**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики-процессов управления**

**Программа бакалавриата**

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

**ОТЧЕТ**

**по лабораторной работе №6**

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

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

**Студент гр. 23Б15-пу**

**Черевко М. Е.**

**Преподаватель**

**Дик А.Г.**

**Санкт-Петербург**

**2025 г.**

Оглавление

1. [Цель работы 3](#_Toc193498872)
2. [Описание задачи (формализация задачи) 4](#_Toc193498873)
3. [Теоретическая часть 6](#_Toc193498874)
4. [Основные шаги программы 9](#_Toc193498875)
5. [Блок схема программы 13](#_Toc193498876)
6. [Описание программы 15](#_Toc193498877)
7. [Рекомендации пользователя 16](#_Toc193498878)
8. [Рекомендации программиста 17](#_Toc193498879)
9. [Контрольный пример 18](#_Toc193498880)
10. [Анализ 20](#_Toc193498881)
11. [Вывод 22](#_Toc193498882)
12. [Источники 23](#_Toc193498883)

# Цель работы

Разработка программного приложения с графическим интерфейсом для решения задачи коммивояжёра (Traveling Salesman Problem, TSP). Приложение должно обеспечивать возможность создания графа вручную, загрузки графа из файла, поиска оптимального маршрута с минимальной общей стоимостью и визуализации результатов, а также предоставлять функции отмены действий и сброса данных.

# Описание задачи (формализация задачи)

Задача коммивояжёра (Traveling Salesman Problem, TSP) заключается в нахождении кратчайшего замкнутого маршрута, проходящего через заданное множество точек (узлов) и возвращающегося в исходную точку. В контексте данной работы задача формализуется следующим образом:

1. **Входные данные:**

* Множество узлов — количество узлов, каждый из которых имеет уникальный идентификатор и координаты на плоскости.
* Направленный — множество рёбер, представляющих связи между узлами. Каждое ребро соединяет узел с узлом и имеет вес представляющий стоимость перехода от к
* Вес — целое неотрицательное число, задаваемое пользователем вручную или загружаемое из файла.

1. **Ограничения:**

* Граф может быть неполным, то есть не между всеми парами узлов обязательно существует ребро.
* Маршрут должен быть **гамильтоновым циклом**, то есть проходить через каждый узел ровно один раз и возвращаться в начальную точку.
* Если замкнутый маршрут невозможен (например, из-за отсутствия необходимых рёбер), решение не должно быть найдено.

1. **Целевая функция:**

Найти такой порядок посещения узлов , где , , а все различны, который минимизирует суммарную стоимость маршрута:

1. **Метод решения:**

Используется жадный алгоритм (Greedy Algorithm) с возможностью модификации:

* Если модификация включена, перебираются все возможные начальные узлы для поиска наилучшего маршрута.
* Если модификация отключена, начальный узел выбирается эвристически (например, узел с минимальной стоимостью связи с любым соседом).
* На каждом шаге выбирается следующий не посещённый узел с минимальной стоимостью связи от текущего узла.

1. **Выходные данные:**

* Оптимальный маршрут в виде последовательности идентификаторов узлов с возвращением в начальную точку.
* Общая стоимость маршрута .
* Визуализация маршрута на графическом интерфейсе.

Таким образом, задача сводится к построению программы, которая принимает граф, заданный пользователем, применяет алгоритм для поиска оптимального маршрута и отображает результат, учитывая указанные ограничения и требования.

# Теоретическая часть

* 1. **Постановка задачи коммивояжёра**

Задача коммивояжёра (Traveling Salesman Problem, TSP) является одной из классических задач комбинаторной оптимизации и теории графов. Она формулируется следующим образом: дан полный или частичный взвешенный граф — множество вершин (узлов), — множество рёбер, и каждому ребру сопоставлен вес , представляющий расстояние или стоимость перехода между вершинами . Необходимо найти гамильтонов цикл минимальной суммарной стоимости, то есть замкнутый маршрут, который проходит через каждую вершину ровно один раз и возвращается в исходную точку.

Математически задача сводится к минимизации функции:

где — перестановка вершин V, , a — число вершин.

TSP относится к классу -трудных задач, что означает, что точное решение для больших графов требует экспоненциального времени. Поэтому в практических приложениях часто используются приближённые или эвристические алгоритмы.

1. **Основные подходы к решению TSP**

Существует несколько подходов к решению задачи коммивояжёра:

1. **Точное решение:**
   * **Метод полного перебора (Brute Force):** Рассматриваются все возможные перестановки вершин, вычисляется стоимость каждого маршрута, выбирается минимальный. Временная сложность — что делает метод неприменимым для больших .
   * **Динамическое программирование (Held-Karp Algorithm):** Оптимизированный точный метод с временной сложностью , который использует мемоизацию для хранения промежуточных результатов.
2. **Эвристические методы:**
   * **Жадный алгоритм (Greedy Algorithm):** На каждом шаге выбирается ближайший непосещённый узел. Временная сложность — , но не гарантирует оптимального решения.
   * **Метод ближайшего соседа (Nearest Neighbor):** Разновидность жадного алгоритма, где маршрут строится, начиная с произвольной вершины, добавляя ближайшую непосещённую вершину.
   * **Метод минимального остовного дерева (MST):** Построение приближённого решения через минимальное остовное дерево графа.
3. **Метаэвристические методы:**
   * Генетические алгоритмы, имитация отжига, муравьиные алгоритмы и другие подходы, которые ищут близкое к оптимальному решение в разумное время.
   * В данной работе используется модифицированный жадный алгоритм, который сочетает простоту реализации с возможностью улучшения результата за счёт перебора начальных вершин.
4. **Описание используемого алгоритма**

В разработанном приложении применяется жадный алгоритм с опциональной модификацией:

