автоматизации этого процесса.

Генетические алгоритмы представляют собой мощный инструмент для решения задач, связанных с многокритериальной оптимизацией. Они имитируют естественный процесс эволюции, что позволяет находить решения, близкие к оптимальным, даже в условиях сложных и плохо структурированных задач. Использование генетического алгоритма в построении расписания позволяет учитывать множество параметров и находить более сбалансированные решения, чем традиционные методы.

В данной работе будет проведено сравнение двух подходов на основе ряда критериев, включая удобство использования, время разработки расписания и качество полученного результата. Результаты исследования могут быть полезны для транспортных компаний и специалистов в области управления логистикой.**Список используемых технологий**

Для реализации и анализа моделей расписания автобусов в рамках данной курсовой работы используются современные инструменты программирования и вычислительных платформ, а именно язык программирования Python и облачная среда Google Colab. Эти технологии обеспечивают удобство разработки, тестирования и визуализации решений, а также облегчают работу с большими объемами данных.

**1.1 Python**

Python является одним из наиболее популярных языков программирования благодаря его простоте, универсальности и богатой экосистеме библиотек. Для задач, связанных с разработкой расписаний и оптимизацией, Python предоставляет следующие возможности:

* **Библиотеки для научных вычислений и анализа данных**: библиотеки, такие как *NumPy*, *Pandas* и *SciPy*, позволяют эффективно работать с массивами данных, проводить статистические вычисления и анализировать результаты.
* **Инструменты визуализации**: библиотеки *Matplotlib* и *Seaborn* помогают создавать наглядные графики и визуализировать результаты работы алгоритмов.
* **Модули для работы с алгоритмами оптимизации**: для реализации генетического алгоритма можно использовать библиотеки, такие как *DEAP* или писать пользовательские решения с нуля.

**1.2 Google Colab**

Google Colab (Google Collaboratory) — это облачная среда для работы с Python, предоставляющая мощный инструмент для выполнения вычислений без необходимости настраивать локальное рабочее пространство. Основные преимущества Google Colab:

* **Облачные вычисления**: выполнение кода происходит на удалённых серверах Google, что позволяет использовать мощные вычислительные ресурсы без затрат на оборудование.
* **Удобство работы**: поддержка Jupyter Notebook формата делает работу с кодом и результатами его выполнения интуитивно понятной и удобной.
* **Доступность библиотек**: Colab предоставляет доступ к предустановленным библиотекам Python, что ускоряет процесс разработки.
* **Интеграция с Google Drive**: возможность сохранять проекты и данные в облаке упрощает совместную работу и управление файлами.

Использование Python и Google Colab в рамках данной курсовой работы позволяет реализовать сложные модели оптимизации, тестировать их на различных данных и быстро анализировать результаты. Эти инструменты предоставляют гибкость в разработке, что делает их идеальным выбором для задач построения расписания автобусов.

# **Ход работы**

## **Создание симуляции.**

1) Создадим класс водителя. В конструкторе будет его айди рабачие часы и часы перерывов. Так же добавим метод который проверяет, доступен ли водитель в указанный момент времени.

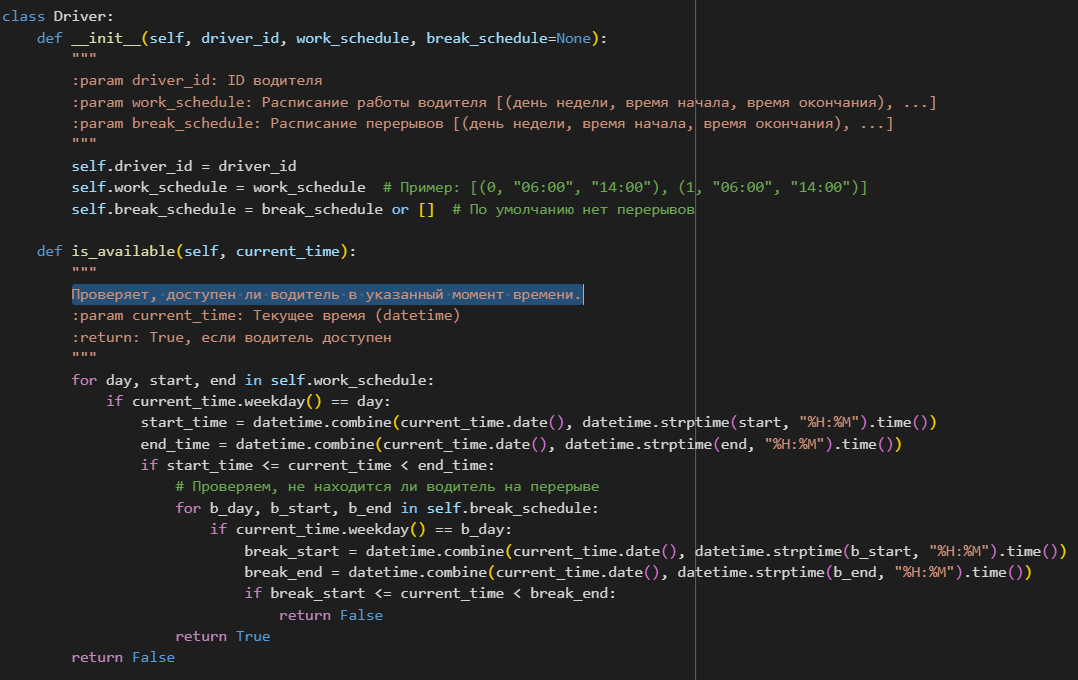


Рис. 1 – Класс водителя

2) Теперь создадим класс автобуса. В конструкторе будем инициализировать его айди маршрут вместимость водителя текущее колличество пассажиров и флаг который будет сообщать о текущем состоянии автобуса(работает или стоит).

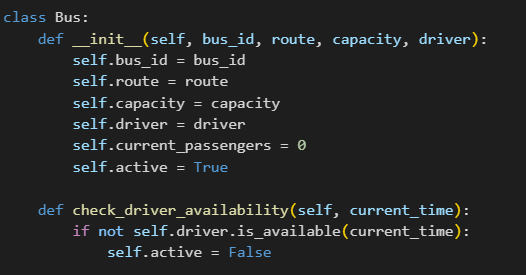


Рис. 2 – Класс автобуса

3) Для удобства добавим класс расписания. В конструкторе инициализируем время начала и конца симуляции текущее время время шага симуляции(поставим одну минуту) часы пик список остановок автобусов водителей общее колличество пассажиров число неперевезённных пассажиров и лог для составления расписания.

