**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики-процессов управления**

**Программа бакалавриата**

**“Большие данные и распределенная цифровая платформа”**

**ОТЧЕТ**

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

**на тему «Решение задачи о коммивояжере с помощью метода ближайшего соседа»**

**Студент гр. 23Б15-пу**

**Абрахин Е.Д.**

**Преподаватель**

**Дик А.Г.**

**Санкт-Петербург**

**2025 г.**

Оглавление

1. [Цель работы 3](#_Toc192027290)
2. [Описание задачи (формализация задачи) 3](#_Toc192027291)
3. [Теоретическая часть 4](#_Toc192027292)
4. [Основные шаги программы 6](#_Toc192027293)
5. [Блок схема программы 8](#_Toc192027294)
6. [Описание программы 9](#_Toc192027295)
7. [Рекомендации пользователя 11](#_Toc192027296)
8. [Рекомендации программиста 11](#_Toc192027297)
9. [Исходный код программы 12](#_Toc192027298)
10. [Контрольный пример 13](#_Toc192027299)
11. [Анализ результатов работы алгоритма 16](#_Toc192027300)
12. [Вывод 18](#_Toc192027301)
13. [Источники 19](#_Toc192027302)

# Цель работы

Целью данной лабораторной работы является исследование метода ближайшего соседа для решения задачи коммивояжера, а также анализ его эффективности в поиске кратчайшего гамильтонова цикла. В ходе работы будет разработана программа, реализующая алгоритм ближайшего соседа, с возможностью визуализации построенного маршрута. Программа будет протестирована на заданном взвешенном орграфе для анализа качества полученного решения и сравнения его с оптимальным маршрутом.

# Описание задачи (формализация задачи)

1. Изучение задачи коммивояжера и метода ближайшего соседа:

Ознакомиться с теоретическими основами задачи коммивояжера, принципами работы алгоритма ближайшего соседа и его применением для поиска кратчайшего маршрута. Исследовать его преимущества и недостатки по сравнению с точными методами решения.

1. Разработка программы для нахождения кратчайшего гамильтонова цикла:

Написать программу, реализующую алгоритм ближайшего соседа для поиска приближенного решения задачи коммивояжера. Реализовать визуализацию построенного маршрута на графе.

1. Тестирование программы на взвешенном орграфе:

Провести тестирование программы на контрольном примере, представленном во взвешенном орграфе (рис. 1). Сравнить полученный маршрут с оптимальным решением, оценив точность и эффективность метода ближайшего соседа.

1. Анализ результатов:

Оценить качество полученного решения, проанализировать влияние структуры графа на эффективность метода ближайшего соседа. Определить, в каких случаях метод показывает наилучшие результаты и какие его ограничения могут повлиять на точность решения.

# Теоретическая часть

**Алгоритм ближайшего соседа**

Алгоритм ближайшего соседа (Nearest Neighbor Algorithm) [[1]](#_Источники_1) с модификацией выбора стартовой вершины — это жадный метод решения задачи коммивояжера, который используется для построения приближенного решения. Алгоритм основан на поочередном выборе ближайшей не посещенной вершины, начиная с произвольной стартовой точки. Этот метод прост в реализации и эффективен по времени выполнения, однако не всегда находит оптимальный маршрут. Модификация заключается в переборе всех стартовых вершин вместо одной, для возможного увеличения качества нахождения пути.

**Параметры алгоритма и их роль**

1. Стартовая вершина:

Выбирается произвольно или на основе определенного критерия. От выбора стартовой вершины может зависеть итоговый маршрут.

1. Метрика расстояний:

Используется для определения ближайшей вершины. Чаще всего применяется евклидово расстояние или заранее заданная матрица смежности.

1. Правило выбора следующей вершины:

Алгоритм всегда выбирает ближайшую непосещенную вершину, минимизируя локальный путь на каждом шаге.

**Принцип работы**

1. Выбирается стартовая вершина.
2. Из текущей вершины выбирается ближайшая непосещенная вершина.
3. Вершина помечается как посещенная, и маршрут продолжается.
4. Когда все вершины посещены, путь замыкается, возвращаясь в стартовую вершину.

