Содержание

ВВЕДЕНИЕ

В современном мире задачи оптимизации путей и поиска кратчайших маршрутов играют ключевую роль в различных областях информатики и телекоммуникаций. Алгоритмы поиска кратчайшего пути применяются в навигационных системах, маршрутизации сетевого трафика, логистике, транспортном планировании, робототехнике и множестве других сфер. Одним из наиболее эффективных и широко используемых методов решения данной задачи является алгоритм Дейкстры, предложенный нидерландским учёным Эдсгером Дейкстрой в 1956 году.

Алгоритм Дейкстры позволяет определить кратчайшие пути от одной выбранной вершины графа до всех остальных, что делает его универсальным инструментом при работе с моделями сетей и графовых структур. Его преимущество заключается в детерминированности, высокой точности и эффективности, особенно при использовании оптимизированных структур данных, таких как приоритетные очереди.

Цель данной курсовой работы — изучить принцип работы алгоритма Дейкстры, провести его программную реализацию и проанализировать эффективность на основе тестовых данных.

Результатом исследования станет программное решение, реализующее алгоритм Дейкстры. Выполнение данной работы позволит углубить знания в области алгоритмов и структур данных, а также закрепить практические навыки разработки программного обеспечения.

1 Постановка задачи

Задачей данной курсовой работы является разработка программы, реализующей быстрый вариант алгоритма Дейкстры для поиска кратчайших путей в ориентированном взвешенном графе с использованием приоритетной очереди, обеспечивающей асимптотическую сложность:

$$O(Elog(V)) \tag{1}$$

где:

E - количество ребер графа,

V - количество вершин графа.

Программа должна обеспечивать чтение графа из входного файла и формировать выходной файл в формате Graphviz для визуализации результата.

Алгоритм Дейкстры применяется для нахождения кратчайшего пути от заданной начальной вершины до всех остальных вершин графа. Кратчайший путь в данном контексте представляет собой цепочку связанных рёбер, соединяющих начальную и целевую вершины, имеющую наименьшую суммарную сумму весов рёбер среди всех возможных маршрутов. При этом учитывается направленность рёбер, и алгоритм корректно работает только для графов с неотрицательными весами рёбер. В графе могут существовать несколько путей с одинаковой минимальной суммарной длиной, и алгоритм должен корректно выбрать один из них.

Для наглядности рассмотрим граф с четырьмя вершинами, представленный в виде списка смежности:

A: (B, 1), (C, 4)

B: (C, 2), (D, 6)

C: (D, 3)

D: -

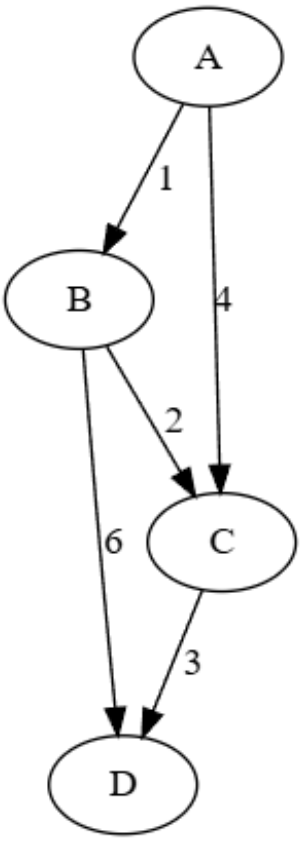

Рис. 1: Изображение графа

В графе, представленном на рис.1, существует несколько маршрутов от вершины A до вершины D:

1. A $\rightarrow$ B $\rightarrow$ D с суммарным весом $1 + 6 = 7$
2. A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D с суммарным весом $1 + 2 + 3 = 6$
3. A $\rightarrow$ C $\rightarrow$ D с суммарным весом $4 + 3 = 7$

Пример показывает, что в графе могут существовать различные маршруты с отличающимися суммарными весами, что обосновывает необходимость применения алгоритма поиска кратчайших путей, такого как алгоритм Дейкстры.

Данная задача подробно описана в книге «Алгоритмы. Построение и анализ» [8].

2 Алгоритм

2.1 Идеи алгоритма

Алгоритм Дейкстры предназначен для нахождения кратчайших путей от одной исходной вершины до всех остальных вершин ориентированного взвешенного графа с неотрицательными весами рёбер. Основная идея алгоритма заключается в жадном выборе вершины с минимальным текущим расстоянием на каждом шаге и обновлении расстояний до соседних вершин через эту вершину (операция «релаксации»).

