МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное учреждение высшего образования

«Московский технический университет связи и информатики»

ФАКУЛЬТЕТ ИНФОРМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Курсовая работа

Выполнила

Живодров Иван Владимирович БВТ2205

https://github.com/HSgivane/SIAOD_kurs_rabota

Преподаватель

Симонов Сергей Евгеньевич

Москва, 2024

Оглавление

Введение

Задача о расписание:

Смысл данной работы заключается в разработке оптимальной системы планирования водителей автобусов с учетом следующих задач:

1. Эффективное распределение ресурсов:

1. Минимизация количества водителей, необходимых для покрытия всех маршрутов.

2. Учёт специфики типов водителей:

Тип А: Ограничен 8-часовой сменой, после которой требуется замена.

Тип В: Работает на протяжении всей смены, что требует минимального количества водителей.

2. Удовлетворение транспортного спроса:

1. Обеспечение соответствия количества работающих автобусов нагрузке в часы пик и вне часов пик (70% и 30% от общего числа, соответственно).

2. Гарантия, что маршруты выполняются бесперебойно, даже если нагрузка изменяется в течение дня.

3. Повышение эффективности планирования за счёт адаптации алгоритма под разные типы водителей.

3. Оптимизация через алгоритмы:

1. Использование генетического подхода для минимизации числа водителей, распределения их на маршруты и соблюдения ограничений их работы.

2. Метод «в лоб». Использование самостоятельно написанной функции для распределения расписания автобусов и сравнение с генетическим подходом.

4. Практическая реализация:

1. Программа предоставляет расписание в удобной форме с указанием времени, задействованных водителей и автобусов.

2. Она учитывает ограничения и требования по рабочему времени водителей, избегая простоев или избыточных назначений.

Постановка задачи

Разработать программу для автоматизированного составления оптимального расписания водителей автобусов с учётом следующих условий:

1. **Транспортная нагрузка:**
    - В часы пик (с 7:00 до 9:00 и с 17:00 до 19:00) требуется задействовать 70% автобусов от их общего числа.
    - В остальное время нагрузка составляет 30% от общего числа автобусов.
2. **Типы водителей:**

    1. Водитель типа A:

    - Работает не более 8 часов подряд.
    - После окончания смены требуется замена другим водителем.

    2. Водитель типа B:

    - Может работать всю смену (с 6:00 до 3:00 следующего дня).
    - Требуется минимизировать количество задействованных водителей.
3. **Оптимизация ресурсов:**
    - Цель программы — минимизировать количество водителей, необходимых для обеспечения выполнения всех маршрутов.
    - Расписание должно учитывать непрерывность работы автобусов и равномерное распределение нагрузки между водителями.

4. **Выходные данные:**

   Готовое расписание, содержащее:

   - Время работы каждого автобуса.
   - Номер водителя, назначенного на конкретный маршрут.
   - Общее количество задействованных водителей.

5. **Гибкость:**

   - Программа должна учитывать изменения входных параметров, таких как количество автобусов, тип водителей и продолжительность смены.
   - Возможность выбора между разными типами водителей (A или B) для планирования.

Цель: Создать инструмент, который автоматизирует процесс составления расписания, снижает трудозатраты и повышает эффективность управления автобусным парком.

Используемые подходы

Метод «в лоб»

Преимущества:

1. **Простота реализации:** Логика работы интуитивно понятна и не требует сложных вычислений.
2. **Детерминированность:** Выдаёт стабильный и предсказуемый результат при одинаковых входных данных.
3. **Скорость выполнения:** Для небольших входных данных работает быстро, так как алгоритм просто последовательно распределяет ресурсы.
4. **Прямое управление:** Легко вносить изменения в алгоритм для адаптации под новые требования.

Недостатки:

1. **Не оптимальность:** Не всегда минимизирует количество водителей, так как не учитывает глобальную картину и возможные перекрытия смен.
2. **Ограниченная гибкость:** Подходит только для простых условий. При усложнении задачи (например, учёте дополнительных ограничений) становится менее эффективным.
3. **Ручная доработка:** Часто требует проверки и корректировок для достижения разумного результата.
4. **Отсутствие адаптивности:** Не использует эвристические подходы, из-за чего может давать результат с избыточным использованием ресурсов.

