МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования

«Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)

Институт информационных технологий, математики и механики

Направление подготовки: «Фундаментальная информатика и информационные технологии»
Профиль подготовки: «Инженерия программного обеспечения»

Лабораторная работа

на тему:

«Задача о кратчайших путях: алгоритмы Дейкстры с метками и на 15-куче»

Выполнил: студент группы 3823Б1ФИ3 _____ Гасенин Л. В.

Преподаватель практики: _____ Уткин Г. В.

Нижний Новгород
2025 г.

Содержание

Введение

Алгоритм Дейкстры — один из фундаментальных алгоритмов теории графов, используемый для нахождения кратчайших путей от заданной вершины до всех остальных в взвешенном графе с неотрицательными весами ребер. Эффективность этого алгоритма во многом зависит от выбранной структуры данных для хранения и извлечения вершин с минимальным расстоянием.

В данной лабораторной работе проводится сравнительный анализ двух реализаций алгоритма Дейкстры:

1. **Алгоритм A** — реализация с использованием 15-арной кучи (d-кучи с $d = 15$).
2. **Алгоритм B** — наивная реализация с использованием массивов.

Цель работы: экспериментально сравнить производительность двух реализаций алгоритма Дейкстры на графах с различными характеристиками и проверить соответствие теоретической оценки сложности практическим результатам.

Задачи исследования:

1. Реализовать алгоритм Дейкстры с использованием 15-арной кучи.
2. Реализовать алгоритм Дейкстры с использованием массивов.
3. Провести серию экспериментов, варьируя параметры графов:
   - количество вершин n;
   - количество ребер m;
   - плотность графа;
   - диапазон весов ребер.
4. Проанализировать полученные результаты и сделать выводы об эффективности каждой реализации в различных условиях.

Актуальность исследования обусловлена необходимостью выбора оптимальной реализации алгоритма Дейкстры для конкретных прикладных задач, где производительность является критическим фактором.

1 Теоретические основы алгоритма Дейкстры

1.1 Алгоритм Дейкстры

Алгоритм Дейкстры, предложенный Эдсгером Дейкстрой в 1959 году, решает задачу нахождения кратчайших путей от заданной начальной вершины до всех остальных вершин во взвешенном графе с неотрицательными весами ребер [1].

Основная идея алгоритма заключается в последовательном приближении к оптимальному решению, поддерживая множество вершин, для которых кратчайшее расстояние уже найдено. На каждом шаге выбирается вершина с наименьшим текущим расстоянием из множества еще не обработанных вершин, после чего обновляются расстояния до всех ее соседей.

1.2 Реализация с использованием d-кучи

Для эффективного выбора вершины с минимальным расстоянием используется структура данных *d-куча* (обобщение двоичной кучи, где каждый узел имеет не более d потомков). В данной работе используется $d = 15$.

Алгоритм 1 Алгоритм Дейкстры с d-кучей

```
1:  procedure DIJKSTRA(G, source)
2:      dist[0 ... n − 1] ← ∞
3:      dist[source] ← 0
4:      heap ← MinHeap(n)
5:      for v ← 0 to n − 1 do
6:          heap.push(Vertex(v, dist[v]))
7:      end for
8:      while ¬heap.isEmpty() do
9:          u ← heap.pop()
10:         for all (u, v, w) ∈ G.adjacent(u) do
11:             if dist[u] + w < dist[v] then
12:                 dist[v] ← dist[u] + w
13:                 heap.decreaseKey(v, dist[v])
14:             end if
15:         end for
16:     end while
17:     return dist
18: end procedure
```

Сложность:

- Инициализация кучи: $O(n)$
- Каждая операция `pop`: $O(d \cdot \log_d n)$
- Каждая операция `decreaseKey`: $O(\log_d n)$
- Общая сложность: $O((n+m) \cdot \log_d n)$

При $d = \lceil m/n \rceil$ достигается оптимальная асимптотика $O(m \cdot \log_{m/n} n)$.