2.2 Описание алгоритма

Быстрый вариант алгоритма реализуется с использованием приоритетной очереди (`std::priority_queue`) с компаратором `std::greater`, что позволяет эффективно выбирать вершину с минимальным расстоянием на каждом шаге. Это обеспечивает сложность алгоритма $O(ElogV)$.

Для практической реализации алгоритма в данной курсовой работе был разработан класс Graph, который содержит:

- конструктор для инициализации графа с заданным числом вершин;
- метод addEdge для добавления рёбер;
- метод dijkstra для вычисления кратчайших расстояний;
- метод readFromFile для загрузки графа из файла;
- метод writeResultToDot для формирования выходного файла в формате Graphviz;
- метод printResult для вывода результатов работы алгоритма на экран.

2.3 Структуры данных

В реализации класса Graph используются следующие структуры данных:

1. Список смежности (`adjacency_list`):
   - Представлен как `std::vector<std::vector<std::pair<int,int> > > adjacency_list`.
   - Каждая вершина графа хранит вектор пар (v, weight), где:
     v — индекс соседней вершины,
     weight — вес ребра.
   - Список смежности позволяет эффективно хранить ориентированные графы и экономить память по сравнению с матрицей смежности, особенно для разреженных графов.
2. Вектор расстояний (`dist`)
   - Представлен как `std::vector<int> dist(vertex_count, std::numeric_limits<int>::max())`.
   - Хранит текущее минимальное расстояние от исходной вершины до каждой вершины графа.
   - Изначально все значения устанавливаются в «бесконечность» (`std::numeric_limits<int>::max()`), кроме стартовой вершины, которая получает `dist[startNode] = 0`.
3. Вектор предыдущих вершин (`prev`)
   - Представлен как `std::vector<int> prev(vertex_count, -1)`.
   - Хранит индекс предыдущей вершины на кратчайшем пути для каждой вершины.
   - Позволяет восстановить полный путь после выполнения алгоритма и используется при формировании выходного файла DOT.

4. Приоритетная очередь (`std::priority_queue`)

   - Реализована как `std::priority_queue<PII, std::vector<PII>, std::greater<PII> > pq`, где PII — это `std::pair<int,int> (distance, vertex)`. Используется `std::greater`, чтобы сделать минимальную кучу. По умолчанию priority_queue — максимальная, то есть извлекает элемент с наибольшим значением. Для алгоритма Дейкстры нужно всегда брать вершину с наименьшим расстоянием, поэтому используется greater, который сравнивает пары по первому элементу (distance) и обеспечивает извлечение минимального расстояния первым.
   - Приоритетная очередь хранит вершины с их текущими минимальными расстояниями. Она реализована с помощью бинарной кучи, поэтому вершина с наименьшим расстоянием всегда находится в корне. Извлечение этой вершины и перестройка кучи выполняются за $O(logV)$.
   - При обновлении расстояния до соседей вершины новые пары (dist[v], v) добавляются в очередь.

На рис.2 представлена блок-схема алгоритма Дейкстры.