Генетический алгоритм

Преимущества:

1. **Оптимизация:** Позволяет находить близкие к оптимальным решения для сложных задач с множеством ограничений.
2. **Адаптивность:** Легко модифицируется для работы с разными типами задач и параметров, например, добавление перерывов или других правил.
3. **Эффективность для больших данных:** Хорошо справляется с задачами, где метод «в лоб» теряет эффективность.
4. **Гибкость:** Элементы алгоритма (оценочная функция, мутации, скрещивание) можно настраивать для улучшения качества решений.

Недостатки:

1. **Сложность реализации:** Требует большего времени на разработку и отладку по сравнению с методом «в лоб».
2. **Стохастичность:** Результат зависит от случайных факторов, что иногда приводит к необходимости повторного запуска алгоритма.
3. **Время выполнения:** Может быть медленным для малых данных, так как требует нескольких поколений для поиска решения.
4. **Подбор параметров:** Трудоемкость в настройке параметров, таких как размер популяции, число поколений, вероятность мутации и т. д.

Сравнение и выбор подхода:

1. Метод «в лоб» подходит для быстрого решения задач с простыми требованиями и небольшим числом параметров. Его можно использовать, когда важно получить результат быстро и ресурсы не так критичны.

2. Генетический алгоритм предпочтителен для сложных сценариев, где необходимо минимизировать ресурсы и учитывать множество ограничений. Однако его сложность оправдана только при значительных объемах данных или высокой стоимости неоптимальных решений.

Реализация методов:

Метод «В лоб»

Шаги реализации:

1. **Расчёт нагрузки:**

    1. Определить количество автобусов, необходимых в каждый час:

        - В часы пик используется 70% автобусов.
        - Вне часов пик — 30%.

    2. Проход по всем часам рабочего дня.

2. **Распределение водителей:**

    1. Для водителей типа A:

        - Каждый водитель работает максимум 8 часов.
        - Назначить водителей на автобусы в каждом часовом интервале.
        - После окончания смены водителя заменить следующим.

    2. Для водителей типа B:

        - Водитель работает всю смену (с 6:00 до 3:00).
        - Назначить минимальное количество водителей, чтобы покрыть все часы.

3. **Заполнение расписания:**

    - Для каждого часа записать, какие автобусы работают и какие водители их обслуживают.
    - Если не хватает водителей, добавить новых, чтобы маршруты не простаивали.

4. **Вывод результата:**

    - Возвратить расписание в виде таблицы с указанием времени, автобусов и водителей.
    - Подсчитать и вывести общее количество водителей.

Реализация генетического алгоритма (описание):
Шаги реализации:

1. **Создание начальной популяции:**

    - Каждая хромосома — это расписание водителей.
    - Заполнить хромосомы случайным образом, соблюдая ограничения (например, количество автобусов в час).

2. **Оценка (fitness):**

    1. Оценить каждую хромосому:

        - Учитывать количество уникальных водителей.
        - Проверять покрытие всех временных интервалов.

    2. Чем меньше водителей и полнее покрытие — тем лучше.

3. **Селекция:**

    - Отобрать лучших хромосом (с меньшим количеством водителей).
    - Удалить менее подходящие хромосомы.

4. **Скрещивание:**

    - Объединить две хромосомы (родителей), чтобы создать новые (детей).
    - Разделить расписания родителей и объединить их в новые хромосомы.

5. **Мутация:**

    - Случайно изменить части расписания в отдельных хромосомах.
    - Это помогает избежать зацикливания и расширяет поиск.

6. **Проверка завершения:**

    - Повторять шаги селекции, скрещивания и мутации несколько поколений.
    - Завершить, когда либо найдено стабильное оптимальное решение, либо достигнуто максимальное число поколений.

7. **Вывод результата:**

    - Выбрать лучшую хромосому как итоговое расписание.
    - Записать расписание водителей и подсчитать их количество.

Общее сравнение методов

Методы «в лоб» и генетический алгоритм решают одну и ту же задачу — составление расписания водителей, но подходят к этому принципиально разными способами.

Метод «в лоб» прост и интуитивно понятен: он последовательно распределяет водителей на автобусы, соблюдая ограничения, например, рабочие часы или смены. Это делает его подходящим для задач с относительно небольшим количеством параметров. Однако такой подход не всегда обеспечивает минимальное количество задействованных водителей, так как работает жёстко по правилам, не оценивая глобальную оптимальность результата. Это приводит к избыточным ресурсам в ситуациях, когда проблема становится более сложной. Тем не менее, его главным преимуществом остаётся скорость и детерминированность — результат всегда можно предсказать.