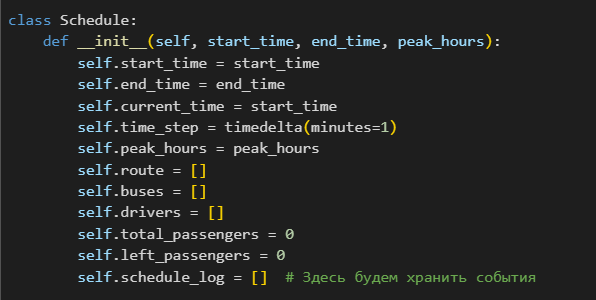


Рис. 3 – Конструктор расписания

4)Реализуем методы добавления остановок автобусов и водителей(перед запуском симуляции нужно будет описать её при помощи этих методов)

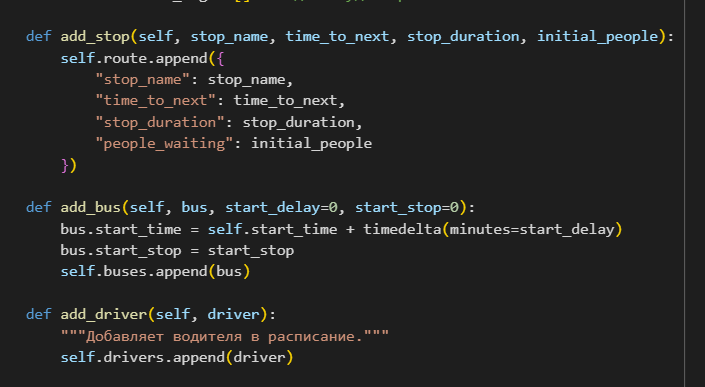


Рис. 3 – Методы добавления информации для симуляции

5) Также добавим вспомогательные методы чтобы улучшить читаемость кода и избежать дублирования.(поиск подходящего водителя определения час пик ли сейчас определение ночное время ли сейчас)

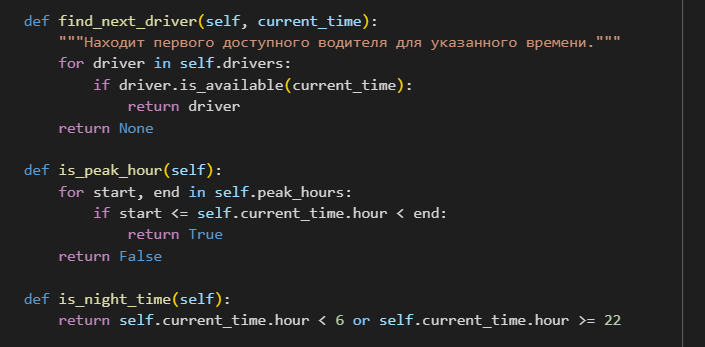


Рис.4 – Вспомогательные методы

6) Запуск симуляции. Заполним изначальное расположение автобусов и их время.

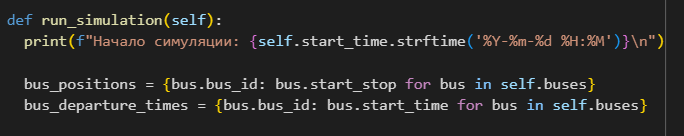


Рис.5 – Заполнение основных параметров

7) Запускаем цикл симуляции. Сначала будем выводить новый день, если он настал. Затем будем добавлять пассажиров на остановки в рандомном количестве и в зависимости от времени суток.

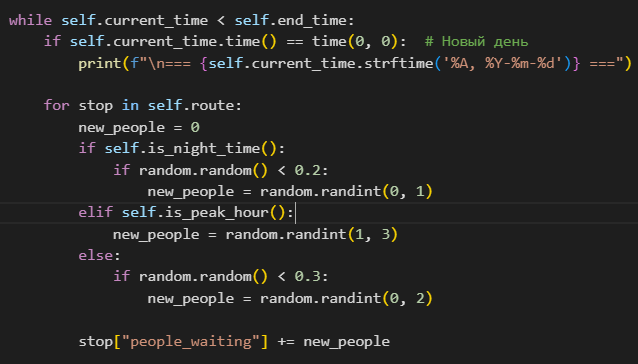


Рис.6 – Добавление пассажиров

8) Затем будем перебирать все автобусы и переназначать водителя, если он не активен(автобус). Затем проверяем не вышло ли время водителя. Если вышло – убираем автобус и переходим к следующему.

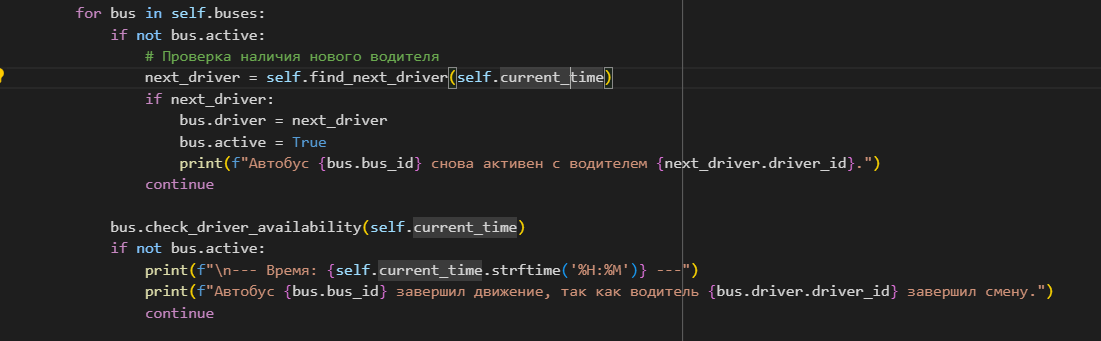


Рис.7 – Проверка состояния автобусов и водителей.

9) Далее в этом же цикле проверяем где находится автобус и получаем данные по остановке. Для вывода расписания логируем прибытие. В зависимости от вместимости подбираем пассажиров и высаживаем старых.