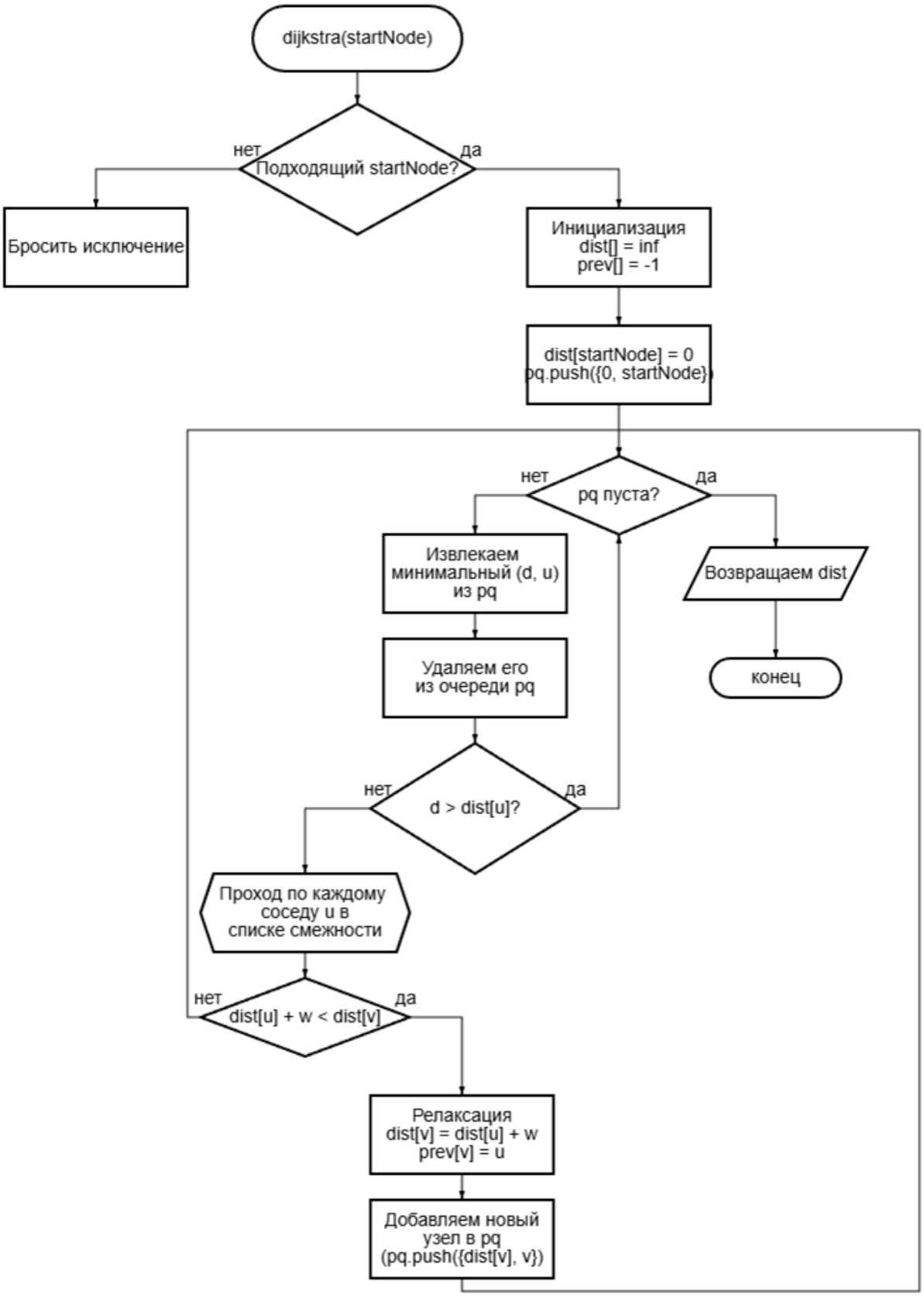

Рис. 2: Блок-схема алгоритма Дейкстры.

2.4 Пример работы алгоритма

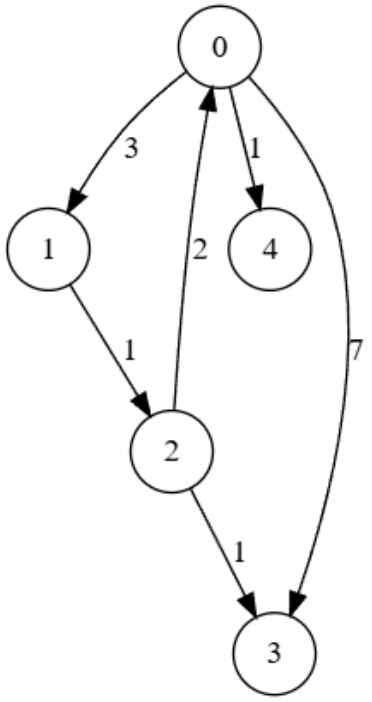

Рис. 3: Входной граф.

На вход программе подается ориентированный взвешенный граф, изображенный на рис.3, в файле input.txt, представленный в виде списка ребер (см. рис.4). Каждое ребро графа задается в формате <u v w>, где

u — индекс исходной вершины,

v — индекс конечной вершины,

w — вес ребра.

```
0 1 3
0 3 7
0 4 1
1 2 1
2 0 2
2 3 1
```

Рис. 4: Входной граф, представленный списком ребер в файле input.txt.