Генетический алгоритм, напротив, имитирует естественный процесс эволюции, чтобы находить почти оптимальные решения. Он использует случайность для поиска решений, которые не очевидны для методов прямого расчёта. Это делает его подходящим для сложных задач, где нужно минимизировать ресурсы при большом количестве ограничений и параметров. Однако за эту гибкость приходится платить: генетический алгоритм требует больше времени на настройку и выполнение, а также может выдавать разные результаты при каждом запуске. Ещё одна сложность — необходимость тонкой настройки параметров (размер популяции, вероятность мутаций), от которых зависит качество решения.

В конечном счёте, метод «в лоб» лучше для быстрых решений в стандартных условиях, где важна простота.

Генетический алгоритм выигрывает в ситуациях, требующих оптимизации, особенно если задача усложняется. Выбор метода зависит от требований задачи: для простого расписания можно обойтись первым, а для сложных условий и ограничений стоит использовать второй.

Вывод сравнений:

- **Метод «в лоб»** — простой в реализации и быстрый для стандартных задач, однако не всегда обеспечивает оптимальное использование ресурсов, что может приводить к избыточному числу водителей в расписании.
- **Генетический алгоритм** — более гибкий и мощный инструмент для сложных задач, позволяющий минимизировать количество водителей, но требует больше времени на настройку, выполнение и может выдавать нестабильные результаты при каждом запуске.

Руководство пользователя

1. Запуск программы:

- Устанавливаем Python и необходимую библиотеку (customtkinter)
- Запускаем программу через интерфейс, либо же терминал.

2. Работа с интерфейсом:

- Вводим кол-во маршрутов, которые нам надо выполнить
- Выбираем тип водители (A или B)

3. Результат:

- Программа выведет составленное расписание водителей и их нумерацию.

Все действия работы с программой одинаковы, что для генетического алгоритма, что для метода «В лоб».

Заключение

В ходе работы была разработана программа для автоматизированного составления расписания водителей автобусов, которая учитывает нагрузку на маршруты, типы водителей и временные ограничения. Были реализованы два подхода — метод «в лоб» и генетический алгоритм, каждый из которых имеет свои особенности и применение.

Метод «в лоб» показал себя как простой и быстрый способ для решения задачи при стандартных условиях. Однако он не всегда обеспечивает минимальное количество водителей, так как распределяет их строго по фиксированным правилам. Это делает его менее подходящим для сложных сценариев, где требуется оптимизация.

Генетический алгоритм, напротив, позволил найти более оптимальное решение, минимизировав количество задействованных водителей. Этот подход оказался эффективным для задач с множеством ограничений и переменных, таких как работа водителей типа B, которые обслуживают маршруты на протяжении всей смены. Однако его использование потребовало большего времени на разработку, настройку параметров и выполнение.

Таким образом, выбор подхода зависит от конкретных требований задачи. Для простых и детерминированных условий метод «в лоб» остаётся более подходящим, тогда как для задач, требующих оптимизации ресурсов, рекомендуется использовать генетический алгоритм.

Работа показала важность автоматизации в управлении ресурсами автобусного парка, предложив решения, которые позволяют экономить ресурсы, поддерживать стабильность маршрутов и адаптироваться к изменениям нагрузки.

Приложение:

Метод «В лоб»:

```
import customtkinter as ctk
import math

# Создаем главное окно приложения
app = ctk.CTk()
app.title("Управление расписанием автобусов")
app.geometry("500x700")

# Функция для обработки введенных данных
def submit_data():
    try:
        num_buses = entry_buses.get().strip()
        if not num_buses.isdigit() or int(num_buses) <= 0:
            raise ValueError("Количество автобусов должно быть положительным
числом.")

        num_buses = int(num_buses)
        driver_type = driver_type_var.get()
        schedule, total_drivers = calculate_schedule(num_buses, driver_type)

        # Вывод расписания в интерфейсе
        output_textbox.delete(1.0, "end")
        output_textbox.insert("end", "Составленное расписание:\n")
        for time, buses in schedule.items():
            output_textbox.insert("end", f"{time}:\n")
            output_textbox.insert("end", "\n".join(buses) + "\n\n")

        output_textbox.insert("end", f"\nОбщее количество водителей:
{total_drivers}\n")
    except ValueError as e:
        output_textbox.delete(1.0, "end")
        output_textbox.insert("end", f"Ошибка: {e}")

# Функция для расчета расписания и минимального числа водителей
def calculate_schedule(num_buses, driver_type):
    peak_hours = [(7, 9), (17, 19)]  # Часы пик
    route_time = 70  # Длина маршрута в минутах

    peak_load = 0.7  # 70% нагрузки в часы пик
    off_peak_load = 0.3  # 30% нагрузки вне часов пик

    work_hours = list(range(6, 24)) + list(range(0, 3))  # Время работы автобусов
с 6:00 до 3:00

    schedule = {}
```

```
    # Логика для водителей типа B
    if driver_type == "B":
        total_drivers = math.ceil(num_buses * peak_load)
        drivers = [f"Водитель {i + 1}" for i in range(total_drivers)]

        for hour in work_hours:
            buses = []

            if any(start <= hour < end for start, end in peak_hours):
                buses_needed = math.ceil(num_buses * peak_load)
            else:
                buses_needed = math.ceil(num_buses * off_peak_load)

            for bus_id in range(1, buses_needed + 1):
                driver = drivers[(bus_id - 1) % total_drivers]
                buses.append(f"Автобус {bus_id} ({driver})")

            schedule[f"{hour:02d}:00"] = buses

        return schedule, total_drivers

    # Для водителей типа A логика остается прежней
    max_work_hours = 8
    driver_pool = []
    driver_id = 1

    for start_hour in range(6, 24, max_work_hours):
        shift_start = start_hour
        shift_end = (start_hour + max_work_hours) % 24
        driver_pool.append({
            "id": driver_id,
            "start": shift_start,
            "end": shift_end
        })
        driver_id += 1

    if 3 > 0:
        driver_pool.append({
            "id": driver_id,
            "start": 0,
            "end": 3
        })

    total_drivers = len(driver_pool)

    for hour in work_hours:
        buses = []
        if any(start <= hour < end for start, end in peak_hours):
            buses_needed = math.ceil(num_buses * peak_load)
        else:
```

```
            buses_needed = math.ceil(num_buses * off_peak_load)

        active_drivers = [
            driver for driver in driver_pool if driver["start"] <= hour <
driver["end"]
        ]

        while len(active_drivers) < buses_needed:
            new_driver_id = len(driver_pool) + 1
            new_start = hour
            new_end = (hour + max_work_hours) % 24
            driver_pool.append({
                "id": new_driver_id,
                "start": new_start,
                "end": new_end
            })
            active_drivers.append(driver_pool[-1])
            total_drivers += 1

        for bus_id in range(1, buses_needed + 1):
            driver = active_drivers[(bus_id - 1) % len(active_drivers)]
            buses.append(f"Автобус {bus_id} (Водитель {driver['id']})")

        schedule[f"{hour:02d}:00"] = buses

    return schedule, total_drivers

# Переменные для хранения данных
entry_buses = ctk.CTkEntry(app, placeholder_text="Введите количество автобусов")
entry_buses.pack(pady=10)

driver_type_var = ctk.StringVar(value="A")

driver_type_label = ctk.CTkLabel(app, text="Выберите тип водителя:")
driver_type_label.pack(pady=5)

driver_type_a = ctk.CTkRadioButton(app, text="Тип A", variable=driver_type_var,
value="A")
driver_type_a.pack(pady=5)

driver_type_b = ctk.CTkRadioButton(app, text="Тип B", variable=driver_type_var,
value="B")
driver_type_b.pack(pady=5)

# Поле для отображения результата
output_textbox = ctk.CTkTextbox(app, height=400)
output_textbox.pack(pady=10)

# Кнопка для подтверждения данных
submit_button = ctk.CTkButton(app, text="Подтвердить", command=submit_data)
submit_button.pack(pady=20)
```

```
# Запуск приложения
app.mainloop()
```