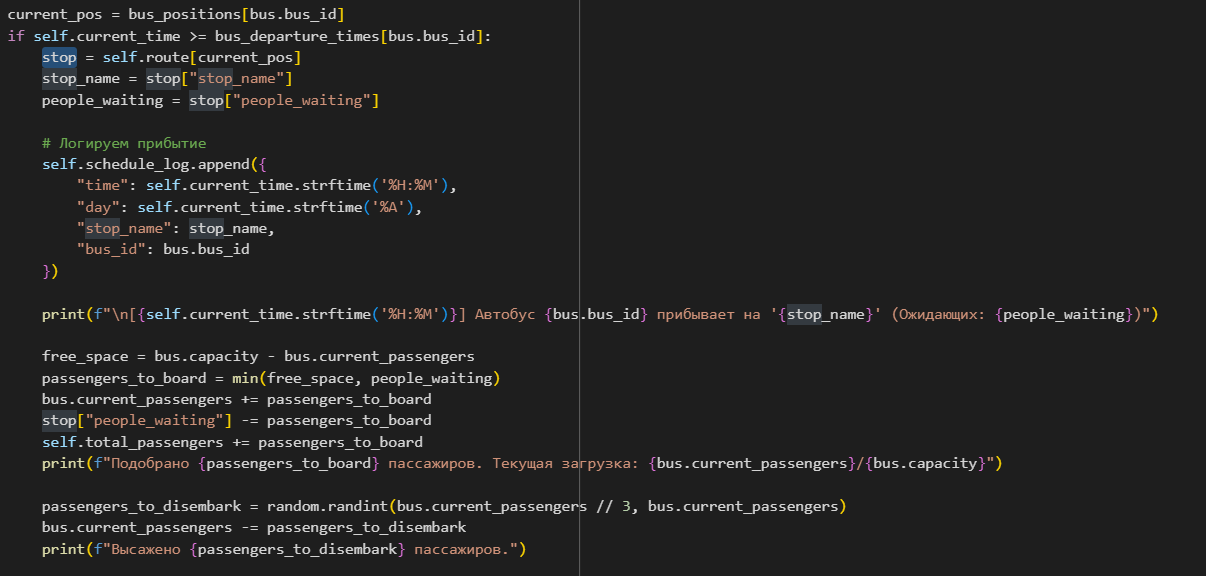


Рис.8 – Прибытие на остановку

10) Передвигаем автобус с остановки.

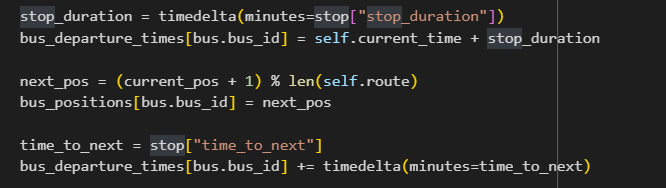


Рис.9 – Отправка автобуса в путь

11) Переходим на следующий шаг и после основного цикла завершаем симуляцию.

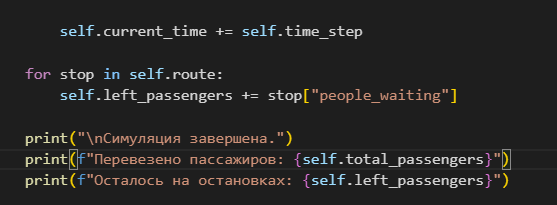


Рис.10 – Завершение симуляции

12) В конце добавим метод вывода расписания, на основе логов

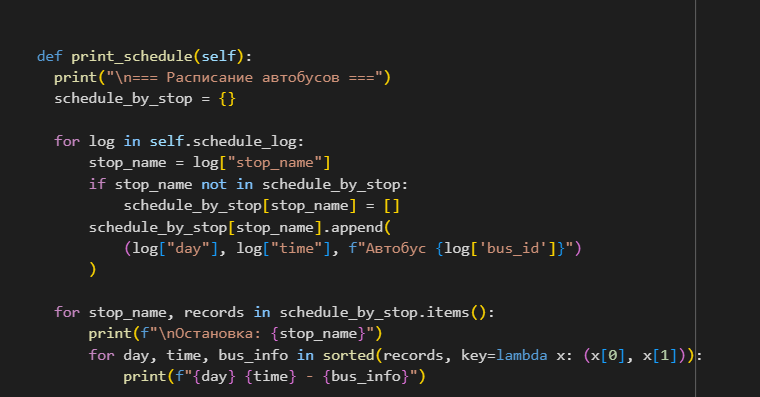


Рис.11 – Вывод расписания

## **Ручное решение**

1. Пусть симуляция будет начинаться с понедельника. Определим часы пик и остановки.

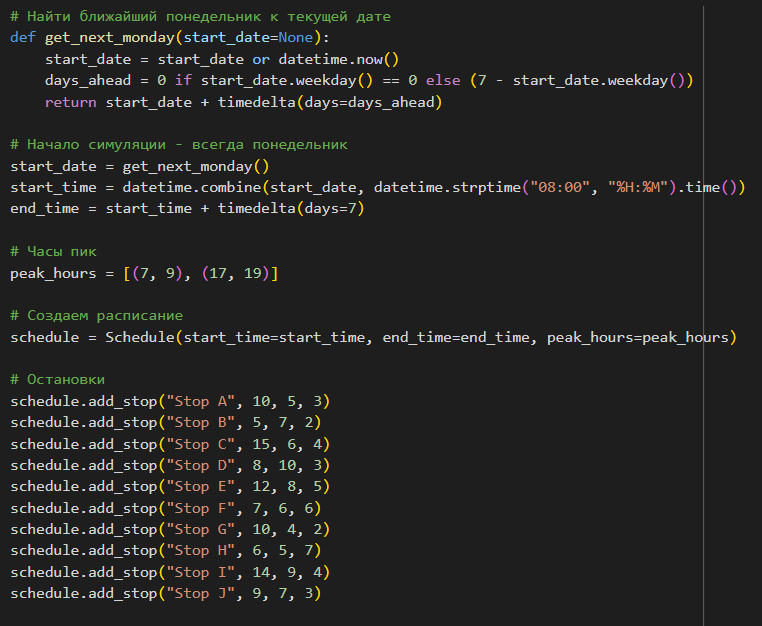


Рис.12 – Заполнение остановок

2)Затем заполним водителей и автобусы.