Рассмотрим пошаговую работу алоритма:

Шаг 0 (инициализация): перед началом работы алгоритма выполняется инициализация структур данных. Граф был прочитан из файла, на его основе сформирован список смежности `adjacency_list`:

`adjacency_list[0] = ((1,3), (3,7), (4,1));`

`adjacency_list[1] = (2,1);`

`adjacency_list[2] = ((0,2), (3,1));`

`adjacency_list[3] =; //Эта условная запись означает, у вершины нет исходящих рёбер.`

`adjacency_list[4] = ;`

Далее, происходит инициализация массивов и приоритетной очереди:

1. Расстояния (`dist`):

```
dist[0] = 0;
dist[1] = std::numeric_limits<int>::max(); // INF
dist[2] = std::numeric_limits<int>::max(); // INF
dist[3] = std::numeric_limits<int>::max(); // INF
dist[4] = std::numeric_limits<int>::max(); // INF
```

2. Предки (`last_prev`):

```
last_prev[0] = -1;
last_prev[1] = -1;
last_prev[2] = -1;
last_prev[3] = -1;
last_prev[4] = -1;
```

3. Приоритетная очередь (`pq`):

   В очередь добавляется стартовая вершина

```
pq.push(std::make_pair(0, 0));
```

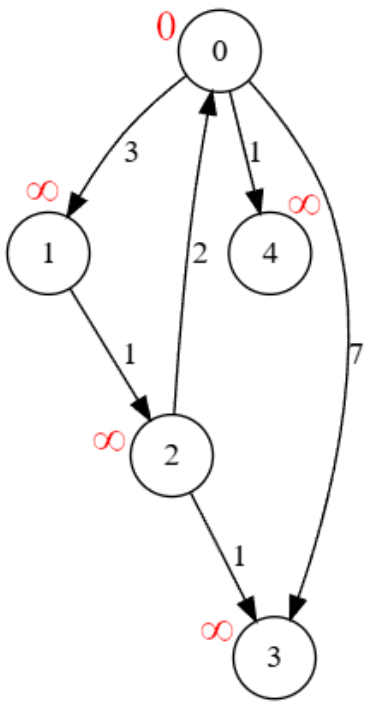

Рис. 5: Начальное состояние графа.

На рис.5 кратчайшие пути, установленные на текущем шаге, изображены красным цветом. Нулевой шаг соответствует состоянию графа до начала выполнения основного цикла алгоритма. На данном этапе происходит инициализация - оценки кратчайших путей до всех вершин равны бесконечности, что отражает отсутствие найденных путей.

Шаг 1 — извлекаем пару (0, 0) из `pq` и обрабатываем соседей. Приоритетная очередь хранит пары вида `(w, v)`, где:

- `w` — текущее минимальное расстояние от стартовой вершины до этой вершины,

- `v` — индекс вершины в графе.

Список смежности `adjacency_list[0]` содержит три ребра: (1,3), (3,7), (4,1). Для каждого соседа выполняется проверка релаксации:

```
if (dist[u] + w < dist[v])
```

1. Для вершины 1: $dist[0] + 3 = 0 + 3 < dist[1] = INF \rightarrow$ условие выполняется. Обновляем:

   ```
   dist[1] = 3
   last_prev[1] = 0
   pq.push((3,1))
   ```

2. Для вершины 3: $dist[0] + 3 = 0 + 3 < dist[1] = INF \rightarrow$ обновляем
   `dist[3] = 7`
   `last_prev[3] = 0`
   `pq.push(7,3))`.

Состояние структур после шага 1:

```
dist = [0, 3, INF, 7, 1];
last_prev = [-1, 0, -1, 0, 0];
pq = { (1,4), (3,1), (7,3) };
```

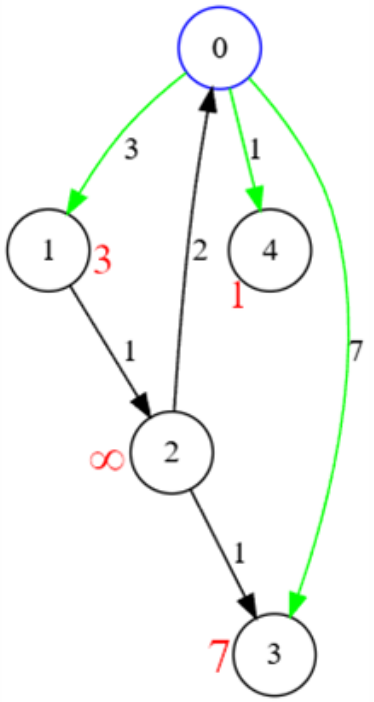

Рис. 6: Состояние графа после первого шага выполнения алгоритма.