1. **Базовый жадный алгоритм:**
   * Выбирается начальная вершина (например, с минимальной стоимостью связи с любым соседом).
   * Из текущей вершины выбирается непосещённая вершина с минимальной стоимостью ребра.
   * Процесс повторяется, пока все вершины не будут посещены.
   * Добавляется ребро от последней вершины к начальной для замыкания цикла.
   * Временная сложность:
2. **Модификация:**
   * Если опция "Использовать модификацию" включена, алгоритм запускается для каждой вершины графа как начальной.
   * Из всех полученных маршрутов выбирается тот, у которого суммарная стоимость минимальна.
   * Временная сложность возрастает до , но вероятность нахождения более оптимального решения увеличивается.
3. **Особенности реализации:**
   * Граф представлен в виде словаря смежности, где для каждой вершины хранится список соседей с соответствующими весами.
   * Алгоритм проверяет наличие рёбер для замыкания маршрута; если маршрут невозможен, возвращается сообщение об ошибке.
4. **Графический интерфейс и визуализация**

Для реализации приложения используется библиотека Tkinter, которая предоставляет инструменты для создания графического интерфейса:

* **Ввод данных:** Пользователь может добавлять узлы кликом мыши, задавать связи с весами через контекстное меню и загружать граф из текстового файла.
* **Хранение данных:** Узлы хранятся как словари с координатами и идентификаторами, связи — как кортежи (начало, конец, вес, идентификатор линии).
* **Визуализация:** Узлы отображаются как окружности, связи — как стрелки. Результат (оптимальный маршрут) рисуется на отдельном холсте красным цветом.
* **Интерактивность:** Поддерживаются функции отмены действий, сброса графа и переключения режима модификации.
* **Cохранение результата:** сохраняется граф результата и полученные значения

1. **Преимущества и ограничения подхода**

**Преимущества:**

* Простота реализации и понимания алгоритма.
* Быстрая работа на небольших графах .
* Возможность визуального контроля процесса и результата.
* Гибкость ввода данных (ручной и из файла).

**Ограничения:**

* Жадный алгоритм не гарантирует глобального оптимума, особенно на сложных графах.
* Модификация увеличивает время выполнения на больших графах.
* Отсутствие проверки на связность графа может привести к невозможности построения маршрута.

# Основные шаги программы

Программа, реализованная в виде класса TSPApp с использованием библиотеки Tkinter, представляет собой приложение для решения задачи коммивояжёра. Ниже приведены основные шаги её работы, соответствующие структуре кода и функциональности:

1. **Инициализация приложения**

* **Цель:** Подготовка графического интерфейса и начальных данных.
* **Реализация:** Метод \_\_init\_\_:
  1. Создаётся главное окно приложения с заголовком "Решение задачи маршрута" и размером 1000x750 пикселей.
  2. Инициализируются списки для хранения узлов (self.nodes), связей (self.connections), истории действий (self.history) и удалённых узлов (self.deleted\_nodes).
  3. Устанавливаются начальные значения: node\_id\_tracker = 0, active\_node = None, флаг модификации (use\_modification\_var = True).
  4. Вызывается метод \_setup\_ui для создания интерфейса и \_center\_window для центрирования окна на экране.

1. **Создание пользовательского интерфейса**

* **Цель:** Обеспечение удобного ввода данных и управления программой.
* **Реализация:** Метод \_setup\_ui:
  1. Создаются два основных холста: input\_area (ввод графа) и output\_area (отображение результата).
  2. Формируется таблица связей (edge\_table) с колонками "Начало", "Конец", "Вес".
  3. Добавляются элементы управления: кнопки "Найти маршрут", "Назад", "Сбросить", "Загрузить граф" и переключатель модификации.
  4. Устанавливаются привязки событий: клик левой кнопкой мыши для добавления узлов, правой — для создания связей.

1. **Добавление узлов**

* **Цель:** Создание вершин графа пользователем.
* **Реализация:** Метод \_place\_node:
  1. При клике левой кнопкой мыши на холсте input\_area проверяется минимальное расстояние до существующих узлов (MIN\_SPACING).
  2. Если расстояние достаточно, создаётся новый узел с уникальным идентификатором (переиспользуется из deleted\_nodes или увеличивается node\_id\_tracker).
  3. Узел рисуется как зелёная окружность с белой меткой идентификатора.
  4. Информация об узле добавляется в self.nodes и self.history.

1. **Создание связей**

* **Цель:** Установка направленных рёбер между узлами с заданными весами.
* **Реализация:** Метод \_pick\_node\_for\_link:
  1. При клике правой кнопкой мыши выбирается первый узел (если active\_node пуст), отображается временная метка.
  2. При повторном клике на другом узле запрашивается вес связи через диалоговое окно (askinteger).
  3. Если связь уже существует, предлагается изменить вес; иначе создаётся новая связь.
  4. Связь отображается стрелкой (\_render\_directed\_link), добавляется в self.connections, self.edge\_table и self.history.

1. **Загрузка графа из файла**

* **Цель:** Импорт графа из текстового файла.
* **Реализация:** Метод \_load\_graph:
  1. Открывается диалог выбора файла (filedialog.askopenfilename).
  2. Файл читается построчно: узлы (формат: id,x,y) и рёбра (формат: from\_id,to\_id,weight) разделяются секциями # Nodes и # Edges.
  3. Узлы и связи отрисовываются на холсте, добавляются в соответствующие структуры данных.
  4. Обрабатываются ошибки (дубликаты узлов, некорректный формат) с выводом сообщений через messagebox.

1. **Решение задачи коммивояжёра**

* **Цель:** Поиск оптимального маршрута с минимальной стоимостью.
* **Реализация:** Метод \_solve\_tsp:
  1. Проверяется наличие минимум двух узлов.
  2. Формируется словарь смежности (graph\_data) на основе self.connections.
  3. Если модификация включена (use\_modification\_var): перебираются все узлы как начальные; иначе выбирается один эвристически (\_choose\_start\_node).
  4. Для каждого начального узла:
     + Жадный алгоритм добавляет ближайший непосещённый узел с минимальным весом.
     + Проверяется возможность замыкания маршрута (наличие ребра к начальной точке).
     + Сохраняется маршрут с минимальной общей стоимостью.
  5. Результат отображается текстом в result\_container и визуально на output\_area (\_display\_optimal\_route).

