МИНОБРНАУКИ РОССИИ
Санкт-Петербургский государственный
электротехнический университет
«ЛЭТИ» им. В.И. Ульянова (Ленина)
Кафедра МО ЭВМ

Отчет
по курсовой работе
по дисциплине «Алгоритмы и структуры данных»
Тема: Нахождение оптимальной реализации алгоритма Дейкстры и его визуализация.

Студентка гр. 3382 _____ Копасова К. А.

Преподаватель _____ Иванов Д. В.

Санкт-Петербург
2024

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студентка Копасова К. А,

Группа 3382

Тема работы: Нахождение оптимальной реализации алгоритма Дейкстры и его визуализация.

Исходные данные:

Вариант 1:

Один из алгоритмов поиска кратчайших путей в графе - алгоритм Дейкстры. Основная его идея - улучшение пути через улучшающие вершины. Оно происходит при просмотре вершины и посещении смежных с нею не просмотренных вершин.

Вершина состоит из 3х полей: название, оценка длины пути до нее, предыдущая вершина.

При просмотре вершин они проверяются на то, были ли уже просмотрены до этого. А при рассмотрении новой вершины, она добавляется в структуру данных.

Для данной задачи скорость проверки и добавления вершины превалирует над затратами по памяти.

Решите, какая структура данных лучше подходит для решения задачи проверки вершины на то, была ли она уже просмотрена. Реализуйте выбранную структуру данных.

Содержание пояснительной записки:

«Содержание», «Задание», «Описание алгоритма», «Исследование», «Реализация алгоритма», «Визуализация алгоритма», «Тестирование», «Вывод», «Приложение».

Предполагаемый объем пояснительной записки:

Не менее 20 страниц.

Дата выдачи задания: 06.11.2024

Дата сдачи реферата: 08.12.2024

Дата защиты реферата: 12.12.2024

Студентка гр. 3382 _____ Копасова К. А.

Преподаватель _____ Иванов Д. В.

АННОТАЦИЯ

Программа реализует поиск кратчайшего пути в графе с использованием алгоритма Дейкстры. Граф представлен в виде списков смежности, где каждая вершина связана с соседними вершинами и весами рёбер. Пользователь может задавать граф через ввод данных в консоли или использовать заранее предопределённый пример.

Процесс работы алгоритма визуализируется в режиме реального времени: обрабатываемые рёбра выделяются, а кратчайший путь отображается отдельным цветом. Программа также выводит итоговый маршрут и общую длину пути.

Предусмотрены возможности интерактивного взаимодействия и проверки на примере заданного графа.

SUMMARY

The program implements Dijkstra's algorithm to find the shortest path in a graph. The graph is represented as adjacency lists, where each vertex is connected to its neighbors with weighted edges. The user can define the graph via console input or use a predefined example.

The algorithm's execution is visualized in real time: processed edges are highlighted, and the shortest path is displayed in a distinct color. The program also outputs the final route and the total path length.

Interactive input and verification using a predefined graph example are supported.

Содержание

Введение

Цель работы: изучить работу алгоритма Дейкстра, выявить оптимальную структуру данных для его реализации и визуализировать получившийся алгоритм.

Задачи:

1) Изучить алгоритм Дейкстры;
2) Проанализировать возможные системы данных для реализации алгоритма Дейкстры;
3) Реализовать алгоритм Дейкстры;
4) Визуализировать полученный алгоритм.

Основная часть

Описание алгоритма

Алгоритм Дейкстры - это алгоритм, который используется для нахождения кратчайших путей от одной стартовой вершины до всех остальных вершин в графе с неотрицательными весами рёбер. Работает для графов без рёбер отрицательного веса.

Алгоритм Дейкстры широко применяется;

1) Для расчёта маршрутов (навигационные системы);

2) В сетевых протоколах (OSPF, IS-IS).

Изначальная реализация алгоритма:

Обозначения:

a) V - множество вершин графа;

b) E - множество рёбер графа;

c) w[ij] - вес (длина) ребра ij;

d) a - вершина, расстояния от которой ищутся;

e) U - множество посещённых вершин;

f) d[u] - по окончании работы алгоритма равно длине кратчайшего пути из a до вершины u;

g) p[u] - по окончании работы алгоритма содержит кратчайший путь из a в u;

h) v - текущая вершина.

Описание:

В простейшей реализации для хранения чисел d[i] () можно использовать массив чисел, а для хранения принадлежности элемента множеству U - массив булевых переменных.

В начале алгоритма расстояние для начальной вершины полагается равным нулю, а все остальные расстояния заполняются большим положительным числом (большим максимального возможного пути в графе). Массив флагов заполняется нулями. Затем запускается основной цикл.

На каждом шаге цикла мы ищем вершину v с минимальным расстоянием и флагом равным нулю. Затем мы устанавливаем в ней флаг в 1 и проверяем все соседние с ней вершины u. Если в них (в u) расстояние больше, чем сумма расстояния до текущей вершины и длины ребра, то уменьшаем его. Цикл завершается, когда флаги всех вершин становятся равны 1, либо когда у всех вершин с флагом 0 d[i] = ∞. Последний случай возможен тогда и только тогда, когда граф G несвязный.

Проанализируем различные структуры данных для решения задачи про вершины на то, была ли она уже просмотрена.