**Оценка эффективности**

Алгоритм ближайшего соседа имеет сложность O(n\*log(n)). Основные критерии эффективности:

* Качество решения:

Насколько близок найденный путь к оптимальному гамильтонову циклу.

* Скорость выполнения:

Количество итераций, необходимых для построения маршрута.

* Зависимость от стартовой вершины:

Итоговый маршрут может отличаться в зависимости от выбора начальной точки.

Хотя метод ближайшего соседа работает быстро, он не гарантирует нахождение глобального оптимума и может приводить к субоптимальным маршрутам.

# Основные шаги программы

PSO- Основные шаги программы The\_nearest\_neighbor\_method.pyw

1. **Запуск программы:**
   * Инициализация графического интерфейса с настройками и элементами управления:
     + Область для построения графа (визуальное отображение вершин и ребер).
     + Таблица для ввода и редактирования весов ребер.
     + Текстовое поле для вывода результатов.
     + Галочка для использования модификации.
     + Кнопки управления: «Рассчитать», «Отмена», «Очистить».
2. **Настройка графа:**
   * Пользователь добавляет вершины графа, кликая по области визуализации.
   * Добавление ребер происходит путём последовательного выбора вершин:
     + При соединении автоматически вычисляется расстояние между выбранными вершинами.
     + В таблице отображаются данные о соединённых вершинах и весах ребер.
   * Возможность ручного редактирования весов через таблицу.
3. **Запуск алгоритма:**
   * При нажатии кнопки «Рассчитать» запускается алгоритм поиска кратчайшего гамильтонова цикла методом ближайшего соседа.
   * Использование модификации зависит от того, стоит ли галочка «Использовать модификацию»
   * Алгоритм может перебрать все возможные стартовые вершины для выбора оптимального решения.
4. **Поиск решения:**
   * Алгоритм начинает с выбранной стартовой вершины и на каждом шаге:
     + Определяет ближайшую не посещённую вершину (с учётом заданных весов).
     + Последовательно формирует путь, пока все вершины не будут посещены.
   * По завершении маршрут замыкается, возвращаясь к стартовой вершине.
   * Вычисляется общая длина полученного пути и время выполнения алгоритма.
   * При отсутствии замыкания выводится сообщение о неудаче нахождения пути.
5. **Обновление интерфейса:**
   * Графическая область обновляется: рисуются вершины и ребра (со стрелками, указывающими направление).
   * Таблица ребер заполняется актуальными данными.
   * В текстовом поле выводятся результаты: найденный путь, его длина и время выполнения алгоритма.
6. **Дополнительные возможности:**
   * Возможность отмены последнего действия (удаление вершины или ребра) с помощью кнопки «Отмена».
   * Очистка текущего графа и всех результатов для проведения нового расчёта с помощью кнопки «Очистить».