Генетический алгоритм:

```
import customtkinter as ctk
import math
import random

# Создаем главное окно приложения
app = ctk.CTk()
app.title("Управление расписанием автобусов")
app.geometry("500x700")

# Функция для обработки введенных данных
def submit_data():
    try:
        num_buses = entry_buses.get().strip()
        if not num_buses.isdigit() or int(num_buses) <= 0:
            raise ValueError("Количество автобусов должно быть положительным
числом.")

        num_buses = int(num_buses)
        driver_type = driver_type_var.get()

        schedule, total_drivers = genetic_algorithm_schedule(num_buses,
driver_type)

        # Вывод расписания в интерфейсе
        output_textbox.delete(1.0, "end")
        output_textbox.insert("end", "Составленное расписание:\n")
        for time, buses in schedule.items():
            output_textbox.insert("end", f"{time}:\n")
            output_textbox.insert("end", "\n".join(buses) + "\n\n")

        output_textbox.insert("end", f"\nОбщее количество водителей:
{total_drivers}\n")
    except ValueError as e:
        output_textbox.delete(1.0, "end")
        output_textbox.insert("end", f"Ошибка: {e}")

# Реализация генетического алгоритма
def genetic_algorithm_schedule(num_buses, driver_type):
    peak_hours = [(7, 9), (17, 19)]  # Часы пик
    peak_load = 0.7
```

```
    off_peak_load = 0.3
    # Рабочий день: с 6 до 24 часов, плюс 0-2 часа следующего дня, всего 20 часов
(пример)
    work_hours = list(range(6, 24)) + list(range(0, 3))

    # Параметры генетического алгоритма
    population_size = 50
    generations = 100
    mutation_rate = 0.1

    max_work_hours = 8 if driver_type == "A" else 24

    def create_initial_population():
        population = []
        for _ in range(population_size):
            individual = []
            for hour in work_hours:
                # Определяем, час пик или нет
                if any(start <= hour < end for start, end in peak_hours):
                    buses_needed = math.ceil(num_buses * peak_load)
                else:
                    buses_needed = math.ceil(num_buses * off_peak_load)

                hour_assigned_drivers = set()
                drivers = []
                for _ in range(buses_needed):
                    while True:
                        driver = random.randint(1, num_buses * 2)
                        # Проверяем, что один водитель не ведет два автобуса в
один час
                        if driver not in hour_assigned_drivers:
                            hour_assigned_drivers.add(driver)
                            drivers.append(driver)
                            break
                individual.append(drivers)
            population.append(individual)
        return population

    def fitness(schedule):
        # Фитнес: минимизируем количество уникальных водителей и штрафы за
переработку
        unique_drivers = set()
        driver_hours_count = {}

        for hour_drivers in schedule:
            for d in hour_drivers:
                unique_drivers.add(d)
                driver_hours_count[d] = driver_hours_count.get(d, 0) + 1

        # Штраф за переработку
        penalty = 0
```

```
        for d, count in driver_hours_count.items():
            if count > max_work_hours:
                penalty += (count - max_work_hours)

        # Цель - минимизировать число уникальных водителей + штраф
        return len(unique_drivers) + penalty

    def select(population):
        scored_population = [(individual, fitness(individual)) for individual in
population]
        scored_population.sort(key=lambda x: x[1])
        return [individual for individual, _ in
scored_population[:population_size // 2]]

    def crossover(parent1, parent2):
        point = random.randint(0, len(parent1) - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2

    def mutate(individual):
        # Мутация: случайная замена одного из водителей в случайный час
        for hour_drivers in individual:
            if random.random() < mutation_rate and len(hour_drivers) > 0:
                index = random.randint(0, len(hour_drivers) - 1)
                old_driver = hour_drivers[index]
                # Убираем старого водителя из этого часа
                # Находим нового водителя, не совпадающего с другими в том же
часу
                current_hour_set = set(hour_drivers)
                current_hour_set.remove(old_driver)
                while True:
                    new_driver = random.randint(1, num_buses * 2)
                    if new_driver not in current_hour_set:
                        hour_drivers[index] = new_driver
                        break

    # Создаем начальную популяцию
    population = create_initial_population()

    # Запускаем эволюцию
    for _ in range(generations):
        selected = select(population)
        next_generation = []

        while len(next_generation) < population_size:
            parents = random.sample(selected, 2)
            child1, child2 = crossover(parents[0], parents[1])
            mutate(child1)
            mutate(child2)
            next_generation.extend([child1, child2])
```

```
        population = next_generation

    # Находим лучшее решение
    best_schedule = min(population, key=fitness)
    schedule_dict = {}

    for i, hour_drivers in enumerate(best_schedule):
        hour = work_hours[i]
        buses = [f"Автобус {j + 1} (Водитель {driver})" for j, driver in
enumerate(hour_drivers)]
        schedule_dict[f"{hour:02d}:00"] = buses

    return schedule_dict, fitness(best_schedule)

# Переменные для хранения данных
entry_buses = ctk.CTkEntry(app, placeholder_text="Введите количество автобусов")
entry_buses.pack(pady=10)

driver_type_var = ctk.StringVar(value="A")

driver_type_label = ctk.CTkLabel(app, text="Выберите тип водителя:")
driver_type_label.pack(pady=5)

driver_type_a = ctk.CTkRadioButton(app, text="Тип A", variable=driver_type_var,
value="A")
driver_type_a.pack(pady=5)

driver_type_b = ctk.CTkRadioButton(app, text="Тип B", variable=driver_type_var,
value="B")
driver_type_b.pack(pady=5)

# Поле для отображения результата
output_textbox = ctk.CTkTextbox(app, height=400)
output_textbox.pack(pady=10)

# Кнопка для подтверждения данных
submit_button = ctk.CTkButton(app, text="Подтвердить", command=submit_data)
submit_button.pack(pady=20)

# Запуск приложения
app.mainloop()
```

