МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по учебной практике

Тема: Генетические алгоритмы в задаче коммивояжёра с приоритетами

Студент гр. 3388 Снигирёв А.А.

Студентка гр. 3388 Титкова С.Д.

Преподаватель Жангиров Т.Р.

Санкт-Петербург
2025

ЗАДАНИЕ
НА УЧЕБНУЮ ПРАКТИКУ

Студент Снигирёв А.А. группы 3388

Студентка Титкова С.Д. группы 3388

Тема практики: разработка генетического алгоритма для задачи коммивояжёра с приоритетами

Задание на практику:

Командная итеративная разработка генетического алгоритма на C++ с графическим интерфейсом.

Алгоритм: Генетический алгоритм для задачи коммивояжёра с приоритетами.

Сроки прохождения практики: 25.06.2024 – 06.07.2024

Студент Снигирёв А.А.

Студентка Титкова С.Д.

Руководитель Жангиров Т.Р.

АННОТАЦИЯ

Цель практики — освоение генетических алгоритмов на примере решения задачи коммивояжёра с приоритетами. В ходе практики изучены ключевые понятия: генотип, популяция, функция приспособленности, отбор, скрещивание и мутация. Реализован генетический алгоритм для поиска оптимального маршрута с учётом приоритетов городов. Дополнительно разработан графический интерфейс для визуализации работы алгоритма.

ВВЕДЕНИЕ

Целью учебной практики является изучение генетических алгоритмов как метода эволюционной оптимизации и их применение для решения задачи коммивояжёра с приоритетами.

В ходе практики были поставлены следующие задачи: ознакомление с базовыми понятиями генетических алгоритмов (генотип, популяция, функция приспособленности, отбор, скрещивание, мутация); разработка и реализация генетического алгоритма для задачи коммивояжёра с учётом приоритетов посещаемых городов; создание графического интерфейса для наглядной демонстрации работы алгоритма.

Реализуемый алгоритм основан на принципах естественного отбора: популяция потенциальных решений эволюционирует через последовательность поколений, улучшая свои характеристики. В ходе работы алгоритм оптимизирует маршрут коммивояжёра, минимизируя общую длину пути с учётом заданных приоритетов.

Генетические алгоритмы широко используются для решения NP-трудных задач, включая маршрутизацию, планирование и другие оптимизационные задачи. Реализованный подход может быть адаптирован для более сложных условий, таких как динамически изменяющиеся параметры городов или многокритериальная оптимизация.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

- Программа должна иметь GUI
- Должна быть возможность задать данные из разных источников по выбору пользователя: из файла, ввод через GUI, случайная генерация
- При реализации алгоритмов нельзя использовать библиотеки, решающие задачу напрямую. Генетический алгоритм должен быть реализован вручную. Использовать библиотеки для графического интерфейса/загрузки данных – можно.
- Пользователь должен иметь возможность задавать параметры генетического алгоритма через GUI.
- В приложении должна быть реализована пошаговая демонстрация работы генетического алгоритма. Т.е. должна быть кнопка «следующий шаг» и кнопка «выполнить до конца» (чтобы перейти к конечным результатам). На каждом шаге алгоритма должна быть показано графическое представление наилучшего решения, стоимость этого решения, средняя стоимость поколения.
- В GUI должно быть поле с графиком изменения приспособленности лучший и средней в зависимости от поколения. График должен строиться по ходу выполнения генетического алгоритма.
- После завершения выполнения генетического алгоритма, программа не должна закрывать, а должна быть возможность выполнить его на этих же данных, но с другими параметрами алгоритма, либо загрузить новые данные.
- Плюсом будет реализация следующих функций:
    - На каждом шаге можно выбрать и просмотреть любое решение
    - Возможность просмотра сразу нескольких решений, чтобы их можно было сравнить
    - Возможность вернуться на несколько шагов назад

- Выбор различных модификаций генетических алгоритмов

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

Дата	Этап	Выполнено
27.06.25	Формирование бригады, выбор ЯП, распределений ролей в бригаде, выбор задачи.	да
30.06.25	Демонстрация прототипа GUI и плана решения задачи (описание формата данных, используемых функций качества, и т.д.)	
02.07.25	Промежуточная сдача программы. На данном этапе может быть не быть реализован весь функционал, GUI еще не связан с алгоритмом.	
04.07.25	Промежуточная сдача программы. На данном этапе может быть не быть реализован весь функционал, GUI еще не связан с алгоритмом.	
06.07.25	Демонстрация финальной версии программы. Беседа по написанному коду и решаемой задаче.	