# Блок схема программы

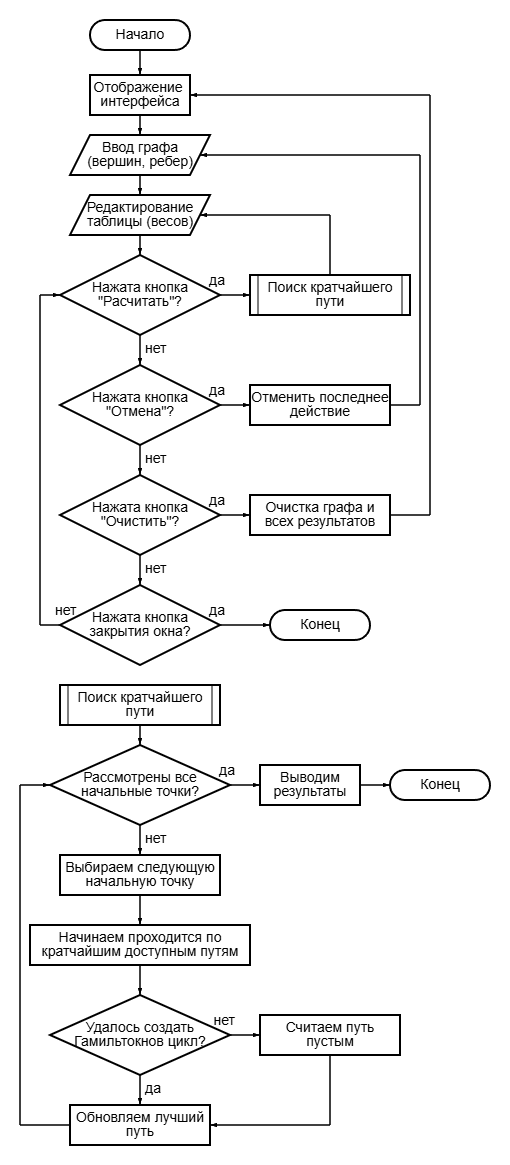


Рис 1. Блок-схема The\_nearest\_neighbor\_method.py

# Описание программы

Программа написана на языке Python 3.12.6 и использует библиотеки numpy [[2]](#_Источники_1) и PyQt5 [[3]](#_Источники_1) для реализации графического интерфейса решения задачи о коммивояжере с использованием алгоритма ближайшего соседа. В программе реализован только один класс – **TSPApp**, который отвечает за создание интерфейса, обработку графических событий, построение графа, вычисление кратчайшего пути и визуализацию результатов. Класс включает в себя 15 методов, каждый из которых выполняет свою задачу в процессе работы приложения:

Таблица 1. PSO-Algorithm.py

|  |  |  |
| --- | --- | --- |
| Метод | Описание | Возвращаемое значение |
| \_\_init\_\_ | Инициализация интерфейса и параметров приложения. | None |
| initUI | Настройка графического интерфейса приложения, создание и размещение виджетов. | None |
| addEdge | Добавляет ребро между двумя выбранными вершинами, вычисляя расстояние между ними. | None |
| addNode | Добавляет новую вершину в граф по заданной позиции. | None |
| addNodeOrEdge | Определяет, следует ли добавить новую вершину или создать ребро между существующими вершинами, в зависимости от положения курсора. | None |
| clearGraph | Очищает граф, таблицу ребер, историю действий и текстовое поле с результатом. | None |
| drawArrow | Отрисовывает стрелку между вершинами, корректируя начало и конец линии, чтобы не накладываться на узлы. | None |
| drawSolution | Отображает найденный маршрут и вершины графа на отдельной сцене для визуализации решения. | None |
| findClickedNode | Определяет, на какую вершину произведён клик; возвращает идентификатор вершины или None, если клик не попал на узел. | int или None |
| getDistance | Возвращает расстояние между двумя вершинами, если соответствующее ребро существует, иначе — float('inf'). | float |
| lockColumns | Делает первые два столбца таблицы недоступными для редактирования. | None |
| redrawGraph | Перерисовывает граф и таблицу ребер после внесённых изменений. | None |
| solveTsp | Вычисляет кратчайший путь (гамильтонов цикл) с помощью алгоритма ближайшего соседа, измеряет время выполнения и выводит результаты. | None |
| undoAction | Отменяет последнее действие (добавление вершины или ребра) и обновляет граф. | None |
| updateEdgeWeight | Обновляет вес ребра в таблице при изменении значения, если ввод корректен. | None |

# Рекомендации пользователя

Программа позволяет решить задачу о коммивояжере с помощью метода ближайшего соседа. С ее помощью можно построить граф, добавить узлы и ребра, а затем найти кратчайший гамильтонов цикл.

1. Запустите программу, например, выполнив команду:

python The\_nearest\_neighbor\_method.pyw

1. В графическом интерфейсе:
   * Добавление узлов:

Щелкайте на области графа для создания новых узлов.

* + Добавление ребер:

Для соединения узлов выберите сначала один узел, затем второй – программа автоматически создаст ребро между ними.

