**Санкт-Петербургский государственный УНИВЕРСИТЕТ**

**Факультет прикладной математики и процессов управления**

**отчет**

**по лабораторной работе №1**

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

**на тему «Алгоритм ближайшего соседа»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 22Б16 |  | Шувалов Ф.В. |
| Преподаватель |  | Дик А.Г. |

**Санкт-Петербург**

**2023 г.**

**Оглавление**

[1. Цель работы 3](#_Toc152621849)

[3. Введение 3](#_Toc152621850)

[4. Теоретическая часть 3](#_Toc152621851)

[5. Схема выполнения алгоритма 4](#_Toc152621852)

[7. Представление программы на языке программирования: 5](#_Toc152621853)

[8. Спецификация программы 6](#_Toc152621854)

[9. Раскрытие смысла работы алгоритма: 6](#_Toc152621855)

[10. Контрольный пример 7](#_Toc152621856)

[11. Результаты тестирования 8](#_Toc152621857)

[12. Анализ результатов тестирования 9](#_Toc152621858)

[13. Вывод 10](#_Toc152621859)

[14. Литература 10](#_Toc152621860)

# **Цель работы**

Цель работы – исследование особенностей решения задачи о коммивояжере с помощью алгоритма ближайшего соседа.

1. **Задачи**
2. Необходимо формализовать задачу о коммивояжере с помощью

алгоритма ближайшего соседа.

1. Подготовить контрольный пример, используя взвешенный орграф
2. Найти кратчайший гамильтонов цикл.

# **Введение**

Название «Задача о коммивояжере» устойчиво закрепилось за одной из самых интересных, практически значимых и одновременно сложных задач теории графов. Задача, берущая свое начало из работ Гамильтона, состоит в определении кратчайшего гамильтонова цикла в графе. Ее решение связано с решением задачи о назначениях и с задачей об остове наименьшего веса.

Исследование алгоритма ближайшего соседа для задачи нахождения гамильтонова цикла имеет важное значение, так как этот алгоритм является простым и интуитивно понятным, что облегчает его понимание и реализацию. Кроме того, изучение его свойств и ограничений может помочь в разработке более эффективных методов решения задачи нахождения гамильтонова цикла в различных приложениях, таких как маршрутизация в компьютерных сетях или оптимизация планирования маршрутов для транспортных средств.

# **Теоретическая часть**

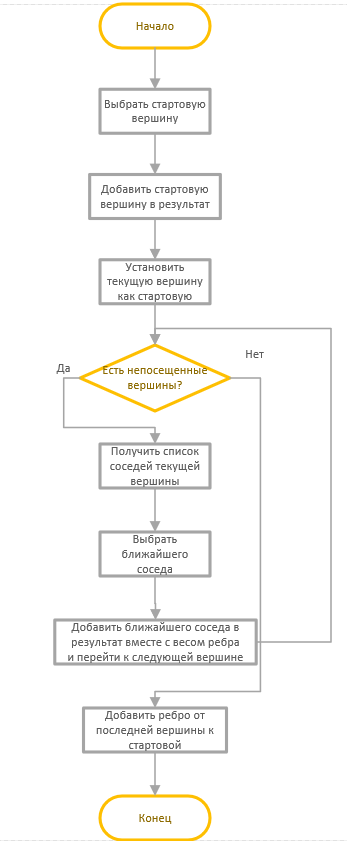
**Необходимые термины:**

Вот список терминов, связанных с алгоритмом ближайшего соседа для решения задачи о нахождении гамильтонова цикла:

* 1. Гамильтонов цикл: Путь в графе, который проходит через каждую вершину ровно один раз и возвращается в начальную вершину.
  2. Алгоритм ближайшего соседа (Nearest Neighbor Algorithm): Простой алгоритм для нахождения приближенного решения задачи коммивояжера, который начинает с произвольной начальной вершины и на каждом шаге выбирает ближайшую доступную вершину для посещения.
  3. Граф: Математическая структура, состоящая из вершин и рёбер, которые соединяют эти вершины.
  4. Вершина графа: Один из узлов или точек в графе.
  5. Ребро графа: Связь между двумя вершинами в графе.
  6. Цикл: Путь в графе, который начинается и заканчивается в одной и той же вершине.
  7. Приближенный алгоритм: Алгоритм, который может дать приближенное решение оптимизационной задачи за разумное время, но без гарантии оптимальности.