1. **Визуализация результата**

* **Цель:** Отображение найденного маршрута.
* **Реализация:** Метод \_display\_optimal\_route:
  1. Очищается холст output\_area.
  2. Рисуются узлы из маршрута как зелёные окружности с метками.
  3. Рисуются связи между последовательными узлами красными стрелками (\_render\_link\_on\_output).
  4. Добавляется финальная связь от последнего узла к первому.

1. **Отмена последнего действия**

* **Цель:** Возврат к предыдущему состоянию графа.
* **Реализация:** Метод \_revert\_last\_step:
  1. Извлекается последнее действие из self.history.
  2. Для "node\_added": удаляется узел, его фигуры стираются, идентификатор добавляется в deleted\_nodes.
  3. Для "link\_added": удаляется связь из self.connections и edge\_table, стирается стрелка.
  4. Для "link\_updated": восстанавливается старый вес связи.

1. **Сброс всех данных**

* **Цель:** Очистка графа и возврат к начальному состоянию.
* **Реализация:** Метод \_reset\_all:
  1. Удаляются все элементы из edge\_table, input\_area, output\_area, result\_container.
  2. Очищаются списки nodes, connections, сбрасываются node\_id\_tracker и active\_node.
  3. Отображается сообщение "Очищено".

1. **Сохранение результатов**

* **Цель:** Сохраняет найденный маршрут в файл .txt
* **Реализация**: **Метод** \_save\_result
  1. Проверяет, есть ли решение (если нет — ругается)
  2. Показывает окно выбора папки
  3. Пишет в файл: координаты точек, связи между ним, итоговую длину маршрута
  4. Отчитывается: сохранилось или ошибка

1. **Запуск программы**

* **Цель:** Инициализация и запуск основного цикла приложения.
* **Реализация:** Блок if \_\_name\_\_ == "\_\_main\_\_":
  + Создаётся экземпляр Tk, инициализируется TSPApp, запускается mainloop для обработки событий.

# Блок схема программы

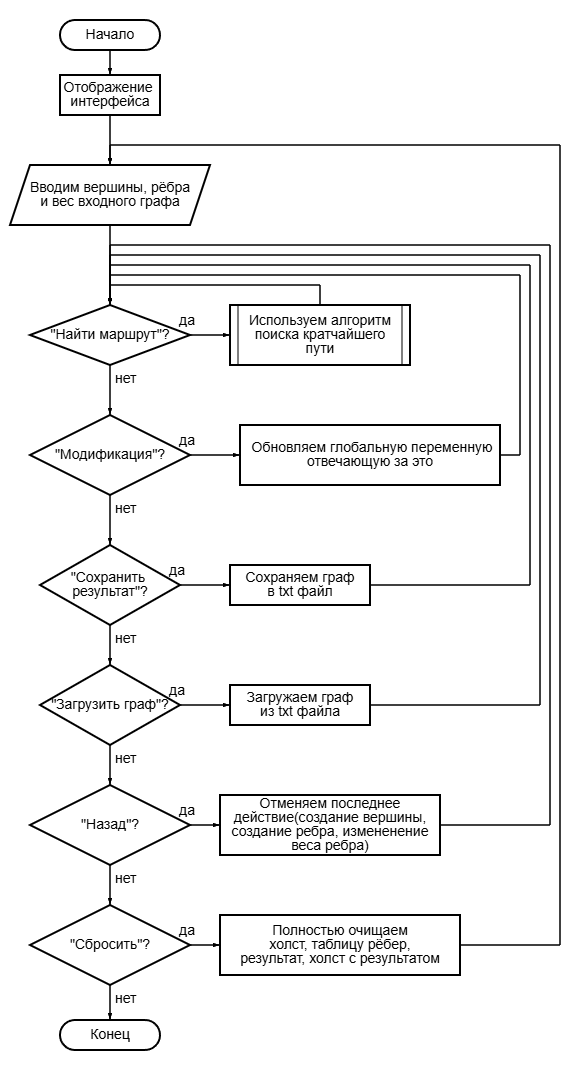
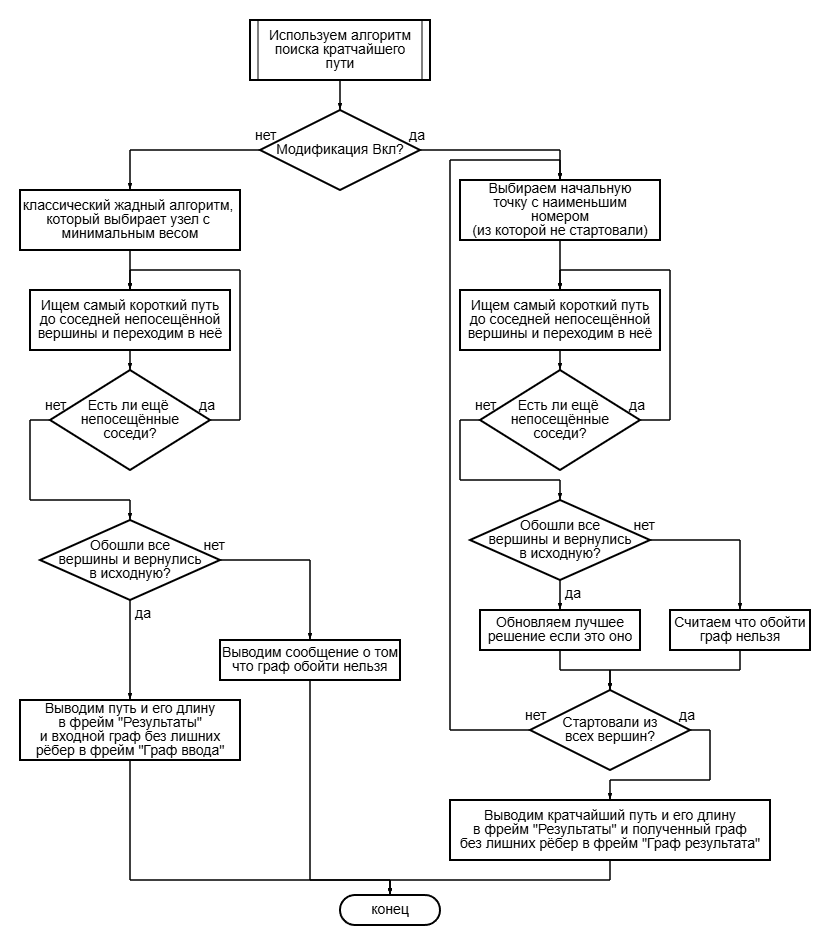