* + Редактирование:

При необходимости измените вес ребра в таблице.

* + Выбор модификации:

Для использования модификации перебора начальных вершин установите галочку в «Использовать модификацию».

1. Нажмите кнопку **«Рассчитать»** для выполнения алгоритма. Программа вычислит кратчайший путь (гамильтонов цикл) и отобразит:
   * В поле **«Рассчитанный путь»** — последовательность вершин кратчайшего маршрута.
   * На графике — построенный маршрут с выделенными ребрами.
2. Используйте кнопку **«Отмена»** для удаления последнего действия или **«Очистить»** для сброса всего графа.

# Рекомендации программиста

Для корректной работы программы убедитесь, что установлены следующие компоненты и выполнены необходимые шаги:

* Требования:
  + Python версии 3.12.0 или выше.
  + Необходимые библиотеки: PyQt5, numpy.
* Шаги по установке:
  + Выполните команду для запуска программы:

python The\_nearest\_neighbor\_method.pyw

* + Проверьте, что графический интерфейс запускается корректно и результаты алгоритма (нахождение кратчайшего гамильтонова цикла) отображаются в соответствии с заданием.

# Исходный код программы

<https://github.com/FasterXaos/Algorithms_and_Data_Structures>

# Контрольный пример

PSO- The\_nearest\_neighbor\_method.py

1. Запуск программы и формирование графа

Для запуска программы откройте файл, содержащий код класса TSPApp. Программа откроет графический интерфейс, где в левой части окна расположены таблица для ввода данных о ребрах и панель для отображения вычисленного маршрута, в правой – область для построения графа (Рис. 2).

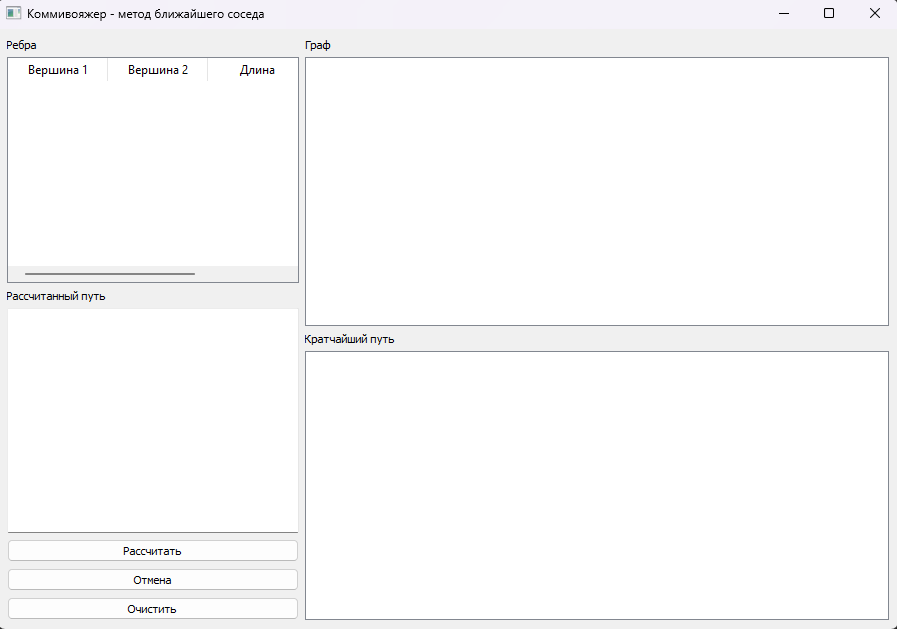


Рис 2. Пример окна программы

1. Ввод исходных данных

После запуска программы можно добавить вершины и соединить их ребрами следующим образом:

* Добавление вершин:

щёлкните в области графа (правой части окна) для создания новой вершины.

* Соединение вершин:

после выбора первой вершины щёлкните по другой, чтобы добавить ребро между ними. Вес ребра вычисляется автоматически как расстояние между центрами вершин.