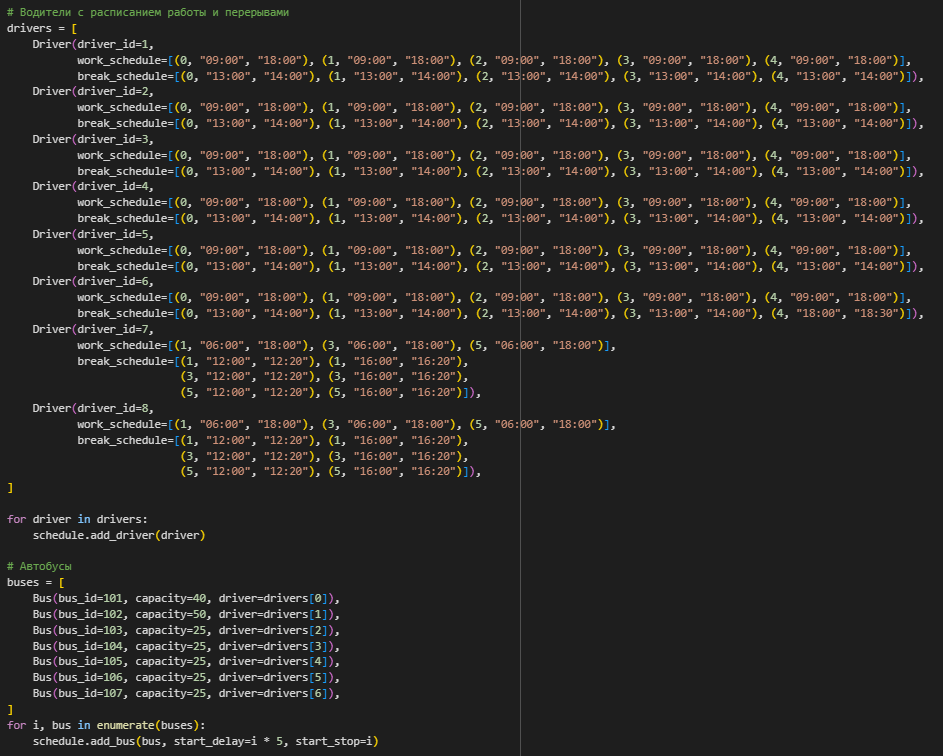


Рис.13 – Добавление водителей и автобусов

3) Запустим симуляцию.



Рис.14 – Результат запуска.

4) После запуска можно вывести расписание для остановок.

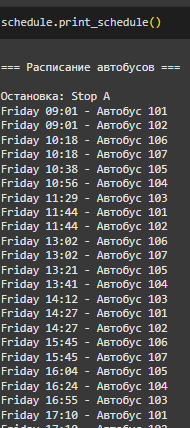


Рис.15 – Вывод расписания для каждой остановки

**Генетический алгоритм**

1. Для начала создадим константы типов водителей.

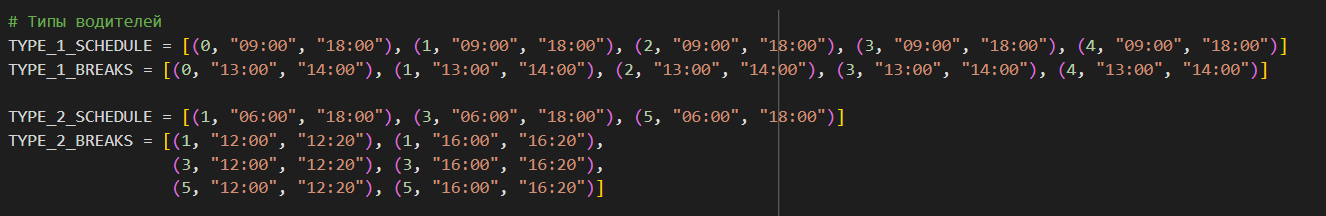


Рис.16 – Типы водителей

1. Создадим класс генетического алгоритма с параметрами конструктора ниже.

num\_generations: количество поколений, через которое будет происходить эволюция популяции.

population\_size: количество индивидов (решений) в каждой популяции.

mutation\_rate: вероятность мутации для каждого индивида.

population: список текущих индивидов в популяции.

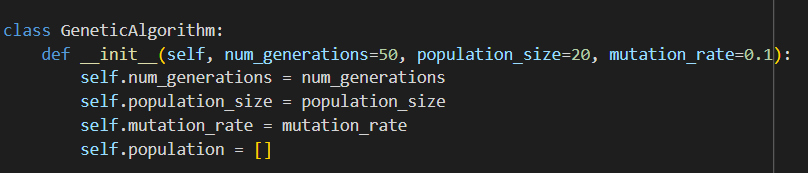


Рис.17 – Класс генетического алгоритма

1. Будем делать так: Каждый раз будем создавать новый экземпляр расписания. Остановки и основные параметры будут одинаковы, но водителей каждый раз будем назначать других. Затем будем запускать симуляцию и оценивать её по количеству оставшихся на остановках пассажирах(left\_passengers).

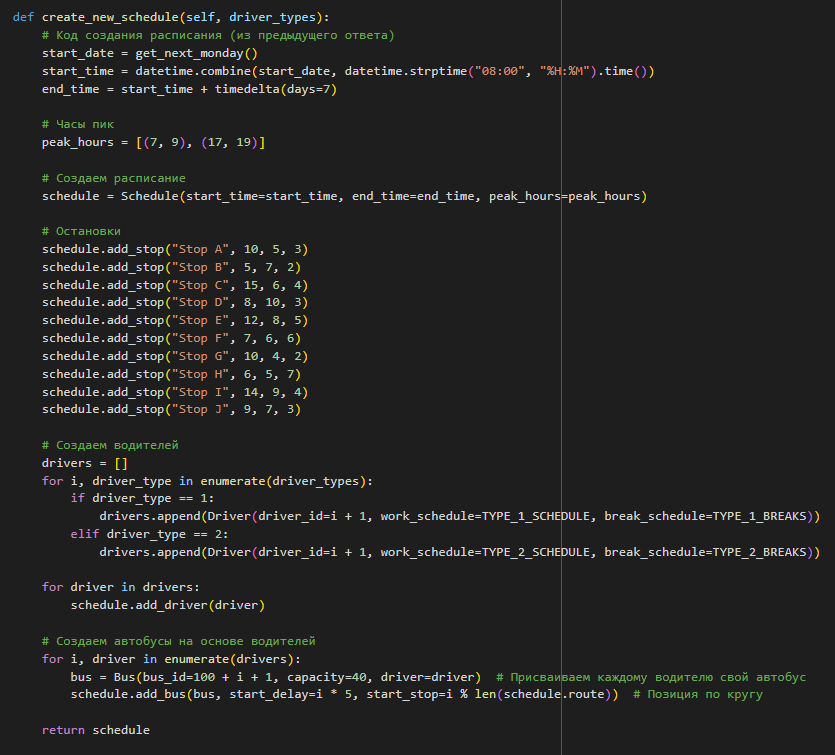


Рис.18 – Метод для создания нового расписания в классе генетического алгоритма

1. Теперь создадим метода инициализации популяции. Для каждого индивида в популяции создаётся список из 8 элементов, где каждый элемент представляет тип водителя (1 или 2). Типы выбираются случайным образом с равной вероятностью. Получившийся список driver\_types добавляется в популяцию.