Анализ структур данных

Сначала определим, какие структуры данных больше подходят для модели графов. Составим таблицу для анализа различных структур данных для вставки и поиска элемента:

Таблица 1 - Вставка элемента.

Структура данных	Лучший случай	Средний случай	Худший случай
Фибоначчиева куча	O(1)	O(1)	O(1)
Бинарная куча	O(1)	O(log n)	O(log n)
Бинарное дерево	O(log n)	O(log n)	O(n)
AVL-дерево	O(log n)		
Красно-черное дерево	O(log n)		
Массив	O(1)		
Односвязный список	O(1)		
Двусвязный список	O(1)		
Хэш-таблица	O(1)	O(1)	O(n) при коллизиях

Исходя из этой таблицы нужно просмотреть несколько структур данных: фибоначчиеву кучу, массив, односвязный и двусвязный списки и хэш-таблица, поскольку они являются самыми оптимальными из всех остальных для реализации графов.

Составим таблицу сложности для поиска элемента для перечисленных выше структур данных:

Таблица 2 - Поиск элемента.

Структура данных	Лучший случай	Средний случай	Худший случай
Фибоначчиева куча	O(n)		
Массив	O(1)	O(n)	O(n)
Односвязный список	O(1)	O(n)	O(n)
Двусвязный список	O(1)	O(n)	O(n)
Хэш-таблица	O(1)	O(1)	O(n) при коллизиях

Получается, что лучшая структура данных для вставки и поиска элемента - **хэш-таблица.** Удаление элемента не является главным действием для графа, поэтому его сложность не учитываем (но хэш-таблица так же является самой оптимальной при удалении элементов).

Реализация графов будет основана на **хэш-таблицах**.

Для алгоритма Дейкстры - поиска минимального пути в графе - используется **очередь** - не самая оптимальная структура данных для больших данных. Составим сравнительную таблицу различных структур данных для нахождения наилучшего варианта реализации алгоритма Дейкстры:

Таблица 3 - Сравнение структур данных для алгоритма Дейкстры.

Структура данных	Извлечение минимума	Вставка элемента	Обновление ключей	Сложность Дейкстры	Когда использовать
Массив	O(n)	O(1)	O(n)	$O(V^2 + E)$	Простой вариант для малых графов
Очередь (FIFO)	O(n)	O(1)	O(1)	$O(V^2 + E)$	Простой вариант для малых графов
Двоичная куча	O(log n)	O(log n)	O(log n)	O((V+E)*log V)	Универсальное решение
Фибоначиева куча	O(log n)	O(1)	O(1)	O(V*log V+E)	Для графов с большим числом ребер
D-куча	$O(\log_d n)$	$O(\log_d n)$	$O(\log_d n)$	$O((V+E)*\log_d V)$	При необходимости уменьшения глубины кучи за счет увеличения степени
Стек	O(n)	O(1)	O(1)	$O(V^2 + E)$	Для малых графах

Получаем, что самая оптимальные структуры данных - **фибоначчиева куча**. Изучим ее более подробно и убедимся в оптимальности.

Фибоначчиева куча теоретически сложна. Для реализации фибоначчиевой кучи требуется: динамическое управления несколькими деревьями, поддержание минимального дерева. *Это увеличивает вероятность ошибок. На практике она менее эффективна, чем в теории.*

Рассмотрим второй по оптимальности вариант - **двоичная куча**

Двоичная куча основана на простом массиве, что позволяет легко управлять ее структурой.

Несмотря на теоретическую сложность O(log n), *размер кучи минимален.*

Получаем, что, несмотря на теоретический выигрыш фибоначчиевой кучи, на практике более оптимальной оказывается двоичная куча, поэтому выберем ее для реализации алгоритма Дейкстры.

Реализация двоичной кучи

Реализуем двоичную кучу с помощью массива. Создадим для этого класс BinaryHeap.

Опишу методы данного класса:

1) Конструктор: инициализируем пустую кучу. Зададим объект self.heap - это список, представляющий двоичную кучу, где каждый элемент представлен как кортеж (priority, item), где priority - приоритет элемента, item - сам элемент. Использование кортежа позволяет сохранить пару, где первый элемент определяет порядок в куче.

2) Метод push(self, priority, item): добавление нового элемента с приоритетом в кучу. Метод добавляет элемент в конец списка heap , а после с помощью метода move_up поднимает элемент вверх для сохранения свойства кучи, если priority у нового элемента меньше, чем у предыдущего.

3) Метод pop(self): удаляет и возвращает элемент с минимальным приоритетом - корень кучи. В данном методе рассматривается две ситуации:

a) Если в куче один элемент (корень кучи), то он удаляется и возвращается.

b) Иначе корень (минимальный элемент) удаляется, а последний элемент массива перемещается в корень и снова с помощью move_down перемещается вниз для сохранения свойства кучи.

В итоге метод возвращает удалённый минимальный элемент.

4) Метод move_up(self, index): перемещает элемент вверх для сохранения свойств кучи. Метод находит индекс текущего родительского узла с помощью формулы (index - 1)//2 (index - индекс элемента, который нужно переместить выше в куче для сохранения ее свойств), сравнивает приоритет текущего элемента с приоритетом родительского элемента. Если приоритет текущего элемента меньше, то меняет их местами, обновляет index = (index - 1)//2 и находит следующий родительский узел с помощью той же формулы. Данный процесс продолжается, пока свойства кучи не будут восстановлены.