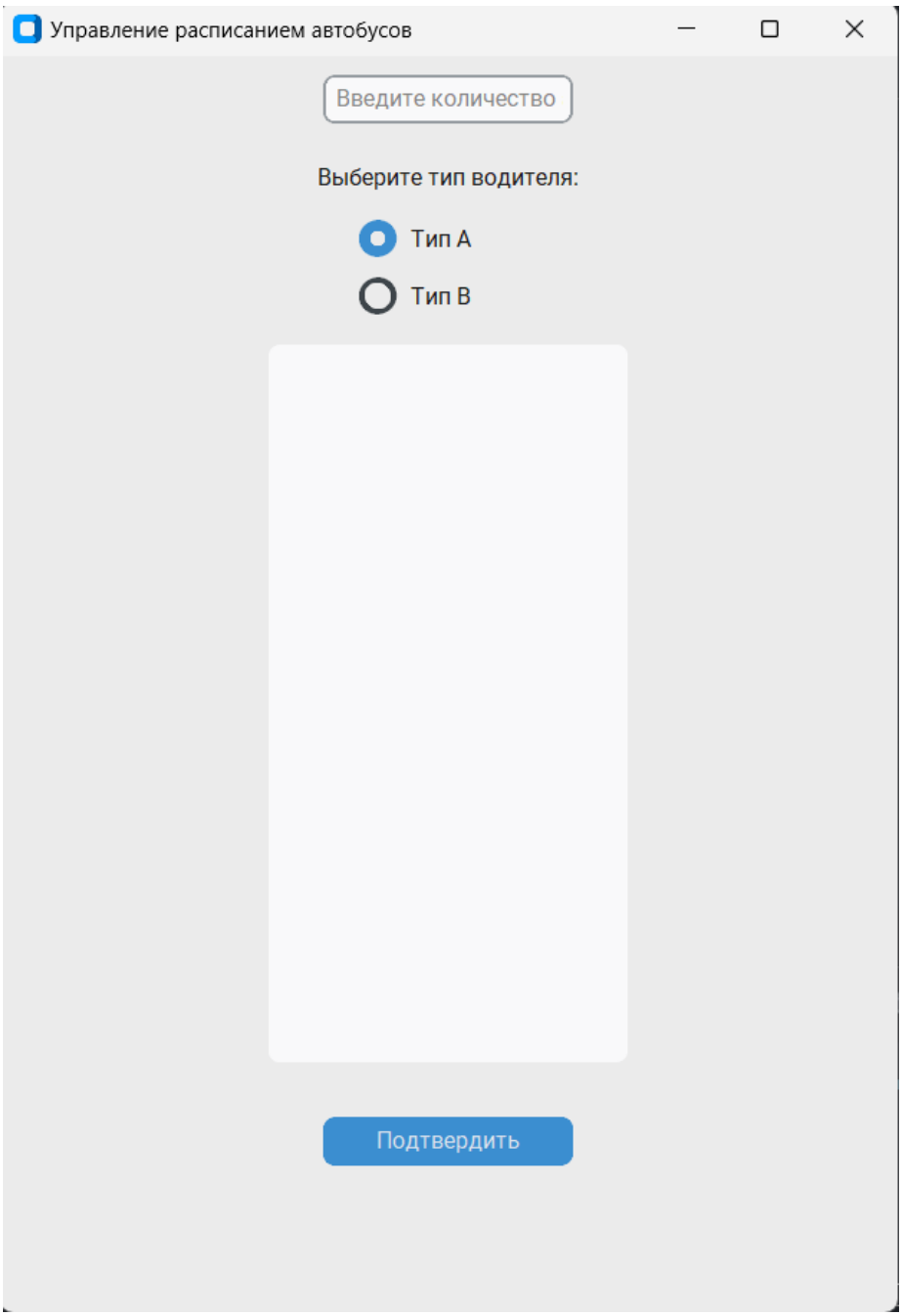

Рисунок 1 – Общий интерфейс двух программ.