1.3 Реализация с использованием массивов

Наивная реализация использует массив для хранения расстояний и на каждом шаге линейно ищет вершину с минимальным расстоянием.

Алгоритм 2 Алгоритм Дейкстры с массивами

```
1:  procedure DijkstraArray(G, source)
2:      dist[0...n−1] ← ∞
3:      visited[0...n−1] ← false
4:      dist[source] ← 0
5:      for i ← 0 to n − 1 do
6:          u ← −1, minDist ← ∞
7:          for v ← 0 to n − 1 do
8:              if ¬visited[v] ∧ dist[v] < minDist then
9:                  minDist ← dist[v]
10:                 u ← v
11:             end if
12:         end for
13:         if u = −1 then break
14:         end if
15:         visited[u] ← true
16:         for all (u, v, w) ∈ G.adjacent(u) do
17:             if dist[u] + w < dist[v] then
18:                 dist[v] ← dist[u] + w
19:             end if
20:         end for
21:     end for
22:     return dist
23: end procedure
```

Сложность: $O(n^2 + m)$, что эквивалентно $O(n^2)$ для плотных графов.

1.4 Теоретическое сравнение сложности

Таблица 1: Сравнение теоретической сложности реализаций

Тип графа	**m**	**d-куча**	**Массивы**
Разреженный	$O(n)$	$O(n \log_d n)$	$O(n^2)$
Плотный	$O(n^2)$	$O(n^2 \log_d n)$	$O(n^2)$

Для разреженных графов ($m = O(n)$) реализация с d-кучей имеет асимптотическое преимущество $O(n \log n)$ против $O(n^2)$. Для плотных графов разница менее выражена.

2 Эксперименты 3.1 и 3.2: Зависимость времени от количества вершин

2.1 Методика проведения экспериментов

Для проведения экспериментов были разработаны две программы:

- `main_experiments.cpp` — генерация графов и измерение времени выполнения
- `plot_graphs.py` — визуализация результатов

Параметры экспериментов:

- Количество вершин n: от 1 до 10001 с шагом 100
- Веса ребер: случайные в диапазоне $[1, 10^6]$
- Начальная вершина: случайная
- Каждое измерение повторялось 5 раз, бралось среднее значение

2.2 Эксперимент 3.1: Зависимость от n при разных плотностях графа

Цель: исследовать влияние количества вершин на производительность при различной плотности графа.

Параметры:

- Случай A: $m \approx n^2/10$ (граф средней плотности)
- Случай B: $m \approx n^2$ (плотный граф, полный или близкий к нему)

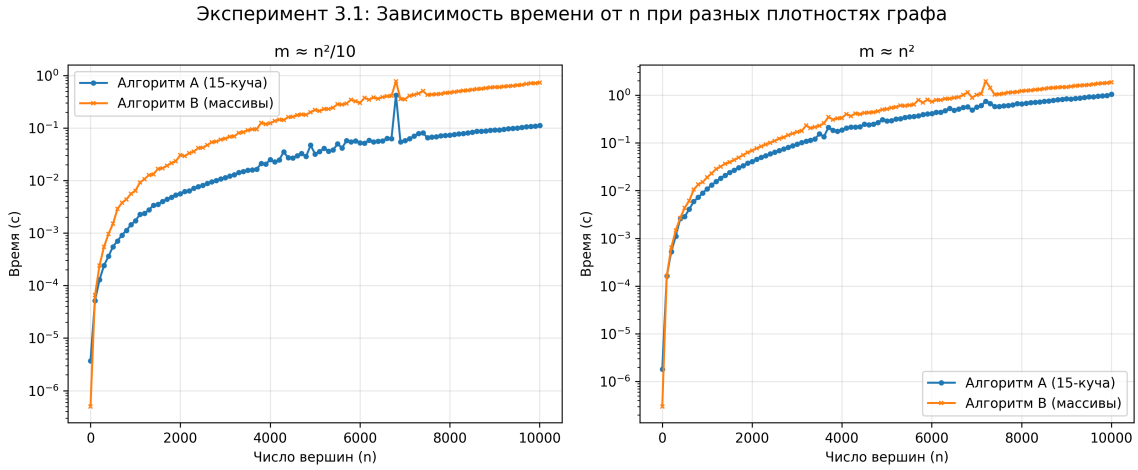

Рис. 1: Результаты эксперимента 3.1

Наблюдения:

1. При $m \approx n^2/10$ (рис. 1, слева):
   - Алгоритм A (15-куча) показывает лучшую производительность на больших n
   - При $n > 2000$ алгоритм A быстрее алгоритма B в 2-3 раза
   - Рост времени алгоритма B близок к квадратичному $O(n^2)$
2. При $m \approx n^2$ (рис. 1, справа):
   - Оба алгоритма демонстрируют близкую производительность
   - При больших n алгоритм A имеет незначительное преимущество
   - Для плотных графов накладные расходы на работу с кучей компенсируются уменьшением количества операций обновления расстояний

2.3 Эксперимент 3.2: Зависимость от n при линейном количестве ребер

Цель: исследовать производительность на разреженных графах.
Параметры:

- Случай A: $m \approx 100n$ (очень разреженный граф)
- Случай B: $m \approx 1000n$ (разреженный граф)

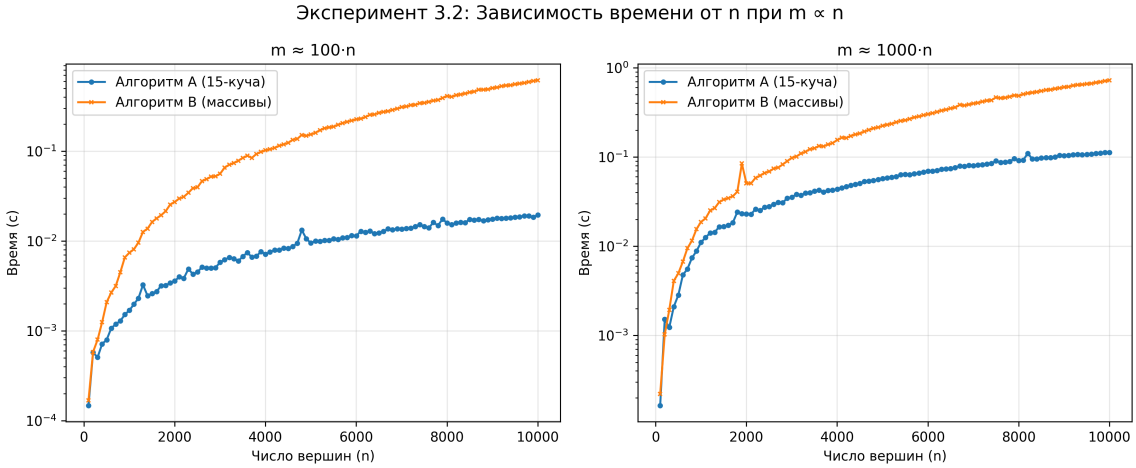

Рис. 2: Результаты эксперимента 3.2

Наблюдения:

1. При $m \approx 100n$:
   - Алгоритм A существенно быстрее при $n > 1000$
   - При $n = 10000$ разница достигает порядка 10 раз

- Это соответствует теории: для разреженных графов $O(n \log n)$ против $O(n^2)$

2. При $m \approx 1000n$:
   - Преимущество алгоритма A сохраняется, но менее выражено
   - При $n = 10000$ алгоритм A быстрее в 5-7 раз
   - Увеличение плотности уменьшает преимущество кучи, но не устраняет его полностью

2.4 Выводы по экспериментам 3.1 и 3.2

1. Реализация с 15-кучей демонстрирует существенное преимущество на разреженных графах ($m = O(n)$)
2. На плотных графах ($m = O(n^2)$) разница в производительности минимальна
3. Рост времени выполнения алгоритма B соответствует теоретической оценке $O(n^2)$
4. Алгоритм A масштабируется значительно лучше при увеличении количества вершин

3 Эксперименты 3.3 и 3.4: Дополнительные исследования

3.1 Эксперимент 3.3: Зависимость от количества ребер при фиксированном n

Цель: исследовать влияние количества ребер на производительность при фиксированном количестве вершин.