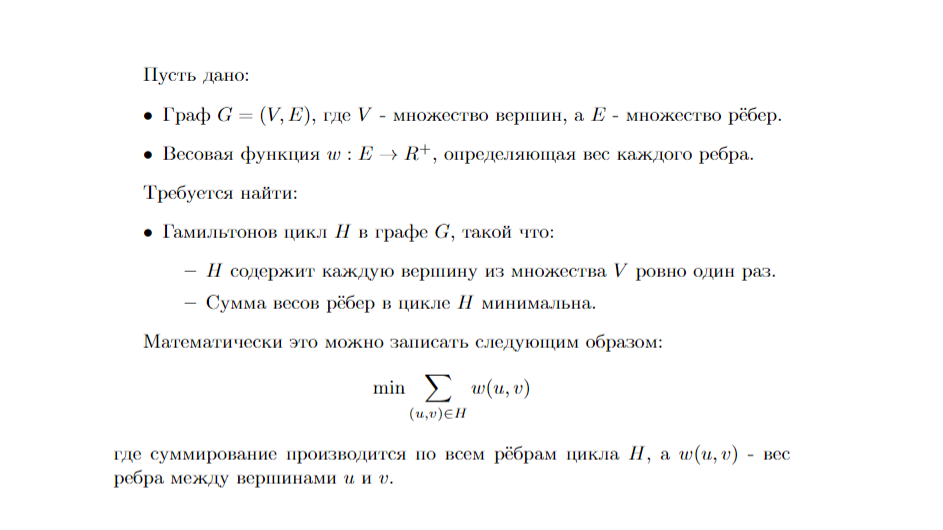
# **Схема выполнения алгоритма**

На рисунке 5.1 представлена блок-схема алгоритма.



*Рисунок 5.1 Блок-схема алгоритма*

1. **Формализация задачи**



*Рисунок 6.1 Формализация задачи*

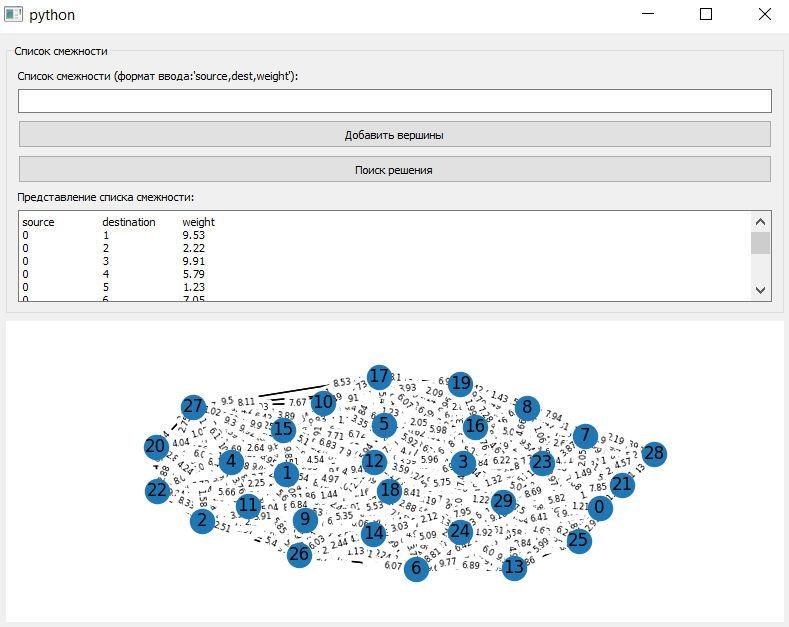
# **Представление программы на языке программирования**

Алгоритм ближайшего соседа реализован на выбранном языке программирования (Python3) с использованием объектно-ориентированного подхода. Программа состоит из двух основных классов: `ClosestNeighborAlgo` и `MCNA`. Класс `ClosestNeighborAlgo` представляет реализацию алгоритма ближайшего соседа, а класс ` MCNA ` представляет его модификацию.