5) Метод move_down(self, index): перемещает элемент вниз для сохранения свойств кучи. Метод находит индексы дочерних узлов с помощью формул: левый ребенок = 2*index + 1, правый ребенок = 2*index + 2. После определяет меньшего из детей и если приоритет текущего узла больше, чем у меньшего ребенка, меняет их местами. Далее обновляет индекс и продолжает процесс для нового индекса, пока не будет восстановлено свойство кучи.

6) Метод is_empty(self): проверяет, пуста ли куча. Если куча пуста (т. е. в ней нет элементов), то метод возвращает True, иначе False.

Реализация двоичной кучи поможет корректно создать алгоритм Дейкстры для поиска наименьшего пути у двух вершин.

Реализация алгоритма Дейкстры

Подключим библиотеки для двоичной кучи и последующей визуализации графа.

```
from bin_heap import BinaryHeap
import matplotlib.pyplot as plt
import network as nx
```

Далее создадим класс Graph, который будет отвечать за работу графов и самого алгоритма. Опишем все методы данного класса:

1) Конструктор: создает два основных поля. Self.edges - словарь, отвечающий за хранение вершин графа в формате {вершина(ключ): [(сосед, вес), …](значение)} (массив может пополняться кортежами в зависимости от того, с какими другими вершинами и в каком количестве соединена изначальная вершина). Self.graph_nx - граф, созданный библиотекой network для визуализации. Self.pos используется для размещения вершин при рисовании графа.

2) Метод add_edge(self, start, end, weight): добавляет новое ребро в граф. Принимает на вход начальную вершину, конечную вершину и вес ребра между ними. Метод сначала удаляет лишние пробелы из имен вершин с помощью strip(), чтобы избежать проблем с некорректными называниями вершин, затем проверяет, существуют ли вершины start и end в графе. Если вершин не существует, то они добавляются с пустым списком «соседей» в словарь self.edges. Далее добавляет ребро между вершинами start и end с указанным весом weight с помощью метода списка append(). И добавляет вершины и ребро в объект self.graph_nx для корректной визуализации графа.

3) Метод draw_graph(self, path=None, highlight_edges=None, title=”Граф”): визуализирует полученный граф. Принимает три опциональных параметра: path - список вершин, которые составляют кратчайший путь (если он указан, то ребра между вершинами в последствии будут выделены зеленым цветом), highlight_edges - список ребер, которые нужно временно выделить (ребра, которые программа проверяет выделятся красным цветом) и title - название графа. Метод очищает текущую визуализацию plt.clf(), размещает вершины на плоскости с помощью кругового алгоритма

(nx.circular_layout), а затем рисует вершины и ребра графа. После рисует посчитанные значения суммы путей над вершинами, если они заданы (изначально над вершиной написано inf - бесконечность). Если заданы параметры пути или выделения ребер, то он добавляет из поверх основного графа. Метод так же масштабирует изображение для корректного отображения (plt.axis(“equal”) и дает паузу для наглядности (plt.pause(1)).

4) Метод algorithm_Dijkstra(self, start, end): ищет кратчайщий путь от вершины start до вершины end с использованием двоичной кучи. Сначала проверяет существование начальной и конечной вершин start и end. Затем создает словарь distances для хранения минимальных расстояний от начальной вершины до каждой другой (изначально inf). Далее словарь previous использует для отслеживания предыдущих вершин в кратчайшем пути. Для управления приоритетами используется двоичная куча (BinaryHeap). Главный цикл извлекает вершины с минимальным накопленным расстоянием (корень), обновляет расстояния до соседей и добавляет их в очередь, если найден более короткий путь. Кадый шаг визуализируется, выделяя текущее обрабатываемое ребро красным цветом и отображая обновленные расстояния. После завершения алгоритма восстанавливается кратчайший путь с помощью словаря previous. Если путь недостижим, то возвращается пустой список и бесконечное расстояние. Найденный путь визуализируется и возвращается вместе с его длиной.

5) Функция main предназначена для ввода графа и вершин пользователем. Сначала пользователь вводит вершины в формате “вершина-начало вес ребра между вершинами вершина-конец”, после чего строится и визуализируется граф. Затем вводятся начальная и конечная вершины для поиска пути. Результат вычислений выводится в консоль: сначала самый выгодный путь из вершин, потом его сумма.

6) Функция check_finish_data создает граф для тестирования без ввода в консоль.

Данные методы и функции позволяют корректно и оптимально работать алгоритму Дейкстры, в чем и была главная задача. Полный код программы можно найти в Приложении 1. В Приложении 2 показана визуализация программы.

Тестирование

Протестируем алгоритм на корректность выполнения:

Таблица 4 - Результаты тестирования основного кода программы:

№п/п	Входные данные	Выходные данные	Комментарии
1	a 1 b b 1 c a 1 d d 1 c c 15 h h 30 n h 29 m n 1 o m 5 o a h	Кратчайший путь: a -> b -> c -> h -> n -> o Общая длина пути: 48	OK
2	a 1 b b 1 c a 1 c a c	Кратчайший путь: a -> c Общая длина пути: 1	OK
3	a 1 b b 3 c a 5 c d 1 c c 15 h a h	Кратчайший путь: a -> b -> c -> h Общая длина пути: 19	OK

Программа работает корректно. Проанализируем ее производительность далее.