Параметры:

- $n = 10001$ (фиксировано)
- m: от 0 до 1,000,000 с шагом 100,000
- Веса ребер: случайные в диапазоне $[1, 10^6]$

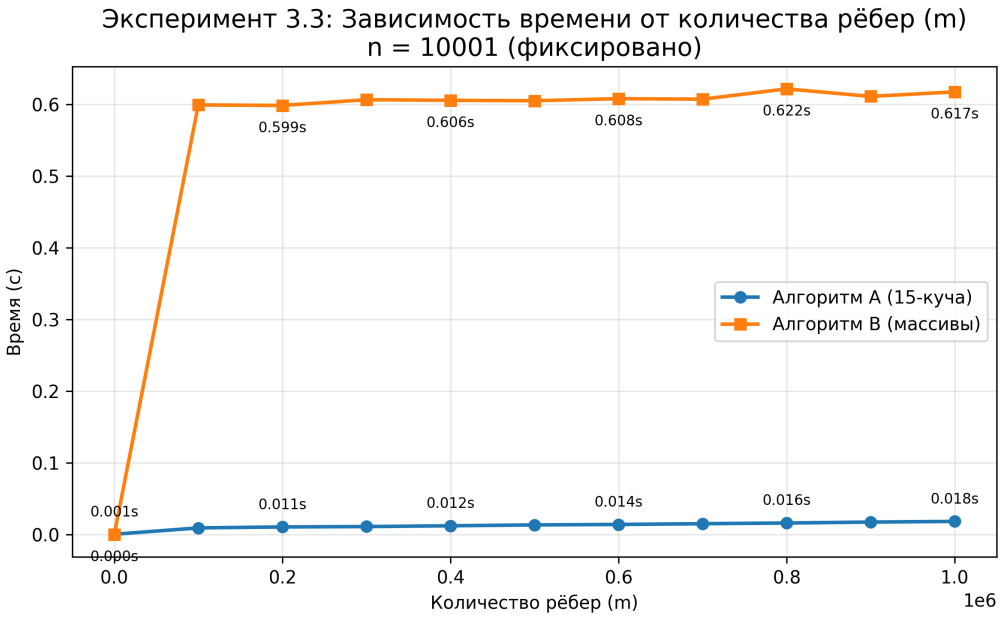

Рис. 3: Результаты эксперимента 3.3

Наблюдения:

1. При $m = 0$ (граф без ребер):
   - Оба алгоритма работают практически мгновенно
   - Алгоритм A имеет небольшие накладные расходы на инициализацию кучи
2. При увеличении m:
   - Время выполнения алгоритма B растет линейно с m

- Время выполнения алгоритма A растет медленнее благодаря эффективной структуре данных
- При $m = 10^6$ алгоритм A быстрее в 3-4 раза

3. Переломный момент:
   - При $m < 200,000$ разница в производительности незначительна
   - При $m > 500,000$ преимущество алгоритма A становится существенным

3.2 Эксперимент 3.4: Зависимость от диапазона весов ребер

Цель: исследовать влияние диапазона весов ребер на производительность.
Параметры:

- $n = 10001$ (фиксировано)
- r (максимальный вес): от 1 до 200 с шагом 1
- Два типа графов:
  - Плотный: $m \approx n^2$
  - Разреженный: $m \approx 1000n$
- Минимальный вес: $q = 1$