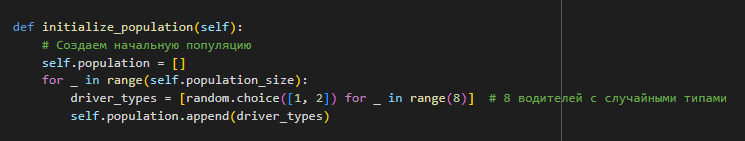


Рис.19 – Инициализация популяции

1. Затем метод для оценки приспособленности. Создаётся расписание на основе типов водителей. Запускается симуляция расписания. После симуляции возвращается количество пассажиров, которые остались на остановках(left\_passengers).

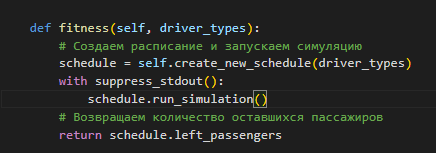


Рис.20 – Оценка приспособленности

1. Примечание: При вызове run\_simulation выводится информация о движении автобусов и пассажиров. Нам это не нужно в генетическом алгоритме. Метод suppress\_stdout() используется для подавления вывода симуляции в консоль, чтобы избежать засорения логов.

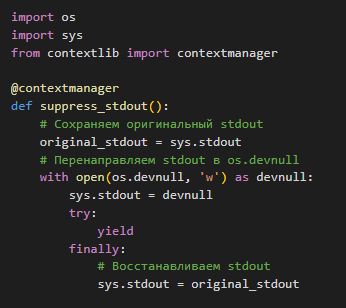


Рис.21 – Перенаправление вывода

1. Отбор родителей.

Приспособленность: Рассчитывается как 1 / (1 + score), где score — количество оставшихся пассажиров. Это преобразование делает приспособленность обратной величине ошибки: меньшее количество score соответствует большей приспособленности. Вероятности выбора: Каждому индивиду предоставляется вероятность пропорционально его приспособленности. Выбор: Используется random.choices для случайного выбора двух родителей с заданными вероятностями.

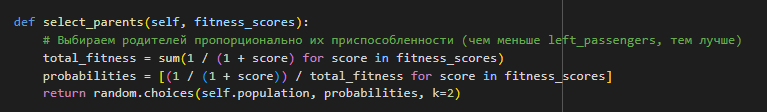


Рис.22 – Отбор родителей

1. Скрещивание. Выбирается случайная точка point в геноме (индексе) родителей. Потомок 1: берёт часть генома от parent1 до точки point, затем часть от parent2. Потомок 2: аналогично, но наоборот.

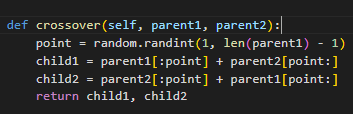


Рис.23 – Скрещивание

1. Мутация. С вероятностью mutation\_rate выбирается один случайный водитель (индекс в списке). Тип водителя меняется на противоположный: если был 1, становится 2, и наоборот.

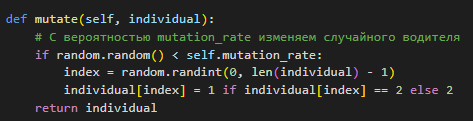


Рис.24 – Мутация

## **Запуск**

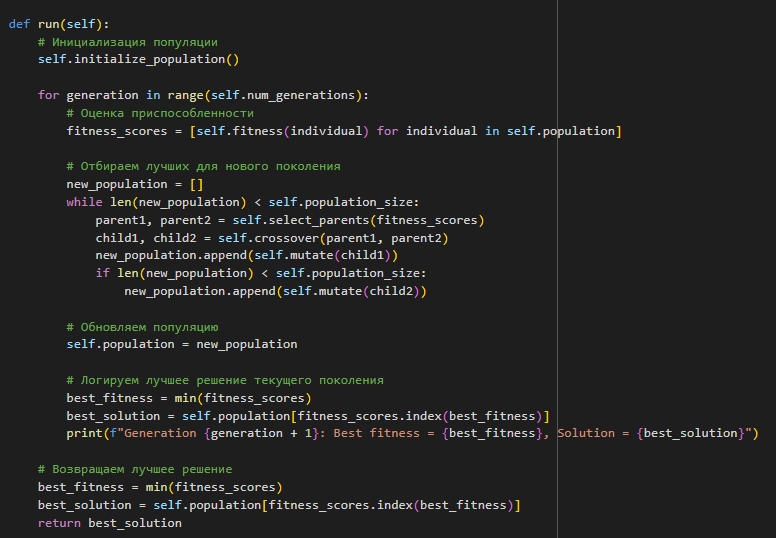


Рис.25 – Метод запуска генетического алгоритма

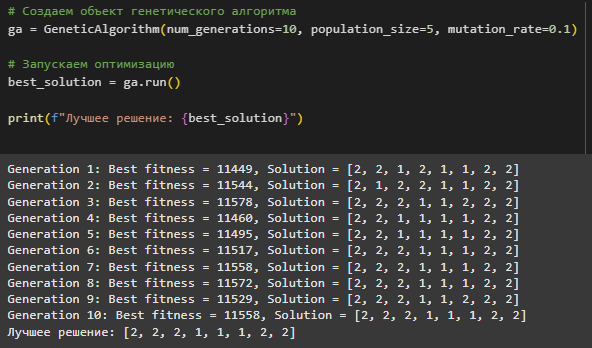


Рис.26 – Запуск и результат генетического алгоритма

**Итоги**

Алгоритм выдал лучшее решение в виде списка из 8 водителей с указанием их типа. В рамках симуляции алгоритм работает достаточно медленно и, например, 100 поколений смысла делать нет. Так же второй тип водителей работает только в определенные дни. Если добавить перемешивание их рабочих дней, то это займет ещё больше времени. В целом я реализовал этот алгоритм больше в целях ознакомления для его применения в прикладных целях.

# **Вывод**

В ходе работы я реализовал генетический алгоритм для оптимизации расписания водителей. Я разработал метод инициализации популяции, который создает начальный набор индивидов с случайными типами водителей. Для оценки пригодности каждого индивида я использовал функцию fitness, которая измеряет качество решения через количество оставшихся пассажиров после симуляции.

Далее я внедрил механизм выбора родителей на основе их пригодности, чтобы лучшие индивиды имели больше шансов передать свои гены следующему поколению. Для создания потомков я реализовал одноточечный кроссовер, который комбинирует характеристики родителей. Также я добавил механизм мутации, позволяющий изменять случайные гены с заданной вероятностью, что способствует сохранению разнообразия популяции и предотвращает преждевременную сходимость алгоритма.

В результате работы алгоритм способен эволюционировать популяцию индивидов через несколько поколений, стремясь к нахождению оптимального решения задачи.

# **Полный код**

from datetime import datetime, timedelta, time

import random

class Driver:

    def \_\_init\_\_(self, driver\_id, work\_schedule, break\_schedule=None):

        """

        :param driver\_id: ID водителя

        :param work\_schedule: Расписание работы водителя [(день недели, время начала, время окончания), ...]