2.2. Распределение ролей в бригаде

Снигирёв А.А.: реализация генетического алгоритма

Титкова С.Д.: реализация графического интерфейса, изучение теории и тестирование кода

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Графический интерфейс

Для реализации графического интерфейса используются следующее:

- *Qt Framework*: используется для создания кроссплатформенного GUI.
- *QMainWindow*: основной класс окна приложения.
- *QGraphicsView/QGraphicsScene*: для отображения городов и путей (графическое представление решения).
- *QtCharts (QChart, QChartView, QLineSeries)*: для построения графика приспособленности.
- *QFileDialog, QTextEdit, QSpinBox, QDoubleSpinBox, QPushButton, QLabel, QComboBox*: для ввода данных, управления и отображения информации.
- *QVBoxLayout, QHBoxLayout*: для организации компоновки элементов.

Общая структура: окно приложения (размер 1400x800 пикселей) разделено на две основные части с помощью горизонтального компоновщика (horizontalLayout):

- *Левая часть* (1/3 ширины): элементы управления, ввод данных и настройки.
- *Правая часть* (2/3 ширины): визуализация решений и график

Элементы интерфейса и их расположение

1. Левая часть: Элементы управления

Расположена слева в вертикальном компоновщике (verticalLayout). Содержит элементы для ввода данных, настройки и управления алгоритмом.

- *Кнопки ввода данных* (на рисунке 1 обозначено цифрой 1) (в горизонтальном компоновщике buttonLayout):

- *"Загрузить из файла"* (loadFileButton): открывает диалог для выбора текстового файла с данными городов (формат: x,y,приоритет).
- *"Сгенерировать случайно"* (generateRandomButton): запрашивает количество городов и генерирует их случайным образом.
- *"Ввести города"* (inputCitiesButton): считывает данные городов из текстового поля.
- *Расположение*: верхняя часть левого столбца, три кнопки в ряд.

- *Текстовое поле* (на рисунке 1 обозначено цифрой 2) (textEdit):
  - Поле для ручного ввода координат и приоритетов городов (формат: x,y,приоритет, по одной строке на город).
  - Имеет подсказку: "Введите города (x,y,приоритет) по одному на строку".
  - Минимальная высота: 100 пикселей.
  - *Расположение:* под кнопками ввода данных.
- *Группа параметров генетического алгоритма* (на рисунке 1 обозначено цифрой 3) (paramsGroup, в компоновщике verticalLayout_2):
  - *"Размер популяции"* (popSizeSpinBox): числовое поле (10–1000, по умолчанию 100).
  - *"Вероятность мутации"* (mutationRateSpinBox): поле для дробных чисел (0–1, шаг 0.01, по умолчанию 0.01).
  - *"Максимум поколений"* (maxGenSpinBox): числовое поле (100–10000, по умолчанию 1000).
  - *"Применить параметры"* (applyParamsButton): кнопка для применения параметров (пока без функционала).
  - *Расположение:* под текстовым полем, элементы сгруппированы в рамке с заголовком "Параметры генетического алгоритма".
- *Возврат назад* (на рисунке 1 обозначено цифрой 4) (в горизонтальном компоновщике backLayout):

    - *"Вернуться назад"* (backButton): кнопка для возврата на предыдущие шаги (не реализована).
    - *Поле шагов* (backStepsSpinBox): указывает количество шагов назад (1–100, по умолчанию 1).
    - *Расположение:* под группой параметров, два элемента в ряд.
- *Кнопки управления алгоритмом* (на рисунке 1 обозначено цифрой 4):
    - *"Следующий шаг"* (runStepButton): выполняет один шаг алгоритма, обновляя решение, график и метки.
    - *"Запустить до конца"* (runToEndButton): выполняет алгоритм до максимального числа поколений.
    - *Расположение*: под блоком возврата, каждая кнопка на отдельной строке.
- *Метки информации* (на рисунке 1 обозначено цифрой 5):
    - *Поколение* (generationLabel): показывает номер текущего поколения (например, "Поколение: 0").
    - *Лучшая приспособленность* (bestFitnessLabel): показывает стоимость лучшего решения.
    - *Средняя приспособленность* (avgFitnessLabel): показывает среднюю приспособленность популяции.
    - *Расположение*: под кнопками управления, каждая метка на отдельной строке.
- *Выбор решения* (на рисунке 1 обозначено цифрой 5) (в горизонтальном компоновщике solutionSelectionLayout):
    - *"Выбрать решение"* (solutionComboBox): выпадающий список для выбора отображаемого решения (пока без функционала).
    - *Расположение*: внизу левого столбца.

2. Правая часть: Визуализация

Расположена справа в вертикальном компоновщике (verticalLayout_3) с соотношением высот 2:3 (верхняя часть к нижней).