# **Раскрытие смысла работы алгоритма**

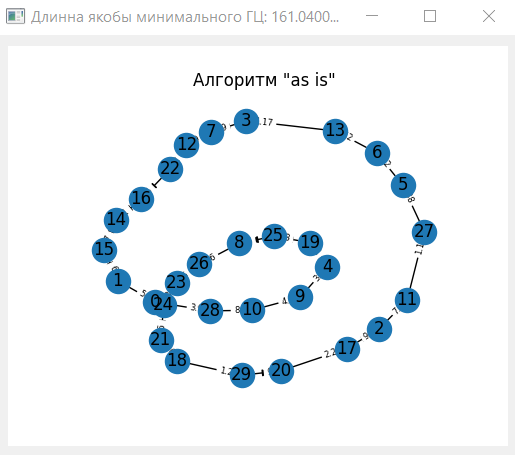
Алгоритм ближайшего соседа для задачи о нахождении гамильтонова цикла представляет собой простой метод, основанный на локальном выборе ближайшей доступной вершины на каждом шаге. На первом шаге выбирается произвольная начальная вершина, а затем на каждом последующем шаге выбирается ближайшая не посещенная вершина и добавляется к циклу. Этот процесс продолжается до тех пор, пока все вершины не будут посещены. Хотя алгоритм является приближенным и не гарантирует оптимального решения, его простота и относительная эффективность делают его полезным в различных практических сценариях. Однако следует отметить, что в худшем случае сложность алгоритма составляет O(n^2), что делает его неэффективным для крупных графов.

# **Контрольный пример**



*Рисунок 10.1 Визуализация программы*

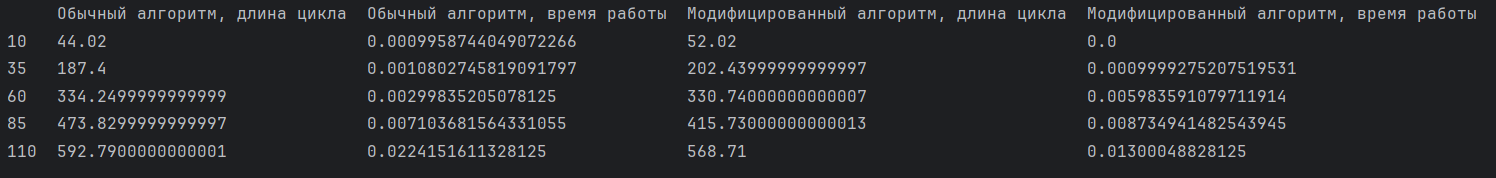
На рисунке 10.1 изображен пользовательский интерфейс программы. В левой части окна пользователь может вводить параметры, такие как: *граф***.**

****

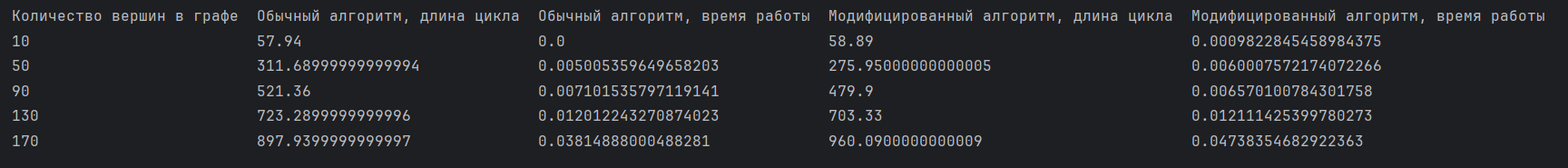
*Рисунок 10.2 результат работы алгоритма*

На рисунке 10.2 изображен результат работы алгоритма.

1. Результаты тестирования



*Рисунок 10.1 Результаты тестирования (запуск 1)*



*Рисунок 10.2 Результаты тестирования (запуск 2)*

1. Модификация

Алгоритм модифицирован добавлением эвристики в операцию выбора вершины: с большим шансом выбирается вершина с наименьшей степенью. Также было модифицировано стартовое условие: алгоритм стартует с вершины с наибольшим количеством ребер.