        :param break\_schedule: Расписание перерывов [(день недели, время начала, время окончания), ...]

        """

        self.driver\_id = driver\_id

        self.work\_schedule = work\_schedule

        self.break\_schedule = break\_schedule or []  # По умолчанию нет перерывов

    def is\_available(self, current\_time):

        """

        Проверяет, доступен ли водитель в указанный момент времени.

        :param current\_time: Текущее время (datetime)

        :return: True, если водитель доступен

        """

        for day, start, end in self.work\_schedule:

            if current\_time.weekday() == day:

                start\_time = datetime.combine(current\_time.date(), datetime.strptime(start, "%H:%M").time())

                end\_time = datetime.combine(current\_time.date(), datetime.strptime(end, "%H:%M").time())

                if start\_time <= current\_time < end\_time:

                    # Проверяем, не находится ли водитель на любом из перерывов

                    for b\_day, b\_start, b\_end in self.break\_schedule:

                        if current\_time.weekday() == b\_day:

                            break\_start = datetime.combine(current\_time.date(), datetime.strptime(b\_start, "%H:%M").time())

                            break\_end = datetime.combine(current\_time.date(), datetime.strptime(b\_end, "%H:%M").time())

                            if break\_start <= current\_time < break\_end:

                                return False

                    return True

        return False

class Bus:

    def \_\_init\_\_(self, bus\_id, capacity, driver):

        self.bus\_id = bus\_id

        self.capacity = capacity

        self.driver = driver

        self.current\_passengers = 0

        self.active = True

    def check\_driver\_availability(self, current\_time):

        if self.driver is None:

            self.active = False

            return

        if not self.driver.is\_available(current\_time):

            self.active = False

class Schedule:

    def \_\_init\_\_(self, start\_time, end\_time, peak\_hours):

        self.start\_time = start\_time

        self.end\_time = end\_time

        self.current\_time = start\_time

        self.time\_step = timedelta(minutes=1)

        self.peak\_hours = peak\_hours

        self.route = []

        self.buses = []

        self.drivers = []

        self.total\_passengers = 0

        self.left\_passengers = 0

        self.schedule\_log = []  # Здесь будем хранить события

    def add\_stop(self, stop\_name, time\_to\_next, stop\_duration, initial\_people):

        self.route.append({

            "stop\_name": stop\_name,

            "time\_to\_next": time\_to\_next,

            "stop\_duration": stop\_duration,

            "people\_waiting": initial\_people

        })

    def add\_bus(self, bus, start\_delay=0, start\_stop=0):

        bus.start\_time = self.start\_time + timedelta(minutes=start\_delay)

        bus.start\_stop = start\_stop

        self.buses.append(bus)

    def add\_driver(self, driver):

        """Добавляет водителя в расписание."""

        self.drivers.append(driver)

    def find\_next\_driver(self, current\_time):

        """Находит первого доступного водителя для указанного времени."""

        for driver in self.drivers:

            if driver.is\_available(current\_time):

                return driver

        return None

    def is\_peak\_hour(self):

        for start, end in self.peak\_hours:

            if start <= self.current\_time.hour < end:

                return True

        return False

    def is\_night\_time(self):

        return self.current\_time.hour < 6 or self.current\_time.hour >= 22

    def run\_simulation(self):

      print(f"Начало симуляции: {self.start\_time.strftime('%Y-%m-%d %H:%M')}\n")

      bus\_positions = {bus.bus\_id: bus.start\_stop for bus in self.buses}

      bus\_departure\_times = {bus.bus\_id: bus.start\_time for bus in self.buses}

      while self.current\_time < self.end\_time:

          if self.current\_time.time() == time(0, 0):  # Новый день

              print(f"\n=== {self.current\_time.strftime('%A, %Y-%m-%d')} ===")

          for stop in self.route:

              new\_people = 0

              if self.is\_night\_time():

                  if random.random() < 0.2:

                      new\_people = random.randint(0, 1)

              elif self.is\_peak\_hour():

                  new\_people = random.randint(1, 3)

              else:

                  if random.random() < 0.3:

                      new\_people = random.randint(0, 2)

              stop["people\_waiting"] += new\_people

          for bus in self.buses:

              if not bus.active:

                  # Проверка наличия нового водителя

                  next\_driver = self.find\_next\_driver(self.current\_time)

                  if next\_driver:

                      bus.driver = next\_driver

                      bus.active = True

                      print(f"Автобус {bus.bus\_id} снова активен с водителем {next\_driver.driver\_id}.")

                  continue

              bus.check\_driver\_availability(self.current\_time)

              if not bus.active:

                  print(f"\n--- Время: {self.current\_time.strftime('%H:%M')} ---")

                  print(f"Автобус {bus.bus\_id} завершил движение, так как водитель {bus.driver.driver\_id} завершил смену.")

                  continue

              current\_pos = bus\_positions[bus.bus\_id]

              if self.current\_time >= bus\_departure\_times[bus.bus\_id]:

                  stop = self.route[current\_pos]

                  stop\_name = stop["stop\_name"]

                  people\_waiting = stop["people\_waiting"]

                  # Логируем прибытие

                  self.schedule\_log.append({

                      "time": self.current\_time.strftime('%H:%M'),

                      "day": self.current\_time.strftime('%A'),

                      "stop\_name": stop\_name,

                      "bus\_id": bus.bus\_id

                  })

                  print(f"\n[{self.current\_time.strftime('%H:%M')}] Автобус {bus.bus\_id} прибывает на '{stop\_name}' (Ожидающих: {people\_waiting})")

                  free\_space = bus.capacity - bus.current\_passengers

                  passengers\_to\_board = min(free\_space, people\_waiting)

                  bus.current\_passengers += passengers\_to\_board

                  stop["people\_waiting"] -= passengers\_to\_board

                  self.total\_passengers += passengers\_to\_board

                  print(f"Подобрано {passengers\_to\_board} пассажиров. Текущая загрузка: {bus.current\_passengers}/{bus.capacity}")

                  passengers\_to\_disembark = random.randint(bus.current\_passengers // 3, bus.current\_passengers)

                  bus.current\_passengers -= passengers\_to\_disembark

                  print(f"Высажено {passengers\_to\_disembark} пассажиров.")

                  stop\_duration = timedelta(minutes=stop["stop\_duration"])

                  bus\_departure\_times[bus.bus\_id] = self.current\_time + stop\_duration

                  next\_pos = (current\_pos + 1) % len(self.route)

                  bus\_positions[bus.bus\_id] = next\_pos

                  time\_to\_next = stop["time\_to\_next"]

                  bus\_departure\_times[bus.bus\_id] += timedelta(minutes=time\_to\_next)

          self.current\_time += self.time\_step

      for stop in self.route:

          self.left\_passengers += stop["people\_waiting"]

      print("\nСимуляция завершена.")