1. Запуск алгоритма

После формирования графа нажмите кнопку «Рассчитать». Программа выполнит вычисление кратчайшего гамильтонова цикла методом ближайшего соседа, определяя маршрут, проходящий через все вершины и возвращающийся в исходную точку (Рис. 3).

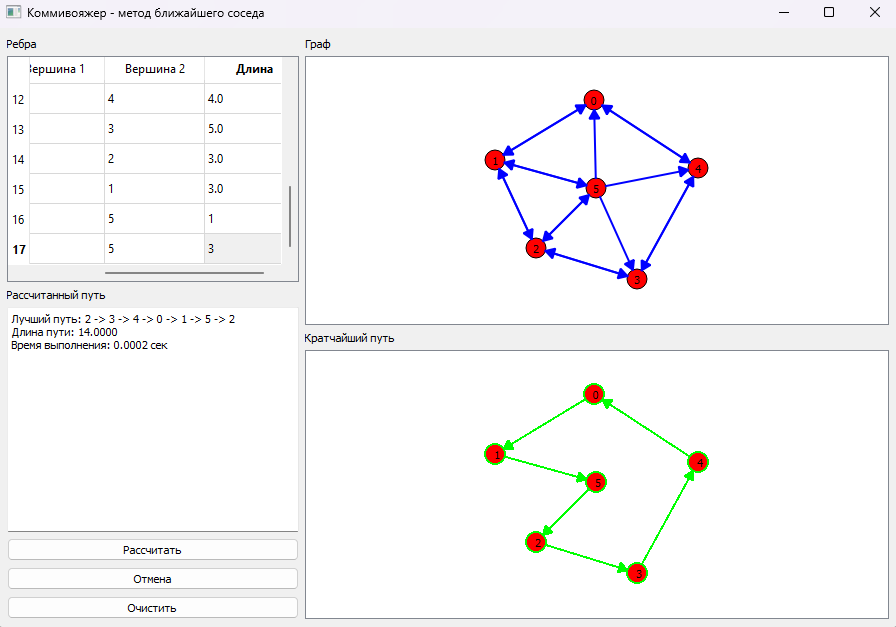


Рис 3. Пример результатов программы

1. Просмотр результатов

После завершения расчёта в нижней части окна будут выведены следующие данные:

* Найденный кратчайший маршрут (последовательность вершин);
* Общая длина маршрута с заданной точностью;
* Время выполнения алгоритма.

Дополнительно в правой части окна отображается графическое представление маршрута с выделенными вершинами и соединяющими их стрелками.

# Анализ результатов работы алгоритма

Анализ работы алгоритма ближайшего соседа демонстрирует его простоту и высокую скорость выполнения, однако за счет этого страдает точность найденных решений. На рис. 3 представлено, что метод успешно нашел минимальный путь в графе из шести вершин. Это подтверждает его эффективность в определенных условиях, когда граф имеет удачную структуру и сравнительно небольшое количество вершин.