Рис. 3 Блок-схема основной программы

 Рис. 4 Блок-схема подпрограммы

# Описание программы

Программная реализация задачи коммивояжёра разработана на языке Python 3.12.7 с применением библиотек tkinter, ttk и math. Приложение представляет собой графический интерфейс, позволяющий пользователю формировать и изменять граф, состоящий из узлов и связей. На основе введённой информации программа решает задачу коммивояжёра, определяя кратчайший маршрут, охватывающий все узлы. В интерфейсе реализованы функции добавления узлов, создания рёбер между ними с указанием весов, а также возможности отмены последнего шага и полного сброса данных. Программа визуализирует результат, отображая оптимальный маршрут и его общую стоимость.

Таблица 1. nearest\_neighbor\_method\_algorithm.py

|  |  |  |
| --- | --- | --- |
| Функция | Описание | Возвращаемое значение |
| \_\_init\_\_ | Инициализирует приложение, создавая интерфейс и переменные. | None |
| \_setup\_ui | Создает пользовательский интерфейс (окна, кнопки, холсты) | None |
| \_center\_window | |  | | --- | |  |  |  | | --- | | Центрирует главное окно на экране | | None |
| \_load\_graph | |  | | --- | | Рисует кратчайший путь, найденный для задачи коммивояжёра. |  |  | | --- | |  | | None |
| \_place\_node | |  | | --- | |  |  |  | | --- | | Добавляет новый узел на холст по клику мыши | | None |
| \_pick\_node\_for\_link | Выбирает узлы для связи (по правому клику) | None |
| \_render\_directed\_link | |  | | --- | |  |  |  | | --- | | Очищает все данные (все вершины и рёбра), сбрасывает интерфейс. | | None |
| \_display\_optimal\_route | Показывает оптимальный маршрут на выходном холсте | None |
| \_render\_link\_on\_output | Отрисовывает связь между узлами на выходном холсте | None |
| \_choose\_start\_node | Определяет начальный узел для алгоритма (по минимальному весу) | dict (данные узла) |
| \_solve\_tsp | Решает задачу коммивояжера (TSP) и отображает результат | None |
| \_save\_result | Сохраняет найденный маршрут в файл | None |
| \_revert\_last\_step | Отменяет последнее действие (добавление узла/связи) | None |
| \_reset\_all | Полностью сбрасывает приложение в исходное состояние | None |

# 

# Рекомендации пользователя

Для корректного выполнения программы следуйте приведённым шагам:

* **Добавление вершин**:

Для добавления вершины на граф, кликните левой кнопкой мыши на поле "Граф ввода". Каждое нажатие создаст новую вершину с уникальным идентификатором.

* **Создание рёбер**:

Для создания рёбер между вершинами, кликните правой кнопкой мыши по одной вершине, чтобы выбрать её, затем кликните правой кнопкой мыши по другой вершине для создания ребра между ними. После этого вас попросят ввести вес рёбра.

* **Просмотр и изменение весов рёбер**:

Если ребро между двумя вершинами уже существует, вы можете изменить его вес. Программа предложит ввести новый вес, если вы выбрали одну из вершин, между которыми уже есть ребро.

* **Отмена последнего действия**:

Для отмены последнего действия (добавление вершины или рёбра) используйте кнопку "Назад". Это позволит вернуть граф в состояние до последнего изменения.

* **Очистка графа**:

Для очистки всего графа (удаления всех вершин и рёбер) нажмите кнопку "Сбросить". После этого граф будет полностью очищен, и вы можете начать с нуля.

* **Расчёт оптимального пути**:

После добавления всех вершин и рёбер, нажмите кнопку "Рассчитать". Программа найдет оптимальный путь для задачи коммивояжера и отобразит результат на поле "Граф результата". На поле также будет показан вес маршрута, а результаты расчета будут отображены в блоке "Результат". Здесь будет показан оптимальный путь и общий вес маршрута.

# Рекомендации программиста

Для корректного функционирования программы рекомендуется выполнить следующие действия:

1. **Установите необходимые библиотеки**:
   * Убедитесь, что у вас установлены библиотеки **tkinter, math, ttk** и **tkinter.simpledialog**. **thinker** идет в стандартной поставке Python, но если она отсутствует, вы можете установить ее через пакетный менеджер вашей операционной системы.
2. **Проверьте версию Python**:
   * Рекомендуется использовать **Python** версии **3.7** или выше, чтобы избежать возможных проблем с совместимостью библиотек и функциональностью **tkinter**.
3. **Проверка функций и интерфейса**:
   * Убедитесь, что все элементы интерфейса (кнопки, текстовые поля и таблицы) работают корректно. Попробуйте вводить различные значения и проверить, что результаты рассчитываются и отображаются правильно.