Анализ производительности программы

Создам специальный файл test.py, где проанализирую, насколько быстро выполняется поиск кратчайшего пути в графах различных размеров (вершины и ребра).

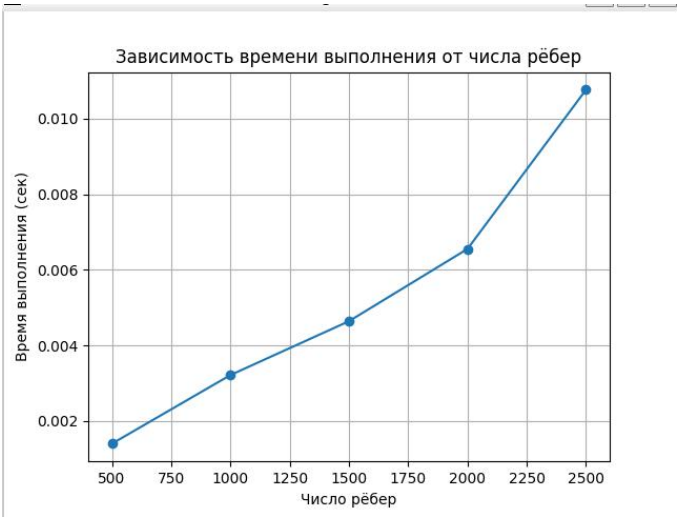
1. Класс Graph переписан из main.py без визуализации, чтобы не усложнять тестовое время на отрисовку.

2. Функция find_furthest_vertices использует BFS (алгоритм обхода графа в ширину) для нахождения удаленной вершины для поданной любой вершины (чтобы вершины не были слишком близки - тогда оценка будет не совсем объективной). Возвращает данная функция вершину, которая наиболее удалена от вершины старта.

3. Функция measure_perfomance создает граф с заданным количеством вершин и ребер, добавляя случайные ребра с весами от 1 до 10. После выбирает случайную начальную величину и находит наиболее удаленную от нее вершину с помощью find_furthest_vertices, а далее запускает алгоритм Дейкстры для поиска кратчайшего пути и замеряет время его выполнения.

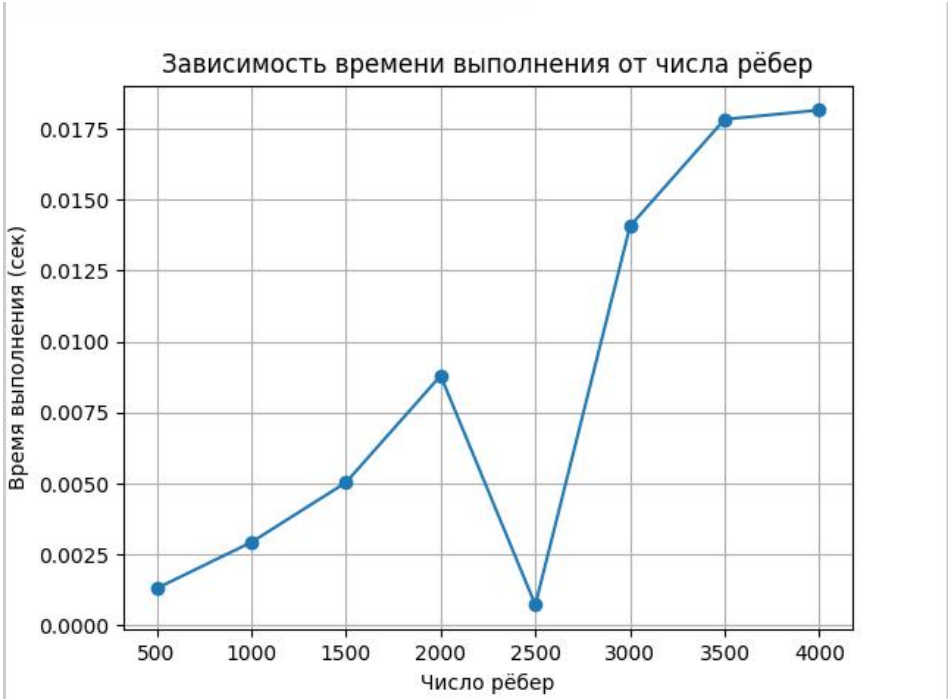
Производительность алгоритма тестируется на графах с разными размерами (100-500 вершин и 500-2500 ребер), а после подсчета выполнения алгоритма строится график зависимости времени выполнения алгоритма Дейкстры от числа ребер в графе.

Получаемые графики:



На данном графике можно увидеть, что при увеличении графа (числа вершин и рёбер) идёт увеличение времени выполнения программы. Но даже при количестве 2500 вершин и рёбер - программа работает быстро.

Повысим число вершин и рёбер до 4000:



Даже при таких данных программа работает быстро. Но есть нюанс: при выборе любой начальной вершины у каждого графа есть вероятность, что в целом его путь будет меньше, чем у прошлого графа (оценка не совсем объективна), но проблема в том, что если программа будет выбирать первую и последнюю вершину для полного прохождения графа, то не факт, что начальная и конечная вершина связанны - именно поэтому этот способ проходки графа не подойдёт для вычисления оптимальности работы алгоритма. Но можно вычислить сложность для выполнения алгоритма при случае, если кратчайший путь ищется между начальной вершиной графа и самой последней добавленной в граф - это будет являться теоретическим результатом. Поэтому, если полученные тестом результаты удовлетворят теоретической сложности алгоритма, то алгоритм оптимален.