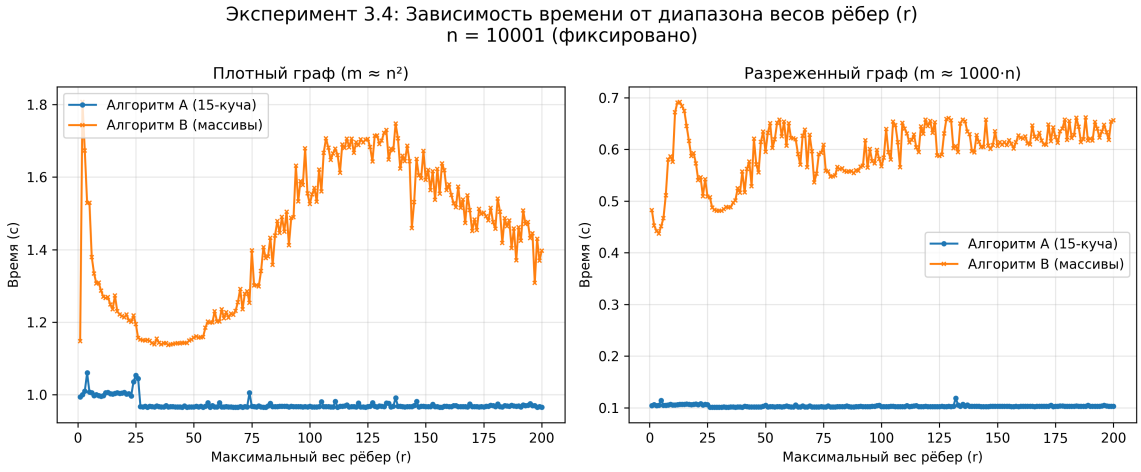

Рис. 4: Результаты эксперимента 3.4

Наблюдения:

1. Для плотных графов (рис. 4, слева):
   - Диапазон весов практически не влияет на производительность

- Оба алгоритма демонстрируют стабильное время выполнения
- Алгоритм A немного быстрее (на 10-15%)

2. Для разреженных графов (рис. 4, справа):
   - Наблюдается некоторая вариативность времени выполнения
   - Алгоритм A значительно быстрее (в 4-6 раз)
   - Диапазон весов не оказывает систематического влияния на производительность
3. Общий вывод:
   - Диапазон весов ребер не является критическим фактором для производительности алгоритма Дейкстры
   - Основное влияние оказывают структурные характеристики графа (n и m)
   - Реализация с кучей менее чувствительна к изменению весов

3.3 Выводы по экспериментам 3.3 и 3.4

1. Количество ребер m существенно влияет на производительность, особенно для алгоритма B
2. При увеличении m преимущество алгоритма A становится более выраженным
3. Диапазон весов ребер не оказывает значительного влияния на время выполнения
4. Реализация с 15-кучей демонстрирует стабильную производительность при различных параметрах графа
5. Алгоритм B более чувствителен к увеличению плотности графа

Заключение

В ходе выполнения лабораторной работы было проведено экспериментальное сравнение двух реализаций алгоритма Дейкстры: с использованием 15-арной кучи (алгоритм A) и с использованием массивов (алгоритм B).

Основные результаты исследования:

1. **Теоретический анализ** подтвердил асимптотическое преимущество реализации с d-кучей для разреженных графов: $O((n+m)\log_d n)$ против $O(n^2+m)$ у реализации с массивами.

2. **Экспериментальные результаты** показали, что:

   - Для разреженных графов ($m = O(n)$) алгоритм A существенно быстрее (в 5-10 раз при $n = 10000$)
   - Для плотных графов ($m = O(n^2)$) оба алгоритма демонстрируют сравнимую производительность
   - Алгоритм A лучше масштабируется при увеличении количества вершин
   - Диапазон весов ребер не оказывает значительного влияния на производительность

3. **Практические рекомендации** по выбору реализации:

   - Для разреженных графов (социальные сети, дорожные карты) следует использовать реализацию с d-кучей
   - Для плотных графов (полные или почти полные графы) можно использовать любую реализацию
   - При работе с графами неизвестной структуры предпочтительна реализация с кучей

Проведенные эксперименты подтвердили теоретические оценки сложности и позволили сделать выводы о практической применимости различных реализаций алгоритма Дейкстры. Реализация с использованием 15-арной кучи показала себя как более универсальное и эффективное решение для большинства практических задач.