**Код программы:**

<https://github.com/FraaaM/Solving_the_Traveling_Salesman_Problem_Using_the_Nearest_Neighbor_Method>

# Контрольный пример

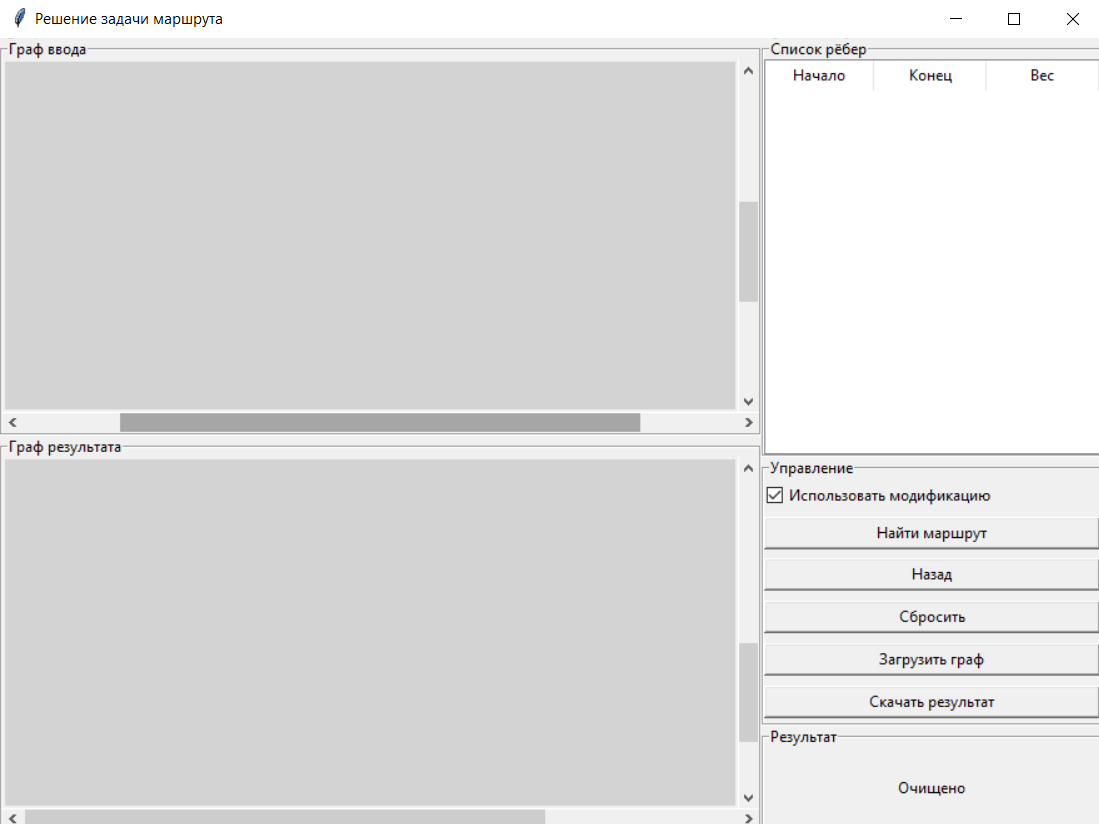
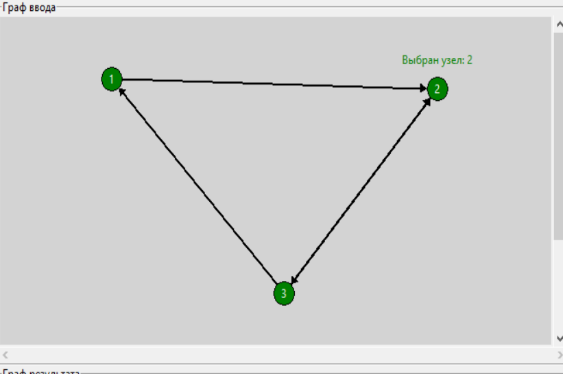
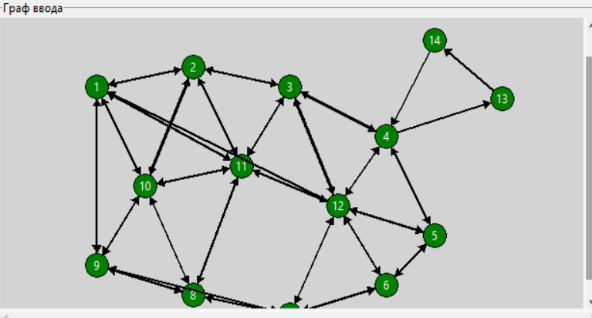
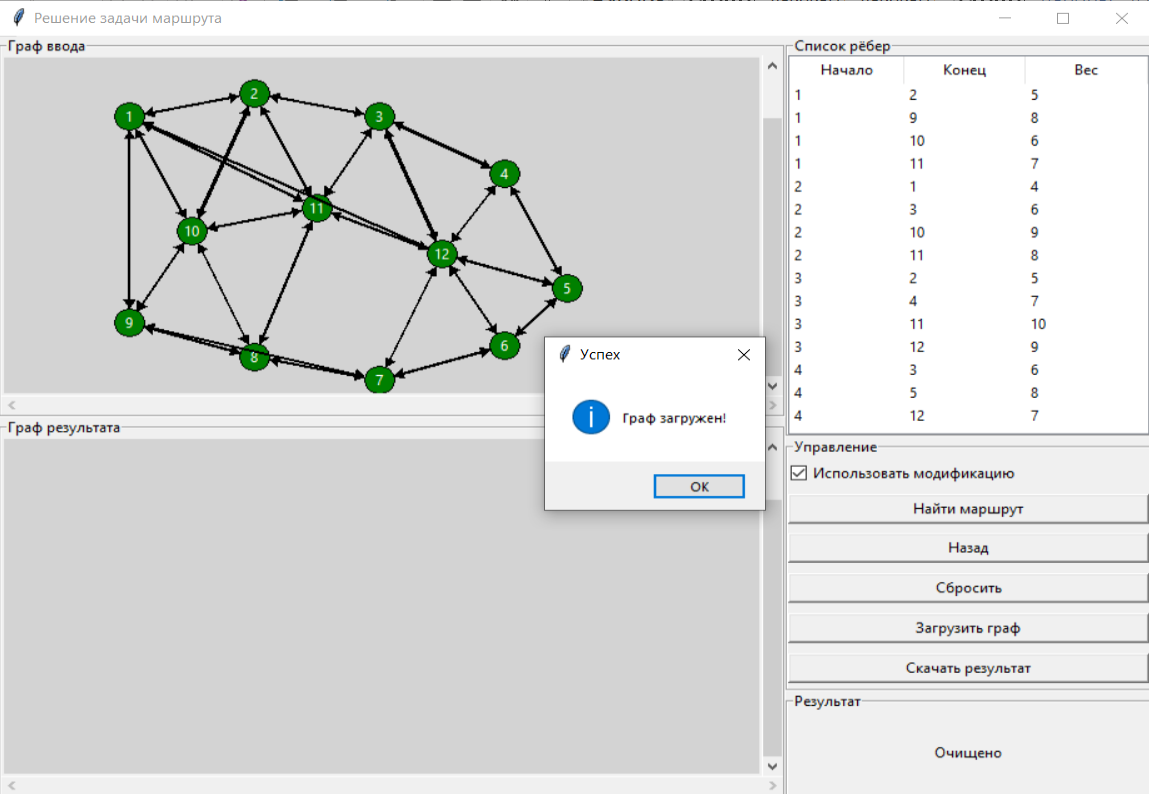
* **Запуск программы:** Для запуска программы используйте файл nearest\_neighbor\_method\_algorithm.py. Программа откроет графический интерфейс (Рис. 4).