1. Анализ результатов тестирования

Анализ результатов тестирования выявил следующие ключевые особенности алгоритма NNA:

1. Незначительное ухудшение скорости сходимости при увеличении размера (количества ребер и вершин) графа.

# **Вывод**

При выполнении данной работы были получены все необходимые навыки и знания по реализации алгоритма ближайшего соседа. В ходе реализации задачи была создана программа для визуализации его работы.

# **Литература**

* + 1. Статья, описывающая принципы работы NNA [Электронный ресурс] - URL:  [https://ru.wikipedia.org/wiki/Алгоритм\_ближайшего\_соседа\_в\_задаче\_коммивояжёра /](https://habr.com/ru/articles/105639/)(дата обращения: 03.03.2024)
    2. Документация библиотеки PyQT [Электронный ресурс] - URL: <https://doc.qt.io/qtforpython-6/> (дата обращения: 25.09.2023)

1. **Листинг**
2. import networkx as nx  
     
     
   class ClosestNeighborAlgo:  
    def \_\_init\_\_(self, graph: nx.DiGraph):  
    self.graph = graph  
     
    def solve(self) -> nx.DiGraph:  
     
    result\_graph = nx.DiGraph()  
     
    nodes = list(self.graph.nodes())  
     
    start\_node = nodes[0]  
     
    result\_graph.add\_node(start\_node)  
     
    current\_node = start\_node  
     
    while len(result\_graph.nodes()) < len(nodes):  
     
    neighbors = list(self.graph.neighbors(current\_node))  
     
    closest\_neighbor = None  
    min\_distance = float('inf')  
    for neighbor in neighbors:  
    if neighbor not in result\_graph.nodes():  
    distance = self.graph[current\_node][neighbor]['weight']  
    if distance < min\_distance:  
    min\_distance = distance  
    closest\_neighbor = neighbor  
     
    result\_graph.add\_node(closest\_neighbor)  
    result\_graph.add\_edge(current\_node, closest\_neighbor, weight=min\_distance)  
     
    current\_node = closest\_neighbor  
     
    result\_graph.add\_edge(current\_node, start\_node, weight=self.graph[current\_node][start\_node]['weight'])  
     
    return result\_graph  
     
     
   import networkx as nx  
   import random  
     
     
   class ModifiedClosestNeighborAlgo:  
    def \_\_init\_\_(self, graph: nx.DiGraph):  
    self.graph = graph  
    self.length = 0  
     
    def solve(self) -> nx.DiGraph:  
    result\_graph = nx.DiGraph()  
    nodes = list(self.graph.nodes())  
    start\_node = self.get\_start\_node()  
    result\_graph.add\_node(start\_node)  
    current\_node = start\_node  
     
    while len(result\_graph.nodes()) < len(nodes):  
    neighbors = list(self.graph.neighbors(current\_node))  
    chosen\_neighbor = self.choose\_next\_node(neighbors, result\_graph)  
    result\_graph.add\_node(chosen\_neighbor)  
    result\_graph.add\_edge(current\_node, chosen\_neighbor,  
    weight=self.graph[current\_node][chosen\_neighbor]['weight'])  
    current\_node = chosen\_neighbor  
     
    result\_graph.add\_edge(current\_node, start\_node, weight=self.graph[current\_node][start\_node]['weight'])  
    self.total\_path\_length(result\_graph)  
    return result\_graph  
     
    def get\_start\_node(self):  
     
    max\_degree\_node = max(self.graph.degree(), key=lambda x: x[1])[0]  
    return max\_degree\_node  
     
    def choose\_next\_node(self, neighbors, result\_graph):  
     
    min\_degree = float('inf')  
    min\_degree\_nodes = []  
    for neighbor in neighbors:  
    if neighbor not in result\_graph.nodes():  
    degree = self.graph.degree(neighbor)  
    if degree < min\_degree:  
    min\_degree = degree  
    min\_degree\_nodes = [neighbor]  
    elif degree == min\_degree:  
    min\_degree\_nodes.append(neighbor)  
    return random.choice(min\_degree\_nodes)  
     
    def total\_path\_length(self, graph: nx.DiGraph) -> float:  
    total\_length = 0  
    for edge in graph.edges(data=True):  
    total\_length += edge[2]['weight']  
    self.length = total\_length