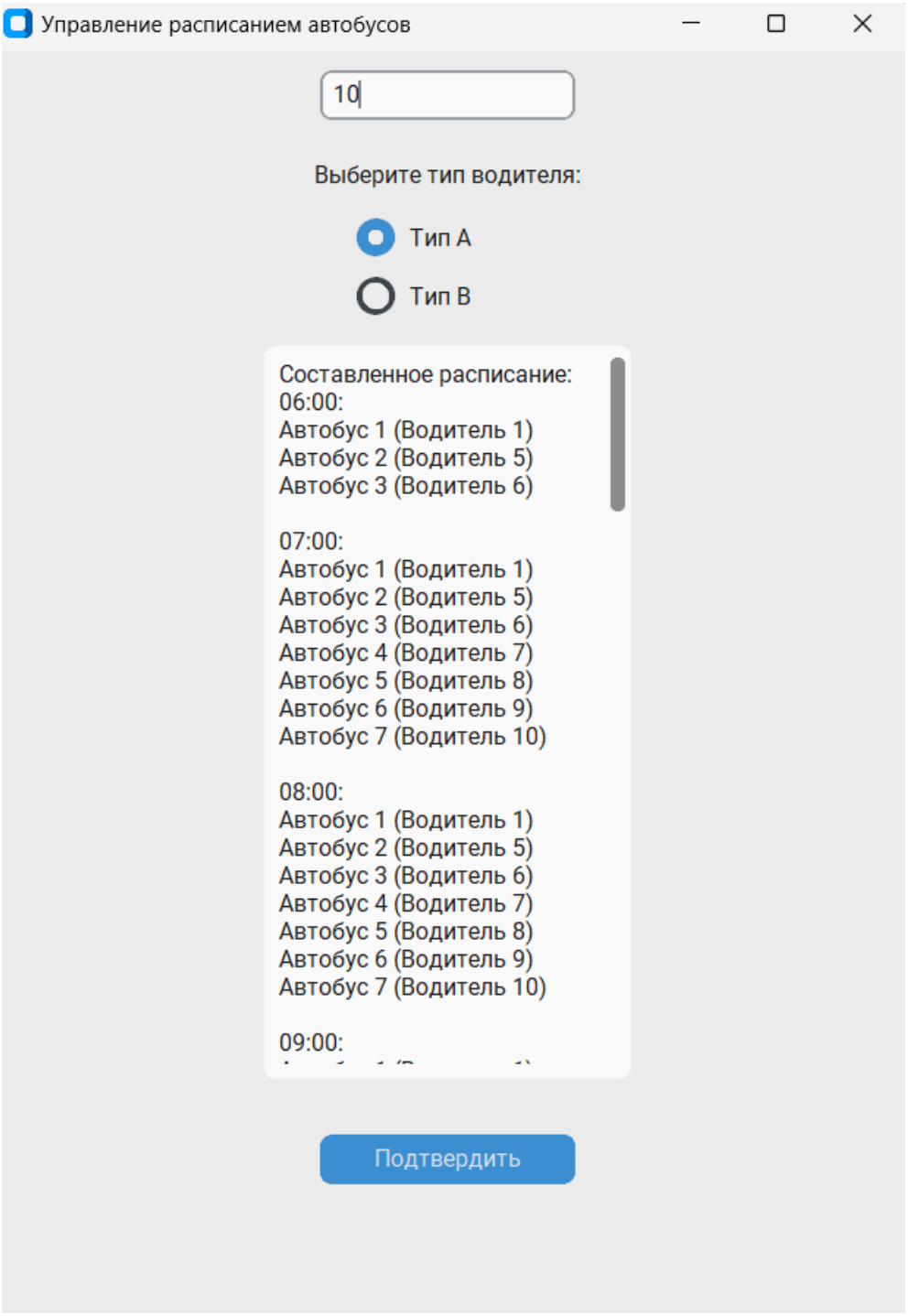

Рисунок 2 – Вывод метода «В лоб»