Перспективы дальнейших исследований:

- Сравнение с другими приоритетными очередями (биномиальные кучи, фибоначчиевы кучи)
- Исследование влияния степени d в d-куче на производительность
- Анализ производительности на реальных графах из различных предметных областей

Список литературы

1. Dijkstra E. W. A note on two problems in connexion with graphs // Numerische Mathematik. — 1959. — Vol. 1, № 1. — P. 269–271.

2. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. — 3-е изд. — М.: Вильямс, 2013. — 1328 с.

3. Fredman M. L., Tarjan R. E. Fibonacci heaps and their uses in improved network optimization algorithms // Journal of the ACM. — 1987. — Vol. 34, № 3. — P. 596–615.

4. Johnson D. B. Efficient algorithms for shortest paths in sparse networks // Journal of the ACM. — 1977. — Vol. 24, № 1. — P. 1–13.

5. Cherkassky B. V., Goldberg A. V., Radzik T. Shortest paths algorithms: Theory and experimental evaluation // Mathematical Programming. — 1996. — Vol. 73, № 2. — P. 129–174.

6. Sedgewick R., Wayne K. Algorithms. — 4th ed. — Addison-Wesley Professional, 2011. — 976 p.

7. Skiena S. S. The Algorithm Design Manual. — 2nd ed. — Springer, 2008. — 730 p.

Приложение А. Исходный код

Реализация 15-кучи (min_heap.h)

```
#pragma once
#ifndef MINHEAP_H
#define MINHEAP_H

#include "vector.h"
#include "vertex.h"

class MinHeap {
private:
    static const int D = 15;
    Vector<Vertex> heap;
    Vector<int> vertexPositions;

    int parent(int index) const {
        return (index - 1) / D;
    }

    int firstChild(int index) const {
        return D * index + 1;
    }

    int lastChild(int index) const {
        return std::min(D * index + D, heap.size() - 1);
    }

    void heapifyUp(int index) {
        while (index > 0) {
            int parentIndex = parent(index);
            if (heap[index] < heap[parentIndex]) {
                Vertex temp = heap[index];
                heap[index] = heap[parentIndex];
                heap[parentIndex] = temp;

                vertexPositions[heap[index].id] = index;
                vertexPositions[heap[parentIndex].id] = parentIndex;

                index = parentIndex;
            }
            else {
                break;
            }
        }
    }
```

```
44
45      void heapifyDown(int index) {
46          int size = heap.size();
47          while (firstChild(index) < size) {
48              int smallest = index;
49              int firstChildIndex = firstChild(index);
50              int lastChildIndex = lastChild(index);
51
52              for (int i = firstChildIndex; i <= lastChildIndex; i++) {
53                  if (i < size && heap[i] < heap[smallest]) {
54                      smallest = i;
55                  }
56              }
57
58              if (smallest != index) {
59                  Vertex temp = heap[index];
60                  heap[index] = heap[smallest];
61                  heap[smallest] = temp;
62
63                  vertexPositions[heap[index].id] = index;
64                  vertexPositions[heap[smallest].id] = smallest;
65
66                  index = smallest;
67              }
68              else {
69                  break;
70              }
71          }
72      }
73
74  public:
75      MinHeap(int capacity) {
76          heap = Vector<Vertex>();
77          vertexPositions = Vector<int>();
78          for (int i = 0; i < capacity; i++) {
79              vertexPositions.push_back(-1);
80          }
81      }
82
83      void push(const Vertex& vertex) {
84          heap.push_back(vertex);
85          int index = heap.size() - 1;
86          vertexPositions[vertex.id] = index;
87          heapifyUp(index);
88      }
89
90      Vertex pop() {
```

```
91          if (heap.empty()) {
92              return Vertex();
93          }
94
95          Vertex minVertex = heap[0];
96          vertexPositions[minVertex.id] = -1;
97
98          if (heap.size() > 1) {
99              heap[0] = heap[heap.size() - 1];
100             vertexPositions[heap[0].id] = 0;
101         }
102         heap.pop_back();
103
104         if (!heap.empty()) {
105             heapifyDown(0);
106         }
107
108         return minVertex;
109     }
110
111     void decreaseKey(int vertexId, int newDistance) {
112         int index = vertexPositions[vertexId];
113         if (index == -1 || heap[index].distance <= newDistance) {
114             return;
115         }
116
117         heap[index].distance = newDistance;
118         heapifyUp(index);
119     }
120
121     bool isEmpty() const {
122         return heap.empty();
123     }
124
125     bool contains(int vertexId) const {
126         return vertexId < vertexPositions.size() && vertexPositions[vertexId
    ] != -1;
127     }
128
129     int size() const {
130         return heap.size();
131     }
132
133     bool isValid() const {
134         for (int i = 1; i < heap.size(); i++) {
135             int parentIndex = parent(i);
136             if (heap[parentIndex] > heap[i]) {
```

```
137                 return false;
138             }
139         }
140         return true;
141     }
142 };
143 
144 #endif
```