На рис. 6 и последующих иллюстрациях приняты следующие условные обозначения: синий цвет используется для маркировки узла, который был выбран в качестве текущего обрабатываемого узла, красным цветом выделены актуальные оценки кратчайших путей до вершин, а зеленым цветом — рёбра, обработанные на текущем шаге работы алгоритма. Данная цветовая кодировка сохраняется для всех дальнейших шагов демонстрации.

Шаг 2 — обработка вершины 4, из очереди извлекается пара (1, 4).

Обработка соседей вершины 4: список смежности `adjacency_list[4]` пуст. Это означает, что у вершины 4 нет исходящих рёбер, поэтому релаксаций не выполняется и массивы `dist` и `last_prev` остаются без изменений.

Состояние структур после шага 2:

```
dist = [0, 3, INF, 7, 1];
last_prev = [-1, 0, -1, 0, 0];
pq = { (3,1), (7,3) };
```

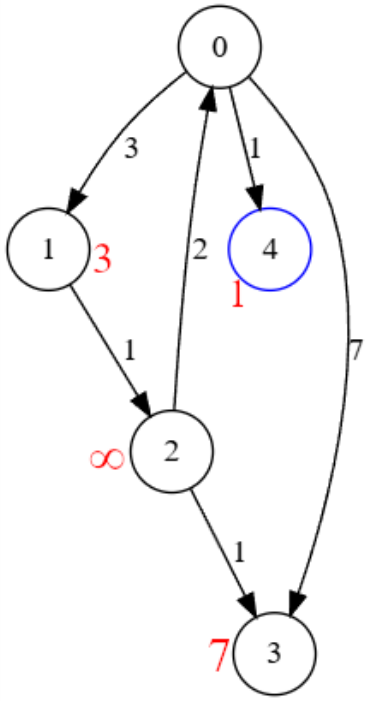

Рис. 7: Состояние графа после второго шага выполнения алгоритма.

Шаг 3 — обработка вершины 1. Извлечение вершины (3, 1).

Обработка соседей вершины 1: список смежности `adjacency_list[1]` содержит одно ребро: `adjacency_list[1]` - (2,1). Для соседа 2 выполняется проверка релаксации: $dist[1]+1 = 3+1 = 4 < dist[2] = INF \rightarrow$ обновляем. Обновляем массивы и очередь:

```
dist[2] = 4;
last_prev[2] = 1;
pq.push((4,2));
```

Состояние структур после шага 3:

```
dist = [0, 3, 4, 7, 1];
last_prev = [-1, 0, 1, 0, 0];
pq = { (4,2), (7,3) };
```

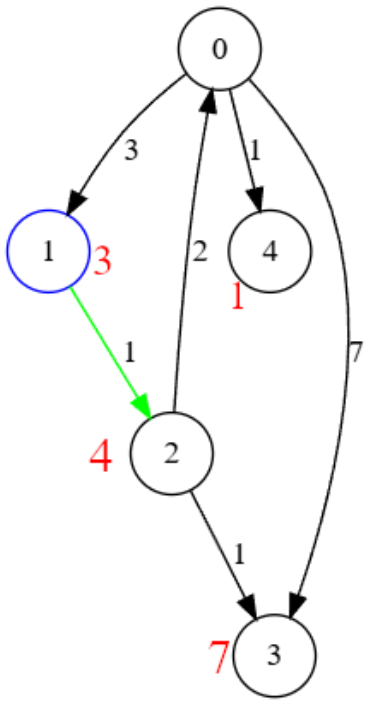

Рис. 8: Состояние графа после третьего шага выполнения алгоритма.

Шаг 4 — обработка вершины 2. На этом шаге извлекается пара (4, 2). Список смежности `adjacency_list[2]` содержит два ребра: `adjacency_list[2]` - (0,2), (3,1). Для каждого соседа выполняем проверку релаксации:

1. Для вершины 0: $\text{dist}[2] + 2 = 4 + 2 = 6 < \text{dist}[0] = 0 \rightarrow$ не выполняется. Значения массивов остаются без изменений.