Рассчитаем сложность O((V + E) * logV) для заданных вершин и графов V = E = [500, 1000, 1500, 2000, 2500, 3000, 3500, 4000]

Таблица 5 - Теоретическая оценка алгоритма Дейкстры с двоичной кучей:

V = E	Сложность: O((V + E) * logV), мс; с
500	8965.75; 8.97
1000	19931.57; 19.93
1500	31652.24; 31.65
2000	43863.14; 43.86
2500	56438.56; 56.44
3000	69304.48; 69.30
3500	82411.97; 82.41
4000	95726.27; 95.73

В результате получаем, что все тестирующие значения намного меньше, чем теоретические, поэтому алгоритм выполняется оптимально.

Вывод

В результате курсовой работы удалось:

1) Изучить алгоритм Дейкстры;

2) Проанализировать возможные системы данных для реализации алгоритма Дейкстры - на данном этапе выбраны хэш-таблица и двочиная куча;

3) Реализовать алгоритм Дейкстры;

4) Визуализировать полученный алгоритм;

5) Провести анализ полученного алгоритма.

Была создана программа, которая находит кратчайший путь в графе между двумя вершинами с подробной визуализацией.

Приложение 1

Файл bin_heap.py:

```
class BinaryHeap:
    def __init__(self):
        self.heap = []

    def push(self, priority, item):
        """Добавление элемента с приоритетом в кучу"""
        self.heap.append((priority, item))
        self.move_up(len(self.heap) - 1)

    def pop(self):
        """Удаление и возвращение элемента с минимальным приоритетом"""
        # если куча содержит один элемент - корень - просто возвращаем его
        if len(self.heap) == 1:
            return self.heap.pop()
        root = self.heap[0]
        self.heap[0] = self.heap.pop()  # перемещаем последний элемент в
корень
        self.move_down(0)
        return root

    def move_up(self, index):
        """Перемещает элемент вверх для сохранения свойств кучи"""
        parent = (index - 1) // 2
        while index > 0 and self.heap[index][0] < self.heap[parent][0]:
            self.heap[index], self.heap[parent] = self.heap[parent],
self.heap[index]
            index = parent
            parent = (index - 1) // 2

    def move_down(self, index):
        """Перемещает элемент вниз для сохранения свойств кучи"""
        child = 2 * index + 1
        while child < len(self.heap):
            # выбираем меньшего ребёнка
            if child + 1 < len(self.heap) and self.heap[child + 1][0] <
self.heap[child][0]:
                child += 1
            # если текущий узел меньше ребёнка, завершить
            if self.heap[index][0] <= self.heap[child][0]:
                break
            # иначе меняем местами
            self.heap[index], self.heap[child] = self.heap[child],
self.heap[index]
            index = child
            child = 2 * index + 1

    def is_empty(self):
        """Проверка кучи на пустоту"""
        return not self.heap
```