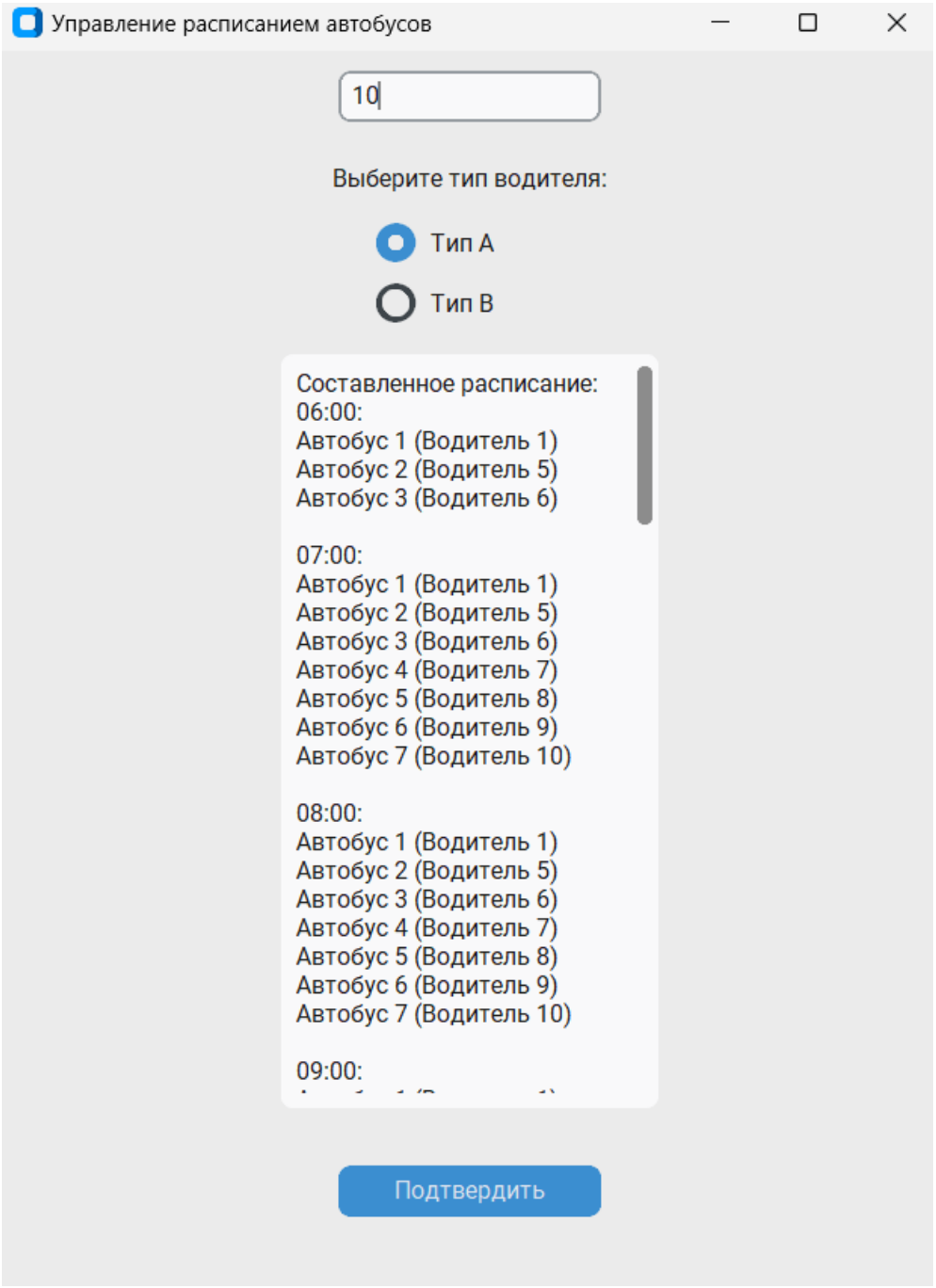

Рисунок 2 – Вывод генетического алгоритма