2. Для вершины 3: $\text{dist}[2] + 1 = 4 + 1 = 5 < \text{dist}[3] = 7 \rightarrow$ обновляем. Обновляем массивы и очередь:

   ```
   dist[3] = 5;
   last_prev[3] = 2;
   pq.push((5, 3));
   ```

Состояние структур после шага 4:

```
dist = [0, 3, 4, 5, 1];
last_prev = [-1, 0, 1, 2, 0];
pq = { (5,3), (7,3) };
```

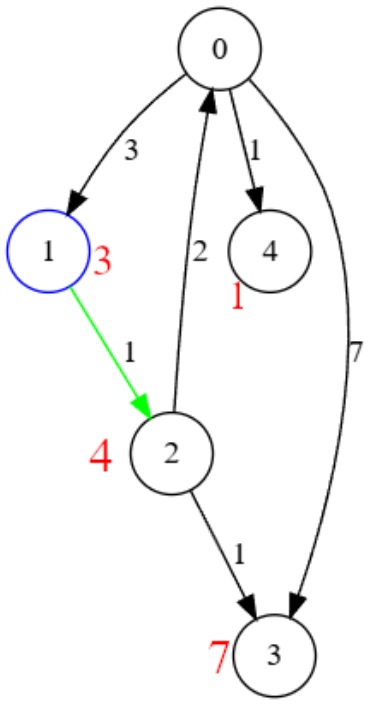

Рис. 9: Состояние графа после четвертого шага выполнения алгоритма.

Шаг 5 — обработка вершины 3. Из очереди извлекается (5, 3). `adjacency_list[3]` пуст, следовательно соседей нет, массивы `dist` и `last_prev` не меняются.

Очередь после шага: `pq = { (7,3) };`

Шаг 6 — обработка устаревшего элемента. Из очереди извлекается `(7, 3)`. Проверка: $d > dist[u]$ß$7 > 5 \rightarrow$ условие верно, элемент пропускается. Очередь пуста, следовательно алгоритм завершает работу.

Итоговые массивы после всех шагов:

```
dist = [0, 3, 4, 5, 1];
last_prev = [-1, 0, 1, 2, 0];
```

После завершения работы алгоритма Дейкстры, программа формирует массивы:

```
dist = [0, 3, 4, 5, 1];
last_prev = [-1, 0, 1, 2, 0];
```

Эти массивы используются для записи результата в файл формата Graphviz. Каждая вершина графа записывается в файл:

```
0 [shape=doublecircle, style=filled, fillcolor=blue]; //
стартовая вершина
1 [shape=circle];
2 [shape=circle];
3 [shape=circle];
4 [shape=circle];
```

Процедура сначала определяет все вершины графа. Стартовая вершина (`startVertex`) визуально выделяется: для неё устанавливается двойной круг, залитый синим цветом (`shape=doublecircle, style=filled, fillcolor=blue`). Все остальные вершины отображаются как обычные круги (`shape=circle`).

Далее программа проходит по списку смежности (`adjacency_list`), чтобы включить в выходной файл все рёбра исходного графа, включая их веса. Для каждого ребра, соединяющего вершину u с вершиной v и имеющего вес w, выполняется ключевая проверка с помощью массива `last_prev`.

Программа проверяет, является ли вершина u непосредственным предшественником вершины v в кратчайшем пути (условие `last_prev[v] == u`). Если ребро является частью кратчайшего пути, оно дополнительно оформляется красным цветом (`color=red`), иначе, оно выводится стандартным образом.

```
0 -> 1 [label=3, color=red];
0 -> 3 [label=7];
0 -> 4 [label=1, color=red];
1 -> 2 [label=1, color=red];
2 -> 0 [label=2];
2 -> 3 [label=1, color=red];
```

Таким образом, в файле содержится описание всех вершин и всех рёбер исходного графа, при этом рёбра, составляющие кратчайшие пути от стартовой вершины до всех достижимых вершин, визуально выделены красным цветом.