      print(f"Перевезено пассажиров: {self.total\_passengers}")

      print(f"Осталось на остановках: {self.left\_passengers}")

    def print\_schedule(self):

      print("\n=== Расписание автобусов ===")

      schedule\_by\_stop = {}

      for log in self.schedule\_log:

          stop\_name = log["stop\_name"]

          if stop\_name not in schedule\_by\_stop:

              schedule\_by\_stop[stop\_name] = []

          schedule\_by\_stop[stop\_name].append(

              (log["day"], log["time"], f"Автобус {log['bus\_id']}")

          )

      for stop\_name, records in schedule\_by\_stop.items():

          print(f"\nОстановка: {stop\_name}")

          for day, time, bus\_info in sorted(records, key=lambda x: (x[0], x[1])):

              print(f"{day} {time} - {bus\_info}")

# Найти ближайший понедельник к текущей дате

def get\_next\_monday(start\_date=None):

    start\_date = start\_date or datetime.now()

    days\_ahead = 0 if start\_date.weekday() == 0 else (7 - start\_date.weekday())

    return start\_date + timedelta(days=days\_ahead)

# Начало симуляции - всегда понедельник

start\_date = get\_next\_monday()

start\_time = datetime.combine(start\_date, datetime.strptime("08:00", "%H:%M").time())

end\_time = start\_time + timedelta(days=7)

# Часы пик

peak\_hours = [(7, 9), (17, 19)]

# Создаем расписание

schedule = Schedule(start\_time=start\_time, end\_time=end\_time, peak\_hours=peak\_hours)

# Остановки

schedule.add\_stop("Stop A", 10, 5, 3)

schedule.add\_stop("Stop B", 5, 7, 2)

schedule.add\_stop("Stop C", 15, 6, 4)

schedule.add\_stop("Stop D", 8, 10, 3)

schedule.add\_stop("Stop E", 12, 8, 5)

schedule.add\_stop("Stop F", 7, 6, 6)

schedule.add\_stop("Stop G", 10, 4, 2)

schedule.add\_stop("Stop H", 6, 5, 7)

schedule.add\_stop("Stop I", 14, 9, 4)

schedule.add\_stop("Stop J", 9, 7, 3)

# Водители с расписанием работы и перерывами

drivers = [

    Driver(driver\_id=1,

           work\_schedule=[(0, "09:00", "18:00"), (1, "09:00", "18:00"), (2, "09:00", "18:00"), (3, "09:00", "18:00"), (4, "09:00", "18:00")],

           break\_schedule=[(0, "13:00", "14:00"), (1, "13:00", "14:00"), (2, "13:00", "14:00"), (3, "13:00", "14:00"), (4, "13:00", "14:00")]),

    Driver(driver\_id=2,

           work\_schedule=[(0, "09:00", "18:00"), (1, "09:00", "18:00"), (2, "09:00", "18:00"), (3, "09:00", "18:00"), (4, "09:00", "18:00")],

           break\_schedule=[(0, "13:00", "14:00"), (1, "13:00", "14:00"), (2, "13:00", "14:00"), (3, "13:00", "14:00"), (4, "13:00", "14:00")]),

    Driver(driver\_id=3,

           work\_schedule=[(0, "09:00", "18:00"), (1, "09:00", "18:00"), (2, "09:00", "18:00"), (3, "09:00", "18:00"), (4, "09:00", "18:00")],

           break\_schedule=[(0, "13:00", "14:00"), (1, "13:00", "14:00"), (2, "13:00", "14:00"), (3, "13:00", "14:00"), (4, "13:00", "14:00")]),

    Driver(driver\_id=4,

           work\_schedule=[(0, "09:00", "18:00"), (1, "09:00", "18:00"), (2, "09:00", "18:00"), (3, "09:00", "18:00"), (4, "09:00", "18:00")],

           break\_schedule=[(0, "13:00", "14:00"), (1, "13:00", "14:00"), (2, "13:00", "14:00"), (3, "13:00", "14:00"), (4, "13:00", "14:00")]),

    Driver(driver\_id=5,

           work\_schedule=[(0, "09:00", "18:00"), (1, "09:00", "18:00"), (2, "09:00", "18:00"), (3, "09:00", "18:00"), (4, "09:00", "18:00")],

           break\_schedule=[(0, "13:00", "14:00"), (1, "13:00", "14:00"), (2, "13:00", "14:00"), (3, "13:00", "14:00"), (4, "13:00", "14:00")]),

    Driver(driver\_id=6,

           work\_schedule=[(0, "09:00", "18:00"), (1, "09:00", "18:00"), (2, "09:00", "18:00"), (3, "09:00", "18:00"), (4, "09:00", "18:00")],

           break\_schedule=[(0, "13:00", "14:00"), (1, "13:00", "14:00"), (2, "13:00", "14:00"), (3, "13:00", "14:00"), (4, "18:00", "18:30")]),

    Driver(driver\_id=7,

           work\_schedule=[(1, "06:00", "18:00"), (3, "06:00", "18:00"), (5, "06:00", "18:00")],

           break\_schedule=[(1, "12:00", "12:20"), (1, "16:00", "16:20"),

                           (3, "12:00", "12:20"), (3, "16:00", "16:20"),

                           (5, "12:00", "12:20"), (5, "16:00", "16:20")]),

    Driver(driver\_id=8,

           work\_schedule=[(1, "06:00", "18:00"), (3, "06:00", "18:00"), (5, "06:00", "18:00")],

           break\_schedule=[(1, "12:00", "12:20"), (1, "16:00", "16:20"),

                           (3, "12:00", "12:20"), (3, "16:00", "16:20"),

                           (5, "12:00", "12:20"), (5, "16:00", "16:20")]),

]

for driver in drivers:

    schedule.add\_driver(driver)

# Автобусы

buses = [

    Bus(bus\_id=101, capacity=40, driver=drivers[0]),

    Bus(bus\_id=102, capacity=50, driver=drivers[1]),

    Bus(bus\_id=103, capacity=25, driver=drivers[2]),

    Bus(bus\_id=104, capacity=25, driver=drivers[3]),

    Bus(bus\_id=105, capacity=25, driver=drivers[4]),

    Bus(bus\_id=106, capacity=25, driver=drivers[5]),

    Bus(bus\_id=107, capacity=25, driver=drivers[6]),

]

for i, bus in enumerate(buses):

    schedule.add\_bus(bus, start\_delay=i \* 5, start\_stop=i)