- *Основное окно визуализации* (на рисунке 1 обозначено цифрой 3) (graphicsView):
    - Показывает города как красные круги (диаметр 10 пикселей) с метками вида Город N (П: M) (N — номер, M — приоритет).
    - Если есть лучшее решение, рисует путь синими линиями, соединяющими города в порядке посещения, с возвратом в начальный город.
    - Минимальный размер: 400x200 пикселей.
    - *Расположение:* верхняя часть правого столбца, занимает около 40% высоты.
- *Окно сравнения* (на рисунке 1 обозначено цифрой 7) (в горизонтальном компоновщике compareLayout):
    - *Окно визуализации* (compareGraphicsView): показывает только города (без пути). Предназначено для отображения альтернативного решения. Минимальный размер: 400x200 пикселей.
    - *"Сравнить с"* (compareComboBox): выпадающий список для выбора решения для сравнения (пока без функционала). Максимальная ширина: 150 пикселей.
    - *Расположение:* под основным окном, compareGraphicsView занимает большую часть ширины, а compareComboBox с меткой — узкую правую часть.
- *График приспособленности* (на рисунке 1 обозначено цифрой 8) (QChartView):
    - Показывает две линии:

- Красная: лучшая приспособленность (стоимость лучшего решения).
- Синяя: средняя приспособленность популяции.

- Оси: горизонтальная — поколения, вертикальная — приспособленность.
- Заголовок: "Динамика приспособленности по поколениям". Легенда включена.
- Минимальный размер: 600x400 пикселей.
- *Расположение:* внизу правого столбца, занимает около 60% высоты.

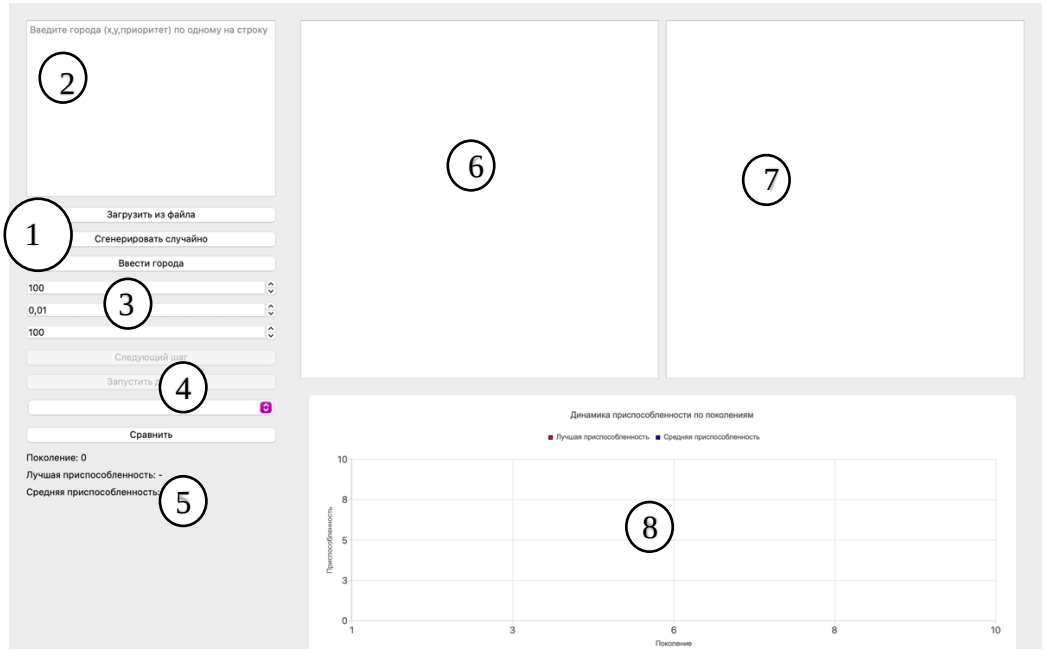

Рисунок 1. Прототип GUI.

3.2. Структуры данных

1. *Структура Town*:
   - Поля:
     - *int priority*: приоритет города.
     - *string name*: название города.
     - *double x, y*: координаты города (x, y).
2. *Вектор std::vector<Town>:*
   - Контейнер для хранения списка объектов типа Town.

- Содержит все города, введённые пользователем, и передаётся в функции для расчёта расстояний и эволюционного алгоритма.