2.5 Оценка сложности алгоритма

Далее будет приведен псевдокод алгоритма Дейкстры:

Algorithm 1 Алгоритм Дейкстры для поиска кратчайших путей

```
1:  for v ← 0 to V − 1 do
2:      dist[v] ← +∞
3:      prev[v] ← −1
4:  end for
5:  dist[startNode] ← 0
6:  Создать приоритетную очередь pq;
7:  Поместить (0, startNode) в pq;
8:  while pq не пуста do
9:      (d, u) ← pq.top();
10:     pq.pop();
11:     if d > dist[u] then
12:         continue;
13:     end if
14:     for каждый (v, w) в adjacency_list[u] do
15:         if dist[u] + w < dist[v] then
16:             dist[v] ← dist[u] + w;
17:             prev[v] ← u;
18:             Поместить (dist[v], v) в pq;
19:         end if
20:     end for
21: end while
22: return dist;
```

Рассмотрим детальный анализ асимптотической сложности для графа с V вершинами и E рёбрами, представленного списком смежности.

В начале алгоритма выполняется инициализация массива расстояний `dist` и массива предшественников `prev` для всех V вершин графа, что требует $O(V)$ операций. Для исходной вершины расстояние устанавливается равным нулю, и она помещается в приоритетную очередь — операция вставки выполняется за $O(logV)$. Общая сложность этапа инициализации: $O(V)$.

Основной цикл алгоритма продолжается до тех пор, пока приоритетная очередь не пуста, выполняя две ключевые операции:

1. Извлечение минимума. На каждой итерации извлекается вершина с минимальным известным расстоянием. В реализации с двоичной

кучей без операции decrease-key допускаются дубликаты вершин в очереди, поэтому в худшем случае операции извлечения выполняются $O(V + E)$ раз ($O(E)$ для устаревших записей). Стоимость одной операции извлечения в двоичной куче составляет $O(logN_{pq})$, где $N_{pq} \leq V + E \approx O(E)$ — текущее число элементов в очереди. Общая стоимость всех извлечений: $O(ElogE)$.

2. Вставка. Операция вставки в приоритетную очередь происходит при каждой успешной релаксации. Поскольку для каждой вершины расстояние может улучшаться несколько раз (от разных входящих ребер), но общее число успешных релаксаций не превышает E (каждое ребро релаксируется ровно один раз, когда исходная вершина обработана), общее число вставок составляет $O(E)$. Стоимость одной вставки в двоичную кучу — $O(logE)$. Общая стоимость всех вставок: $O(ElogE)$.

Внутренний цикл обрабатывает все смежные ребра для каждой извлекаемой вершины, но только если эта вершина извлечена с актуальным минимальным расстоянием. Таким образом, обработка соседей происходит ровно один раз для каждой вершины, когда ее расстояние финализировано. Общее количество проверок условия релаксации равно сумме степеней выхода обработанных вершин, что составляет $O(E)$. Проверка условия релаксации, обновление расстояния и предшественника являются константными операциями $O(1)$. Общая стоимость обработки всех ребер: $O(E)$.

Объединяя все компоненты, общая асимптотическая сложность алгоритма складывается из: $O(V) + O(ElogE) + O(E)$. В связном графе минимальное число ребер равно $E_{min} = V-1$, поэтому даже в наименее плотном графе число ребер сравнимо с числом вершин, а для графов с большим числом ребер $ElogE$ растет быстрее, чем V или E, и, следовательно, доминирует в суммарной временной сложности алгоритма. Поскольку в графе с V вершинами максимальное число ребер равно $E_{max} = V(V-1)/2 \leq V^2$, логарифм $log(E)$ не превышает $log(V^2) = 2logV = O(logV)$. Следовательно, сложность можно выразить как $O(ElogV)$.

3 Инструкция пользователя

Программа предназначена для вычисления кратчайших путей в ориентированном взвешенном графе с помощью алгоритма Дейкстры и сохранения результата в формате DOT для последующей визуализации с помощью программы Graphviz.

Для запуска программы необходимо использовать командную строку с двумя параметрами: имя входного файла, имеющий расширение .txt, с описанием графа и имя выходного файла, имеющий расширение .dot, в который будет сохранено дерево кратчайших путей.

Программа автоматически определяет количество вершин графа на основе максимального индекса, найденного во входном файле, а в качестве стартовой вершины выбирается вершина с минимальным индексом.