Файл main.py:

```
from bin_heap import BinaryHeap
import matplotlib.pyplot as plt
import networkx as nx

class Graph:
    def __init__(self):
```

```
        self.edges = {}  # {вершина: [(сосед, вес), ...]} - формат графа
        self.graph_nx = nx.Graph()  # для визуализации

    def add_edge(self, start, end, weight):
        """Добавление ребра в граф"""
        start, end = start.strip(), end.strip()
        if start not in self.edges:
            self.edges[start] = []
        if end not in self.edges:
            self.edges[end] = []
        self.edges[start].append((end, weight))
        self.edges[end].append((start, weight))
        self.graph_nx.add_edge(start, end, weight=weight)
        self.pos = nx.spring_layout(self.graph_nx)

    def draw_graph(self, path=None, highlight_edges=None, distances=None,
title="Граф"):
        """Визуализация графа"""
        plt.clf()
        plt.title(title)

        # используем кругового размещения для отображения вершин
        self.pos = nx.circular_layout(self.graph_nx)
        # отображаем вершины и ребра
        nx.draw(self.graph_nx,          self.pos,          with_labels=True,
node_color="lightblue", node_size=500, font_size=10, font_weight="bold")
        nx.draw_networkx_edge_labels(self.graph_nx, self.pos, edge_labels={(u,
v): d['weight'] for u, v, d in self.graph_nx.edges(data=True)})

        # смещение координат для отображения меток выше вершин
        offset_pos  =  {node:  (x,  y  +  0.15)  for  node,  (x,  y)  in
self.pos.items()}
        # рисуем значения над вершинами (расстояния)
        if distances:
            nx.draw_networkx_labels(self.graph_nx,                offset_pos,
labels=distances, font_size=12, font_color="red", font_weight="bold")

        # рисуем короткий путь зеленым
        if path:
            path_edges = list(zip(path[:-1], path[1:]))
            nx.draw_networkx_edges(self.graph_nx,                    self.pos,
edgelist=path_edges, edge_color="green", width=2)

        # рисуем проходку по путям красным
        if highlight_edges:
            nx.draw_networkx_edges(
                self.graph_nx,
                self.pos,
                edgelist=highlight_edges,
                edge_color="red",
                width=2.5,
                label="Обрабатываемое ребро",
            )

        # масштабирование для корректного вывода изображения
        plt.axis("equal")
        plt.pause(1)

    def algorithm_Dijkstra(self, start, end):
        """Поиск кратчайшего пути от вершины start до вершины end с
использованием двоичной кучи"""
```

```
        if start not in self.edges or end not in self.edges:
            raise ValueError(f"Одна или обе вершины ({start}, {end}) отсутствуют в графе")

        # расстояния от начальной вершины до всех других
        distances = {vertex: float('inf') for vertex in self.edges}
        distances[start] = 0

        # предыдущие вершины для восстановления пути
        previous = {vertex: None for vertex in self.edges}

        # инициализация бинарной кучи
        priority_heap = BinaryHeap()
        priority_heap.push(0, start)

        while not priority_heap.is_empty():
            # извлекаем вершину с минимальным накопленным весом
            current_distance, current_vertex = priority_heap.pop()

            # если извлечённое расстояние больше, чем текущее, пропускаем (устаревший элемент)
            if current_distance > distances[current_vertex]:
                continue

            # если достигли конечной вершины, завершаем
            if current_vertex == end:
                break

            # обрабатываем всех соседей
            for neighbor, weight in self.edges[current_vertex]:
                distance = current_distance + weight

                # если нашли более короткий путь к соседу
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    previous[neighbor] = current_vertex
                    priority_heap.push(distance, neighbor)

                    # визуализируем выделение рёбер
                    self.draw_graph(
                        highlight_edges=[(current_vertex, neighbor)],
                        title=f"Обработка ребра {current_vertex} -> {neighbor}",
                        distances=distances  # передаем словарь с расстояниями
                    )

        # восстановление пути
        path = []
        current = end
        while current is not None:
            path.append(current)
            current = previous[current]

        # если путь невозможен, возвращаем пустой список и бесконечную длину
        if distances[end] == float('inf'):
            self.draw_graph(path=[], title="Путь недостижим")
            return [], float('inf')

        # визуализация пути
        path = path[::-1]  # переворачиваем путь
```

```
        path_edges = [(path[i], path[i+1]) for i in range(len(path)-1)]
        self.draw_graph(path=path,                highlight_edges=path_edges,
title="Кратчайший путь")

        self.draw_graph(path=path, title="Кратчайший путь")
        # возвращаем путь
        return path, distances[end]

def main():
    """Функция для проверки работы программы с вводом вершин"""
    graph = Graph()

    print("Введите вершины и вес ребра в формате 'начало вес конец' (Enter
для завершения):")
    while True:
        data = input().strip()
        if data == '':
            break
        try:
            start, weight, end = data.split()
            weight = int(weight)
            graph.add_edge(start, end, weight)
        except ValueError:
            print("Ошибка ввода! Формат: 'начало вес конец'")

    # вводим вершины графа
    while True:
        try:
            info = input("Введите начальную и конечную вершины для поиска
кратчейшего пути в формате 'начало конец': ").strip()
            start, end = info.split()
            plt.ion()  # включаем интерактивный режим для matplotlib
            path, total_distance = graph.algorithm_Dijkstra(start, end)
            print("\nКратчайший путь:", " -> ".join(path))
            print("Общая длина пути:", total_distance)

            # поиск кратчайшего пути
            plt.ioff()  # выключаем интерактивный режим
            plt.show()  # оставляем финальный граф для просмотра

            break
        except ValueError:
            print("Ошибка ввода! Формат: 'начало конец'")

def check_finish_data():
    """Функция работы программы без ввода вершин"""
      g = Graph()
    g.add_edge('a', 'b', 1)
    g.add_edge('b', 'c', 3)
    g.add_edge('a', 'c', 5)
    g.add_edge('d', 'c', 1)
    g.add_edge('c', 'h', 15)
    g.add_edge('h', 'n', 30)
    g.add_edge('h', 'm', 29)
    g.add_edge('n', 'o', 1)
    g.add_edge('m', 'o', 5)

    start = 'a'
    end = 'h'
    plt.ion()
```

```
    path, total_distance = g.algorithm_Dijkstra(start, end)

    print("\nКратчайший путь:", " -> ".join(path))
    print("Общая длина пути:", total_distance)


    plt.ioff()
    plt.show()
    pass

# check_finish_data()
main()
```