3. *Матрица расстояний std::vector<std::vector<double>>:*
   - Двумерный вектор, где элемент [i][j] представляет евклидово расстояние между городами i и j.
4. *Словарь приоритетов std::map<int, std::vector<int>>:*
   - Ассоциативный контейнер, где ключ — приоритет, а значение — вектор индексов городов, имеющих этот приоритет.
   - Необходим для группировки городов по приоритетам.
5. *Популяция std::vector<std::vector<int>>:*
   - Двумерный вектор, где каждый элемент представляет одну особь (хромосому) — последовательность индексов городов, образующих маршрут.
   - Создаётся, как начальная популяция для эволюционного алгоритма. Обновляется в процессе эволюции.
6. *Вектор приспособленностей std::vector<double>:*
   - Вектор, содержащий значения приспособленности для каждой особи в популяции.
7. *Вектор лучших особей std::vector<std::vector<int>>:*
   - Двумерный вектор, хранящий лучшие особи для каждой итерации эволюционного алгоритма.
8. *Вектор лучших приспособленностей std::vector<double>:*
   - Вектор, хранящий значения приспособленности лучших особей для каждой итерации.
   - Возвращается как результат работы алгоритма.

3.3. Наработки методов ga

В ходе практики необходимо реализовать 3 эволюционных механизма:

- *Отбор*
    - Из всех видов отбор, мы решили остановиться на турнирном, так как он относительно прост в реализации: случайным образом выбирается небольшая группа особей, и из них выбирается лучшая по значению приспособленности. Это требует минимальных вычислений и легко масштабируется.
- *Скрещивание*
    - В данном механизме мы остановились на упорядоченном скрещивании, так как в задаче коммивояжёра важен порядок посещённых городов.
- *Мутации*
    - В реализации алгоритма используется мутация обменом.

3.4. Методы

- *Town::Town(double x, double y, int priority, std::string name)* - создаёт объект города с координатами (x, y), приоритетом и именем;
- *print_lst(std::vector towns)* - выводит список городов с их координатами и приоритетами;
- *print_matrix(std::vector<std::vector> matrix)* - выводит матрицу расстояний между городами;
- *print_dvector(std::vector& vec)* - выводит вектор фитнес-значений (длин путей);
- *print_vector(std::vector& vec*) - Выводит хромосому (последовательность индексов городов);
- *print_priority_groups(const std::map<int, std::vector>& priority_groups)* - выводит группы городов по приоритетам с их индексами и размерами;
- *print_priority_groups_with_ranges(const std::map<int, std::vector>& priority_groups)* - выводит группы приоритетов с диапазонами позиций в хромосоме;

- *console_input()* - запрашивает количество городов и их координаты с приоритетами, возвращает вектор объектов Town;
- *calculate_distances(std::vector towns)* - вычисляет матрицу расстояний между городами (евклидово расстояние);
- *make_priority_groups(std::vector& towns)* - группирует города по приоритетам, возвращает словарь с индексами городов;
- *make_start_population(std::vector& towns, std::vector<std::vector>& distances, std::map<int, std::vector> priority_groups, int p_size)* - создаёт начальную популяцию из p_size хромосом, упорядоченных по приоритетам.
- *fitness_f(std::vector& individ, std::vector<std::vector>& matrix)* - вычисляет фитнес хромосомы;
- *tournament_selection(const std::vector<std::vector>& population, const std::vector& fitnesses, int k)* - выбирает хромосому с помощью турнирного отбора из k случайных хромосом;
- *calculate_fitnesses(std::vector<std::vector>& population, std::vector<std::vector>& matrix, int p_size)* - вычисляет фитнес для каждой хромосомы в популяции;
- *is_valid_chromosome(const std::vector& individ, const std::map<int, std::vector>& priority_groups)* - проверяет, является ли хромосома допустимой (без повторов и с соблюдением приоритетов);
- *group_crossover(const std::vector& group1, const std::vector& group2, std::vector& child1, std::vector& child2, std::mt19937& gen)* - выполняет кроссовер для одной группы приоритетов между родителями;
- *merge_child_groups(const std::vector<std::vector>& child_fir_groups, const std::vector<std::vector>& child_sec_groups, std::vector& child_fir, std::vector& child_sec)* - объединяет группы потомков в полные хромосомы;
- *ox1_crossover(std::vector& parent_fir, std::vector& parent_sec, std::vector& child_fir, std::vector& child_sec, const std::map<int, std::vector>& priority_groups, double cross_prob)* - выполняет кроссовер (OX1) между родителями с учётом приоритетов и вероятности;
- *mutate(std::vector& individ, const std::map<int, std::vector>& priority_groups, double mutation_prob)* - выполняет мутацию хромосомы (обмен двух элементов в группе приоритетов) с заданной вероятностью;

- *find_best_individ(std::vector& fitnesses)* - находит индекс хромосомы с лучшим фитнес-значением;
- *Evolution(std::vector& towns, int population_size, int generations_number, double mut_prob, double cross_prob)* - реализует генетический алгоритм, создавая популяцию и выполняя эволюцию через поколения;
- *main(int argc, char argv**)* - точка входа: проверяет аргументы, вызывает ввод городов, запускает эволюцию и выводит результаты;