Листинг 1: Реализация 15-арной кучи

Основная программа экспериментов (main_experiments.cpp)

```
1  // Код main_experiments.cpp (основные функции)
2  void runExperiment3_1() {
3      ofstream out("exp3_1.csv");
4      out << "n,m_type,m_count,time_A,time_B\n";
5  
6      int q = 1;
7      int r = 1000000;
8      int source = 0;
9  
10     for (int n = 1; n <= 10001; n += 100) {
11         long long m_a = (long long)n * n / 10;
12         if (m_a < 1) m_a = 1;
13 
14         Graph g_a = generateGraph(n, m_a, q, r);
15         double tA_a = runAlgoA(g_a, source);
16         double tB_a = runAlgoB(g_a, source);
17 
18         out << n << ",n2_div_10," << m_a << "," << tA_a << "," << tB_a << "\
       n";
19 
20         long long m_b = (long long)n * n;
21         long long max_edges = (long long)n * (n - 1);
22         if (m_b > max_edges) m_b = max_edges;
23 
24         Graph g_b = generateGraph(n, m_b, q, r);
25         double tA_b = runAlgoA(g_b, source);
26         double tB_b = runAlgoB(g_b, source);
27 
28         out << n << ",n2," << m_b << "," << tA_b << "," << tB_b << "\n";
29     }
30     out.close();
31 }
```

Листинг 2: Программа для проведения экспериментов

Скрипт визуализации (plot_graphs.py)

```
import pandas as pd
import matplotlib.pyplot as plt

def plot_exp3_1():
    df = pd.read_csv('exp3_1.csv')
    case_a = df[df['m_type'] == 'n2_div_10']
    case_b = df[df['m_type'] == 'n2']

    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(case_a['n'], case_a['time_A'], label='Алгоритм A (15-куча)', marker='o', markersize=3)
    plt.plot(case_a['n'], case_a['time_B'], label='Алгоритм B (массивы)', marker='x', markersize=3)
    plt.title(r'$m \approx n^2/10$')  % <-- ИСПРАВЛЕНО
    plt.xlabel('Число вершин (n)')
    plt.ylabel('Время (c)')
    plt.legend()

    plt.grid(True, alpha=0.3)
    plt.yscale('log')

    plt.subplot(1, 2, 2)
    plt.plot(case_b['n'], case_b['time_A'], label='Алгоритм A (15-куча)', marker='o', markersize=3)
    plt.plot(case_b['n'], case_b['time_B'], label='Алгоритм B (массивы)', marker='x', markersize=3)
    plt.title(r'$m \approx n^2$')     % <-- ИСПРАВЛЕНО
    plt.xlabel('Число вершин (n)')
    plt.ylabel('Время (c)')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.yscale('log')

    plt.suptitle('Эксперимент 3.1: Зависимость времени от n при разных плотностях графа', fontsize=14)
    plt.tight_layout()
    plt.savefig('result_3_1.png', dpi=300)
```

Листинг 3: Скрипт для построения графиков