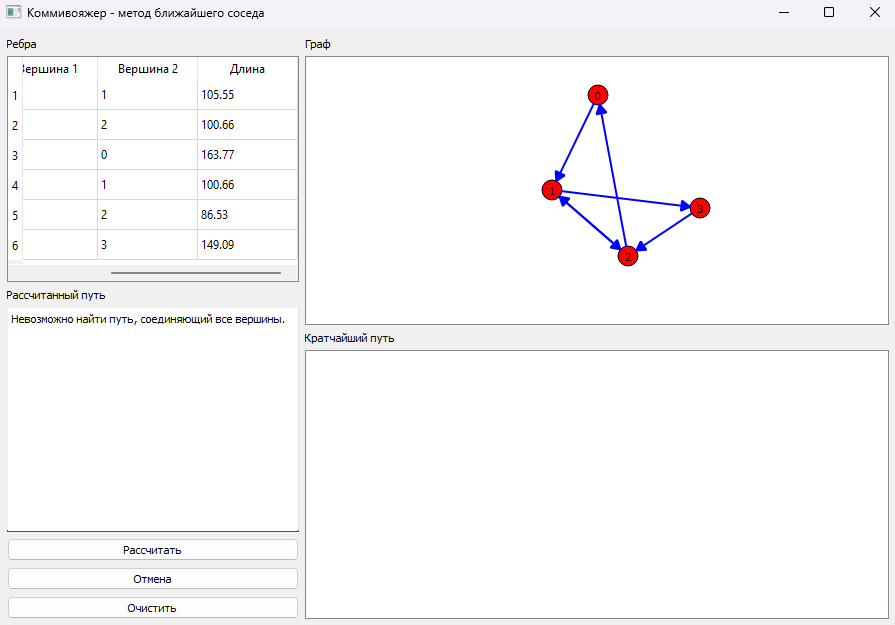


Рис 4. Пример результатов программы

Однако на рис. 4 видно, что алгоритм не смог найти существующий путь в другом примере. Это связано с его жадной стратегией выбора ближайшей вершины на каждом шаге, что может привести к тупиковым ситуациям или неоптимальным маршрутам. Такой подход не учитывает возможные последствия ранних решений, что делает алгоритм неподходящим для сложных графов с большим числом вершин или сильно варьирующимися весами рёбер.

Модификация выбора начальных вершин дает возможность чаще находить более оптимальное решение и иногда выходить из тупиковых ситуаций, когда, например, начиная с вершины «0» метод сразу попадает в тупик. На рис. 3 видно, что был выбран путь начиная с вершины «2», следовательно пути исходящие из вершин «0» и «1» были либо хуже наилучшего, либо не закольцовывались. Однако, такой подход приводит к куда большему количеству вычислений и времени выполнения.

В таблице 2 представлены результаты нахождения пути на случайных графах из тридцати вершин с весами от 0 (нет пути) до 4.

Таблица 1. PSO-Algorithm.py

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер графа | Расстояние без модификации | Время без модификации | Расстояние с модификацией | Время с модификацией |
| 1 | 43 | 0.0054 сек | 34 | 0.1655 сек |
| 2 | Не удалось найти путь | | 32 | 0.1635 сек |
| 3 | 37 | 0.0057 сек | 33 | 0.1643 сек |
| 4 | Не удалось найти путь | | 32 | 0.1607 сек |

Как показывают тесты, алгоритм работает предельно быстро без модификации перебора начальных вершин, но легко заходит в тупик. Перебор начальных вершин позволяет чаще находить путь и делает его более оптимальным, жертвуя куда более значительными затратами по времени.

Таким образом, алгоритм ближайшего соседа демонстрирует хорошую производительность за счёт низкой вычислительной сложности, но при этом имеет ограничения, связанные с точностью и возможностью попадания в тупиковые ситуации.

# Вывод

В ходе работы был реализован алгоритм ближайшего соседа для решения задачи коммивояжера, а также создан графический интерфейс для визуализации процесса поиска решения. Экспериментальные результаты показали, что метод эффективно справляется с некоторыми случаями, но его точность может варьироваться в разных графах, где он может не найти оптимальный или даже корректный маршрут.

Преимуществами алгоритма являются его простота и высокая скорость работы, что делает его подходящим для задач, где важна быстрая оценка маршрута без необходимости нахождения строго оптимального решения. Однако, в задачах с большим количеством вершин или сильно варьирующимися весами рёбер, стоит рассмотреть более точные алгоритмы.

# Источники

1. NumPy documentation // numpy.org URL: https://numpy.org/doc/stable/ (дата обращения: 04.03.2025).
2. PyQt5 Reference Guide // www.riverbankcomputing.com URL: https://www.riverbankcomputing.com/static/Docs/PyQt5/ (дата обращения: 04.03.2025).
3. Алгоритм ближайшего соседа в задаче коммивояжёра // ru.wikipedia.org URL: https://ru.wikipedia.org/wiki/Алгоритм\_ближайшего\_соседа\_в\_задаче\_коммивояжёра (дата обращения: 04.03.2025).