Рис. 3 Графический интерфейс программы

* **Инициализация частиц:** В фрейме “Граф ввода” необходимо при помощи нажатий на правую и левую кнопки мыши изобразить нужный граф. (Рис. 4)
* **Загрузка граф:** так же можно загрузить граф из текстового документа и изменять его. (Рис. 4)
*  Рис. 4 Ввод графа
* **Запуск алгоритма ближайшего соседа с модификацией:** При нажатии на кнопки “Найти маршрут” произойдёт запуск алгоритма и мы получим результат в фрейме “Результаты и его изображение в фрейме “Граф результата”. (Рис. 5 / Рис. 6)

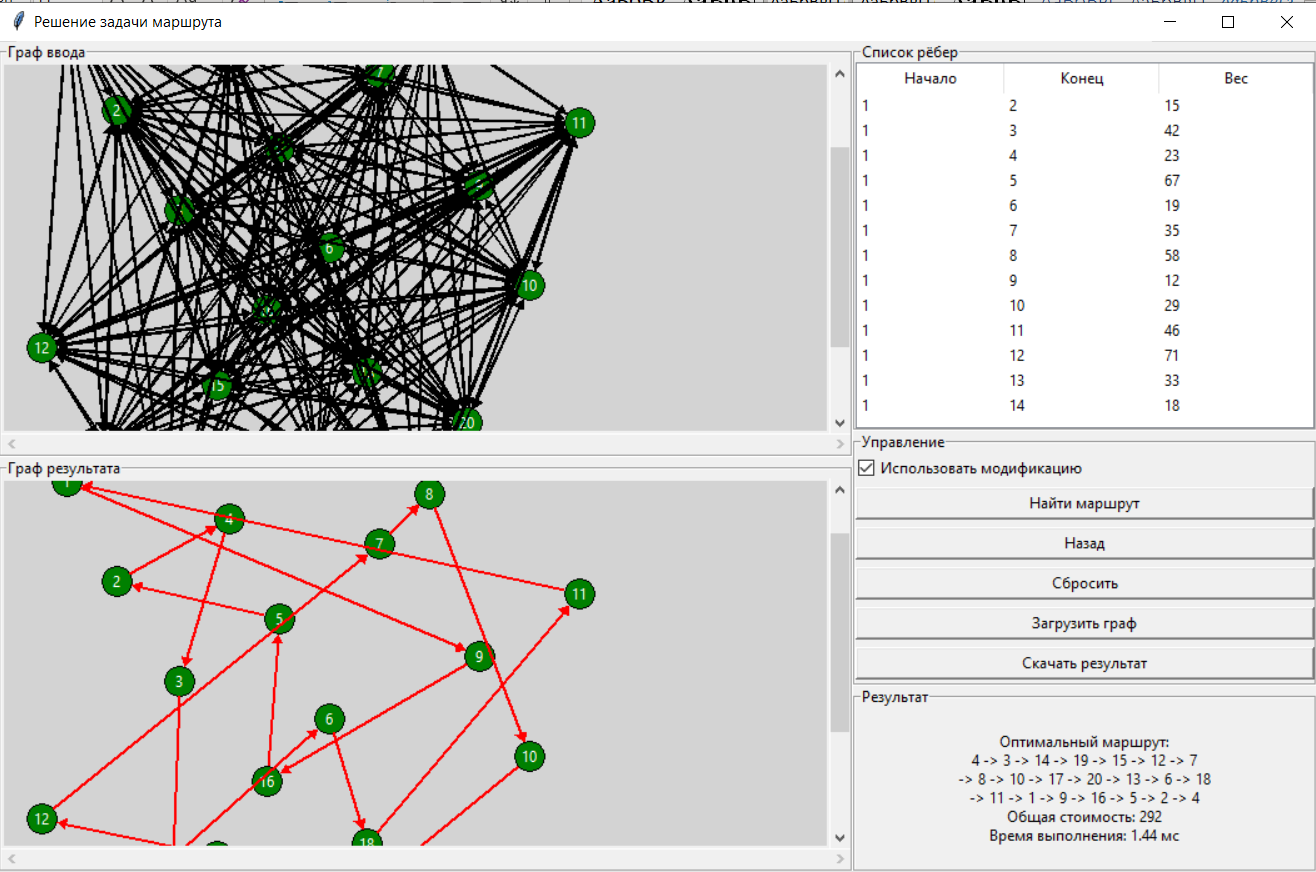
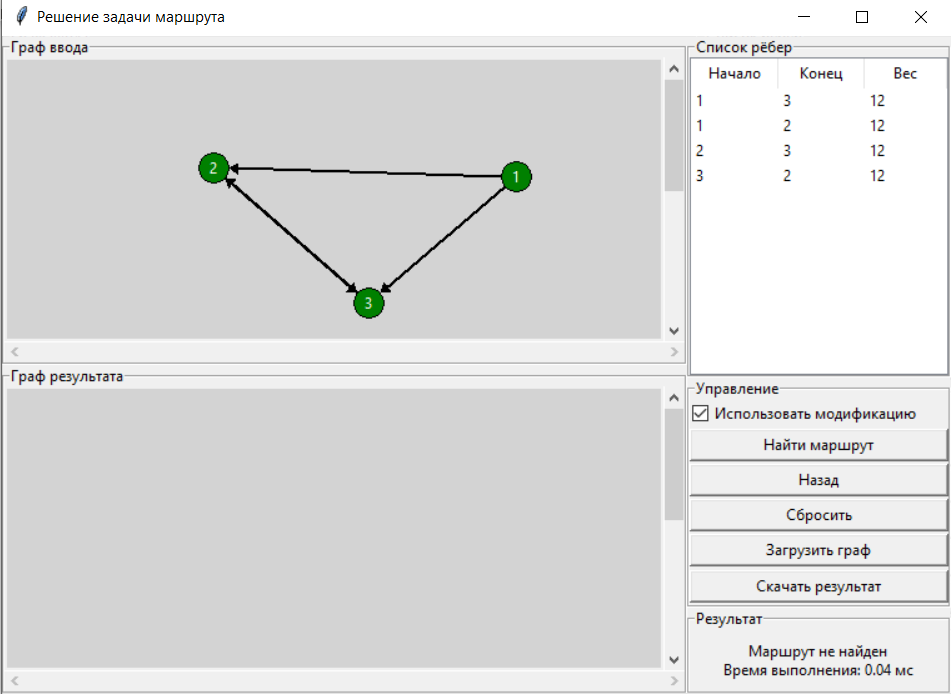
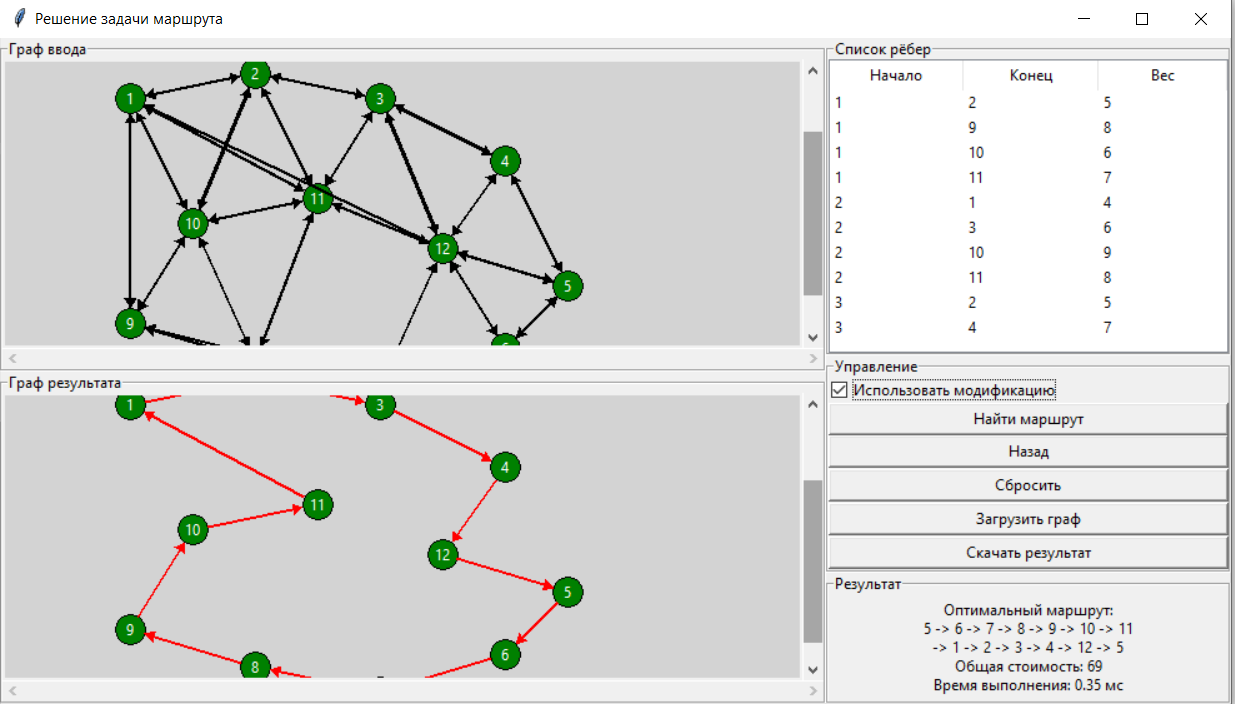


Рис. 5 Результат с модификацией

Рис. 6 Неудачный результат

# Анализ работы алгоритма с модификацией и без

В качестве модификации использовался **метод многократного запуска алгоритма ближайшего соседа** с каждой вершины графа. Вместо того, чтобы начать обход с одной вершины по классическому жадному алгоритму, вычисления выполняются несколько раз, каждый раз начиная с новой вершины и выбирается оптимальный результат.