Файл test.py:

```
import time
import random
import matplotlib.pyplot as plt
from collections import deque
from bin_heap import BinaryHeap

# определяю написанный класс Graph, но без визуализации для быстрого выполнения тестов
class Graph:
    def __init__(self):
        self.edges = {}

    def add_edge(self, start, end, weight):
        """Добавляет ребро в граф."""
        start, end = start.strip(), end.strip()
        if start not in self.edges:
            self.edges[start] = []
        if end not in self.edges:
            self.edges[end] = []
        self.edges[start].append((end, weight))
        self.edges[end].append((start, weight))

    def algorithm_Dijkstra(self, start, end):
        """Поиск кратчайшего пути от вершины start до вершины end с использованием бинарной кучи"""
        if start not in self.edges or end not in self.edges:
            raise ValueError(f"Одна или обе вершины ({start}, {end}) отсутствуют в графе")

        # расстояния от начальной вершины до всех других
        distances = {vertex: float('inf') for vertex in self.edges}
        distances[start] = 0

        # предыдущие вершины для восстановления пути
        previous = {vertex: None for vertex in self.edges}

        # инициализация бинарной кучи
        priority_queue = BinaryHeap()
        priority_queue.push(0, start)

        while not priority_queue.is_empty():
            # извлекаем вершину с минимальным накопленным весом
            current_distance, current_vertex = priority_queue.pop()

            # если извлечённое расстояние больше, чем текущее, пропускаем (устаревший элемент)
            if current_distance > distances[current_vertex]:
```

```
                continue

            # если достигли конечной вершины, завершаем
            if current_vertex == end:
                break

            # обрабатываем всех соседей
            for neighbor, weight in self.edges[current_vertex]:
                distance = current_distance + weight

                # если нашли более короткий путь к соседу
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    previous[neighbor] = current_vertex
                    priority_queue.push(distance, neighbor)

        # восстановление пути
        path = []
        current = end
        while current is not None:
            path.append(current)
            current = previous[current]

        # если путь невозможен, возвращаем пустой список и бесконечную длину
        if distances[end] == float('inf'):
            return [], float('inf')

        # визуализация пути
        path = path[::-1]  # переворачиваем путь
        # возвращаем путь
        return path, distances[end]

def find_furthest_vertices(graph, start_vertex):
    """Находит вершину, наиболее удалённую от start_vertex, с помощью BFS"""
    # проверяем, есть ли начальная вершина в графе
    if start_vertex not in graph.edges:
        raise ValueError(f"Вершина {start_vertex} отсутствует в графе.")

    # используем очередь для BFS
    queue = deque([(start_vertex, 0)])  # храним вершину и расстояние до неё
    visited = set()  # множество посещённых вершин
    furthest_vertex = start_vertex
    max_distance = 0

    while queue:
        current, distance = queue.popleft()

        # если вершина уже посещена, пропускаем её
        if current in visited:
            continue

        # отмечаем вершину как посещённую
        visited.add(current)

        # обновляем наиболее удалённую вершину
        if distance > max_distance:
            max_distance = distance
            furthest_vertex = current

        # добавляем соседей в очередь
        for neighbor, _ in graph.edges[current]:
```

```
            if neighbor not in visited:
                queue.append((neighbor, distance + 1))

    return furthest_vertex

def measure_performance(num_vertices, num_edges):
    graph = Graph()

    # генерация случайных рёбер
    vertices = [str(i) for i in range(num_vertices)]

    for _ in range(num_edges):
        u, v = random.sample(vertices, 2)
        weight = random.randint(1, 10)
        graph.add_edge(u, v, weight)

    # выбираем случайную начальную вершину
    start_vertex = random.choice(vertices)
    # находим самую удалённую вершину от начальной
    end_vertex = find_furthest_vertices(graph, start_vertex)

    print(f"Удалённые вершины: {start_vertex} -> {end_vertex}")

    # измерение времени выполнения алгоритма Дейкстры
    start_time = time.time()
    path, total_distance = graph.algorithm_Dijkstra(start_vertex, end_vertex)
    end_time = time.time()

    elapsed_time = end_time - start_time

    print(f"Время выполнения для {num_vertices} вершин и {num_edges} рёбер:
{elapsed_time:.6f} секунд")
    print(f"Кратчайший путь: {' -> '.join(path)}")
    print(f"Общая длина пути: {total_distance}")
    return elapsed_time

# пример использования: измерение времени для графов разного размера
sizes = [(500, 500), (1000, 1000), (1500, 1500), (2000, 2000), (2500, 2500),
(3000, 3000), (3500, 3500), (4000, 4000), ()]
times = []

for num_vertices, num_edges in sizes:
    elapsed_time = measure_performance(num_vertices, num_edges)
    times.append(elapsed_time)

# построение графика зависимости времени от числа рёбер
plt.plot([x[1] for x in sizes], times, marker='o')
plt.xlabel('Число рёбер')
plt.ylabel('Время выполнения (сек)')
plt.title('Зависимость времени выполнения от числа рёбер')
plt.grid(True)
plt.show()
```

Приложение 2

Визуализация программы для теста №1 из таблицы 4:

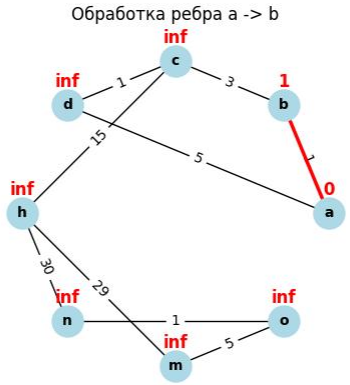


Рис. 1. Обработка ребра a -> b.

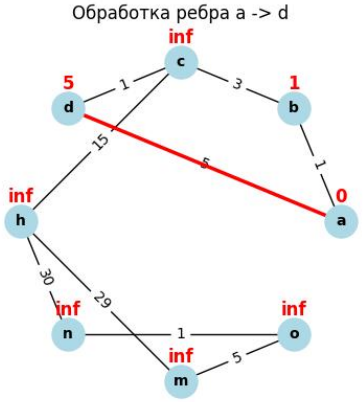


Рис. 2. Обработка ребра a -> d.