# Запуск симуляции

schedule.run\_simulation()

schedule.print\_schedule()

import os

import sys

from contextlib import contextmanager

@contextmanager

def suppress\_stdout():

    # Сохраняем оригинальный stdout

    original\_stdout = sys.stdout

    # Перенаправляем stdout в os.devnull

    with open(os.devnull, 'w') as devnull:

        sys.stdout = devnull

        try:

            yield

        finally:

            # Восстанавливаем stdout

            sys.stdout = original\_stdout

# Типы водителей

TYPE\_1\_SCHEDULE = [(0, "09:00", "18:00"), (1, "09:00", "18:00"), (2, "09:00", "18:00"), (3, "09:00", "18:00"), (4, "09:00", "18:00")]

TYPE\_1\_BREAKS = [(0, "13:00", "14:00"), (1, "13:00", "14:00"), (2, "13:00", "14:00"), (3, "13:00", "14:00"), (4, "13:00", "14:00")]

TYPE\_2\_SCHEDULE = [(1, "06:00", "18:00"), (3, "06:00", "18:00"), (5, "06:00", "18:00")]

TYPE\_2\_BREAKS = [(1, "12:00", "12:20"), (1, "16:00", "16:20"),

                 (3, "12:00", "12:20"), (3, "16:00", "16:20"),

                 (5, "12:00", "12:20"), (5, "16:00", "16:20")]

class GeneticAlgorithm:

    def \_\_init\_\_(self, num\_generations=50, population\_size=20, mutation\_rate=0.1):

        self.num\_generations = num\_generations

        self.population\_size = population\_size

        self.mutation\_rate = mutation\_rate

        self.population = []

    def initialize\_population(self):

        # Создаем начальную популяцию

        self.population = []

        for \_ in range(self.population\_size):

            driver\_types = [random.choice([1, 2]) for \_ in range(8)]  # 8 водителей с случайными типами

            self.population.append(driver\_types)

    def fitness(self, driver\_types):

        # Создаем расписание и запускаем симуляцию

        schedule = self.create\_new\_schedule(driver\_types)

        with suppress\_stdout():

            schedule.run\_simulation()

        # Возвращаем количество оставшихся пассажиров

        return schedule.left\_passengers

    def select\_parents(self, fitness\_scores):

        # Выбираем родителей пропорционально их приспособленности (чем меньше left\_passengers, тем лучше)

        total\_fitness = sum(1 / (1 + score) for score in fitness\_scores)

        probabilities = [(1 / (1 + score)) / total\_fitness for score in fitness\_scores]

        return random.choices(self.population, probabilities, k=2)

    def crossover(self, parent1, parent2):

        point = random.randint(1, len(parent1) - 1)

        child1 = parent1[:point] + parent2[point:]

        child2 = parent2[:point] + parent1[point:]

        return child1, child2

    def mutate(self, individual):

        # С вероятностью mutation\_rate изменяем случайного водителя

        if random.random() < self.mutation\_rate:

            index = random.randint(0, len(individual) - 1)

            individual[index] = 1 if individual[index] == 2 else 2

        return individual

    def create\_new\_schedule(self, driver\_types):

        # Код создания расписания (из предыдущего ответа)

        start\_date = get\_next\_monday()

        start\_time = datetime.combine(start\_date, datetime.strptime("08:00", "%H:%M").time())

        end\_time = start\_time + timedelta(days=7)

        # Часы пик

        peak\_hours = [(7, 9), (17, 19)]

        # Создаем расписание

        schedule = Schedule(start\_time=start\_time, end\_time=end\_time, peak\_hours=peak\_hours)

        # Остановки

        schedule.add\_stop("Stop A", 10, 5, 3)

        schedule.add\_stop("Stop B", 5, 7, 2)

        schedule.add\_stop("Stop C", 15, 6, 4)

        schedule.add\_stop("Stop D", 8, 10, 3)

        schedule.add\_stop("Stop E", 12, 8, 5)

        schedule.add\_stop("Stop F", 7, 6, 6)

        schedule.add\_stop("Stop G", 10, 4, 2)

        schedule.add\_stop("Stop H", 6, 5, 7)

        schedule.add\_stop("Stop I", 14, 9, 4)

        schedule.add\_stop("Stop J", 9, 7, 3)

        # Создаем водителей

        drivers = []

        for i, driver\_type in enumerate(driver\_types):

            if driver\_type == 1:

                drivers.append(Driver(driver\_id=i + 1, work\_schedule=TYPE\_1\_SCHEDULE, break\_schedule=TYPE\_1\_BREAKS))

            elif driver\_type == 2:

                drivers.append(Driver(driver\_id=i + 1, work\_schedule=TYPE\_2\_SCHEDULE, break\_schedule=TYPE\_2\_BREAKS))

        for driver in drivers:

            schedule.add\_driver(driver)

        # Создаем автобусы на основе водителей

        for i, driver in enumerate(drivers):

            bus = Bus(bus\_id=100 + i + 1, capacity=40, driver=driver)  # Присваиваем каждому водителю свой автобус

            schedule.add\_bus(bus, start\_delay=i \* 5, start\_stop=i % len(schedule.route))  # Позиция по кругу

        return schedule

    def run(self):

        # Инициализация популяции

        self.initialize\_population()

        for generation in range(self.num\_generations):

            # Оценка приспособленности

            fitness\_scores = [self.fitness(individual) for individual in self.population]

            # Отбираем лучших для нового поколения

            new\_population = []

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

                parent1, parent2 = self.select\_parents(fitness\_scores)

                child1, child2 = self.crossover(parent1, parent2)

                new\_population.append(self.mutate(child1))

                if len(new\_population) < self.population\_size:

                    new\_population.append(self.mutate(child2))

            # Обновляем популяцию

            self.population = new\_population

            # Логируем лучшее решение текущего поколения

            best\_fitness = min(fitness\_scores)

            best\_solution = self.population[fitness\_scores.index(best\_fitness)]

            print(f"Generation {generation + 1}: Best fitness = {best\_fitness}, Solution = {best\_solution}")

        # Возвращаем лучшее решение

        best\_fitness = min(fitness\_scores)

        best\_solution = self.population[fitness\_scores.index(best\_fitness)]

        return best\_solution

# Создаем объект генетического алгоритма

ga = GeneticAlgorithm(num\_generations=10, population\_size=5, mutation\_rate=0.1)

# Запускаем оптимизацию

best\_solution = ga.run()

print(f"Лучшее решение: {best\_solution}")

# **Список литературы**

- <https://docs.python.org/3/>

- <https://habr.com/ru/articles/128704/>