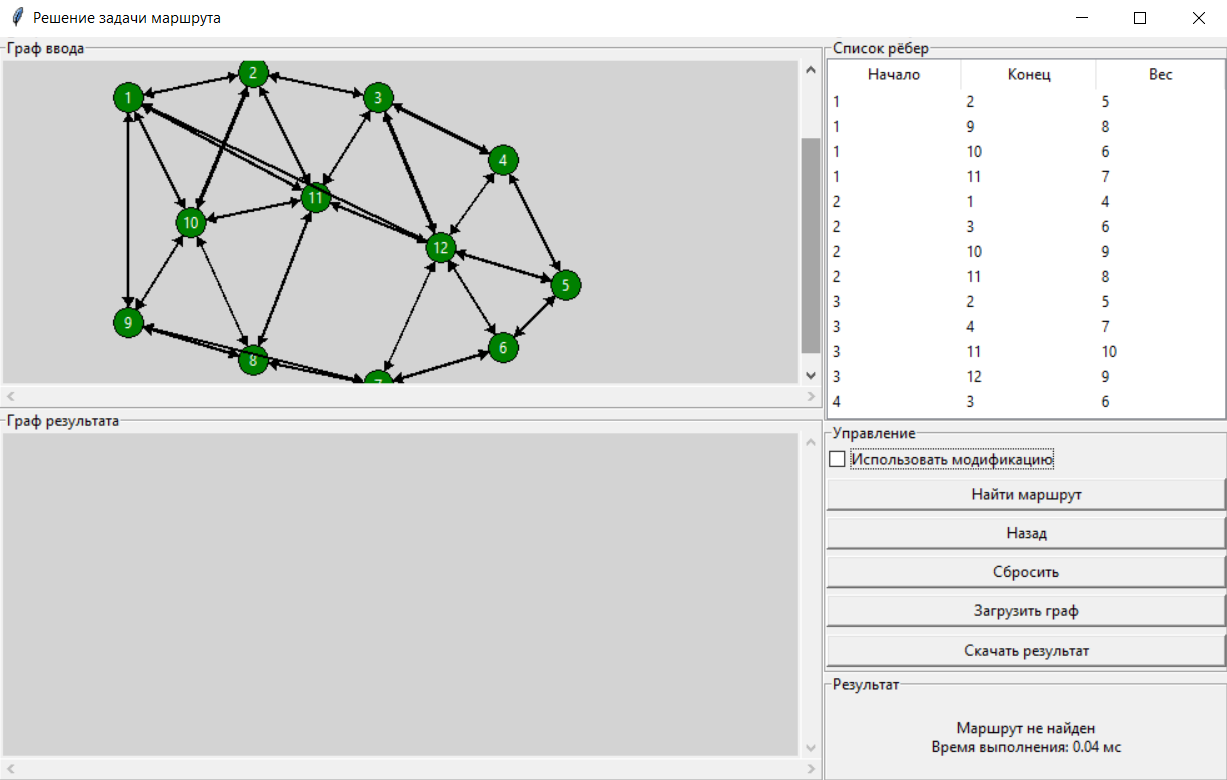
Рис. 7 Результат работы с модификацией

Рис. 8 Результат работы без модификации

Из (Рис. 7) видно, что алгоритм с модификацией, который начинает обход с каждой вершин, выбирает наилучший результат и находит гамильтонов цикл без проблем.

Алгоритм без модификации (Рис. 8) застревает в вершине, поскольку он не может корректно обработать ситуацию с длинным ребром между вершинами, это приводит к тому, что алгоритм выбирает путь с меньшими весами, но не учитывает возможность прохождения через другую вершину в дальнейшем.

Сравнение работы алгоритма с модификацией и без на графах с большим количеством вершин:

Таблица 2. Сравнение работы алгоритма с модификацией и без

|  |  |  |  |
| --- | --- | --- | --- |
| Модификация | Время (в мс) | Длина пути | Кол-во вершин |
| 1) С модификацией | 0.14 | 23 | 6 |
| 1)Без модификации | 0.04 | 23 | 6 |
| 2) С модификацией | 1.43 | 292 | 20 |
| 2) Без модификации | 0.10 | 302 | 20 |
| 3) С модификацией | 8.03 | 336 | 40 |
| 3) Без модификации | 0.23 | 530 | 40 |

Из представленных данных можно сделать следующий вывод:

1. **Время работы:** Модификация алгоритма, занимает больше времени, поскольку она исследует больше возможных путей. Например, на графах с 20 вершинами время увеличивается с 0.10 до 1.43, а на графах с 40 вершинами — с 0.23 до 8.3. Это подтверждает, что добавление этой модификации усложняет вычисления.
2. **Длина пути:** С модификацией алгоритм находит более короткие пути. Например, для графа с 20 вершинами длина пути с модификацией — 292, а без неё — 302. Разница не значительна, но другая тенденция наблюдается и на графе с 40 вершинами: с модификацией путь составляет 336, а без — 530. Это подтверждает, что старт из всех вершин помогает найти более оптимальный маршрут.
3. **Общее заключение:** Модификация улучшает качество найденного пути, но за счет увеличения времени вычислений. В случае, когда важна скорость, можно использовать стандартный алгоритм, однако для достижения лучшего результата в плане длины пути стоит использовать модификацию, несмотря на её более высокое время работы.

# Вывод

В ходе работы был реализован и протестирован жадный алгоритм решения задачи коммивояжёра, а также его модифицированный вариант, позволяющий избежать локального застревания. Эксперименты показали, что стандартная версия алгоритма не всегда находит корректный гамильтонов цикл, так как жадный подход выбирает кратчайшее доступное ребро на каждом шаге, что может привести к ситуации, когда продолжение маршрута становится невозможным.

Таким образом, предложенная модификация делает жадный алгоритм более универсальным, однако его применимость остаётся ограниченной для сложных графов из-за увеличения времени вычислений. Выбор метода решения задачи должен основываться на компромиссе между точностью и скоростью вычислений.

# Источники

* Редактор блок-схем.

[*https://programforyou.ru/block-diagram-redactor*](https://programforyou.ru/block-diagram-redactor)

*дата обращения: (08.03.2025)*

* tkinter — Библиотека для создания графических интерфейсов в Python.

[*https://docs.python.org/3/library/tkinter.html*](https://docs.python.org/3/library/tkinter.html%20)

*дата обращения: (08.03.2025)*

* math — Библиотека для работы с математическими функциями.

[*https://docs.python.org/3/library/math.html*](https://docs.python.org/3/library/math.html)

*дата обращения: (08.03.2025)*

* openpyxl — Библиотека для записи и чтения файлов Excel.

[*https://openpyxl.readthedocs.io/en/stable/*](https://openpyxl.readthedocs.io/en/stable/)

*дата обращения: (16.03.2025)*