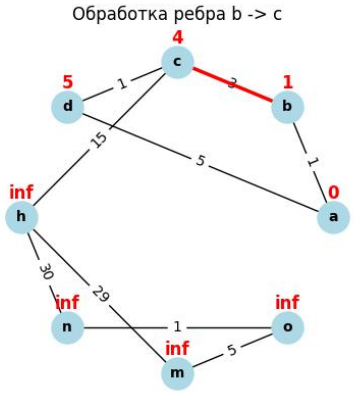


Рис. 3. Обработка ребра b -> c.

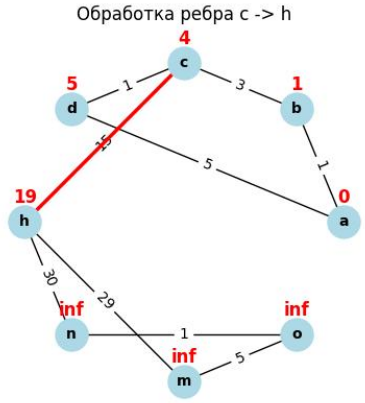


Рис. 4. Обработка ребра c -> h.

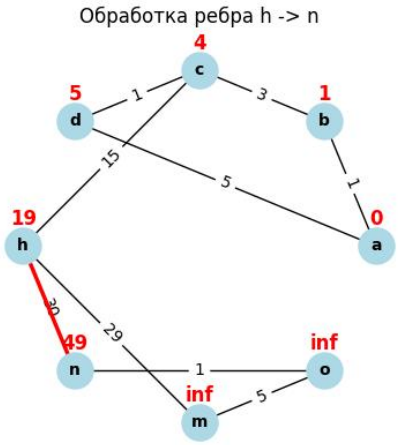


Рис. 5. Обработка ребра h -> n.

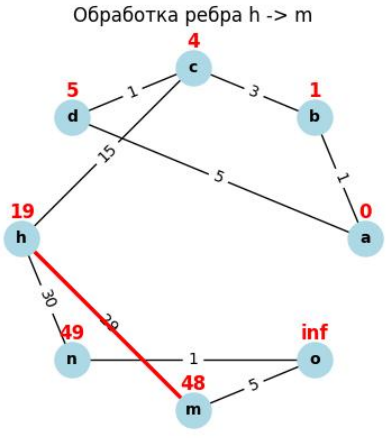


Рис. 6. Обработка ребра h -> m.

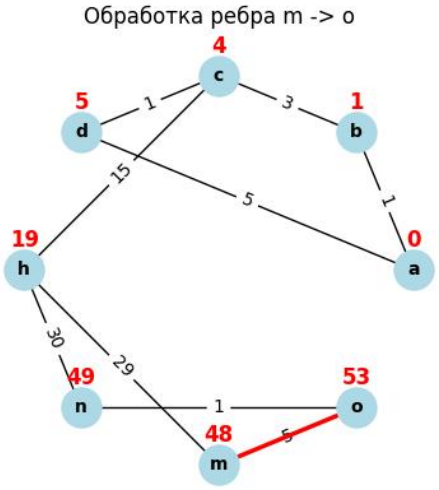


Рис. 7. Обработка ребра m -> o.

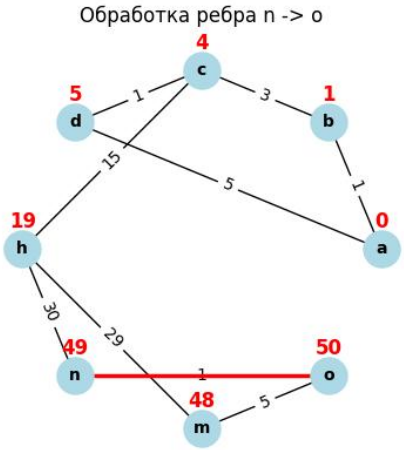


Рис. 8. Обработка ребра n -> o.

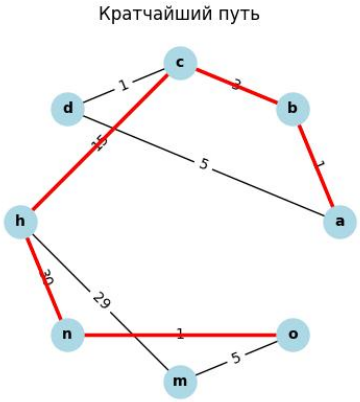


Рис. 9. Кратчайший путь от a до o.

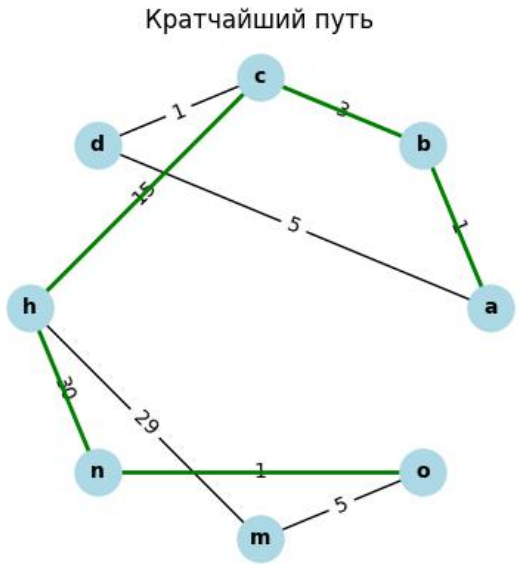


Рис. 10. Итоговый кратчайший путь от a до o.