Входной файл должен быть текстовым и содержать описание ребер графа, по одному ребру на строку. Каждое ребро задается тремя целыми числами: индекс вершины-источника, индекс вершины-приемника и вес ребра. Индексы вершин начинаются с нуля, а вес может быть любым натуральным числом больше нуля. Файл не должен содержать пустых строк или некорректных символов, иначе программа выдаст сообщение об ошибке и завершит работу. Пример входного файла изображен на рис.4

Выходной файл создается в формате DOT. Стартовая вершина выделена синим цветом и двойной рамкой. Ребра, входящие в дерево кратчайших путей, выделены красным цветом, остальные ребра отображаются обычным образом. Пример содержимого выходного файла приведён в приложении 1. Так же в приложении 1 показано, как этот результат выглядит при визуализации в программе Graphviz.

При выполнении программы кратчайшие расстояния от стартовой вершины также выводятся в консоль. Для каждой вершины отображается либо минимальное расстояние от стартовой вершины, либо сообщение о том, что вершина недостижима, если пути к ней нет.

Программа имеет следующие ограничения: индексы вершин должны быть целыми числами от нуля до количества вершин минус один; входной файл должен содержать хотя бы одно ребро; программа работает только с ориентированными графами.

4 Тестовые примеры

1. Тест 1:

   Входной файл `input.txt`:

   ```
   0 1 2
   1 2 3
   2 3 1
   0 3 1
   ```

   Выходной файл `graph.dot`:

   ```
   digraph G {
       0 [shape=doublecircle, style=filled, fillcolor=blue];
       1 [shape=circle];
       2 [shape=circle];
       3 [shape=circle];
       0 -> 1 [label=2, color=red];
       0 -> 3 [label=1, color=red];
       1 -> 2 [label=3, color=red];
       2 -> 3 [label=1];
   }
   ```

   Сообщение в консоль:

   ```
   Results written to: .\output\graph.dot
   Shortest distances from vertex 0:
   Vertex 0: 0
   Vertex 1: 2
   Vertex 2: 5
   Vertex 3: 1
   ```

2. Тест 2:

   Входной файл `input.txt`:

```
0 1 10
0 2 3
1 2 1
1 3 2
2 1 4
2 3 8
2 4 2
3 4 7
4 3 9
```

   Выходной файл `graph.dot`:

```
digraph G {
    0 [shape=doublecircle, style=filled, fillcolor=blue];
    1 [shape=circle];
    2 [shape=circle];
    3 [shape=circle];
    4 [shape=circle];
    0 -> 1 [label=10];
    0 -> 2 [label=3, color=red];
    1 -> 2 [label=1];
    1 -> 3 [label=2, color=red];
    2 -> 1 [label=4, color=red];
    2 -> 3 [label=8];
    2 -> 4 [label=2, color=red];
    3 -> 4 [label=7];
    4 -> 3 [label=9];
}
```

   Сообщение в консоль:

```
Results written to: .\output\graph.dot
Shortest distances from vertex 0:
Vertex 0: 0
```

```
Vertex 1: 7
Vertex 2: 3
Vertex 3: 9
Vertex 4: 5
```

3. Тест 3:

   Входной файл `input.txt`:

```
0 1 4
0 2 2
0 3 7
1 2 1
1 4 5
2 3 2
2 5 3
3 6 1
4 5 2
4 7 6
5 3 1
5 6 4
5 8 5
6 7 3
6 9 8
7 9 2
8 7 1
8 9 3
```

Выходной файл `graph.dot`:

```
digraph G {
    0 [shape=doublecircle, style=filled, fillcolor=blue];
    1 [shape=circle];
    2 [shape=circle];
    3 [shape=circle];
    4 [shape=circle];
    5 [shape=circle];
    6 [shape=circle];
    7 [shape=circle];
    8 [shape=circle];
    9 [shape=circle];
    0 -> 1 [label=4, color=red];
    0 -> 2 [label=2, color=red];
    0 -> 3 [label=7];
    1 -> 2 [label=1];
    1 -> 4 [label=5, color=red];
    2 -> 3 [label=2, color=red];
    2 -> 5 [label=3, color=red];
    3 -> 6 [label=1, color=red];
    4 -> 5 [label=2];
    4 -> 7 [label=6];
    5 -> 3 [label=1];
    5 -> 6 [label=4];
    5 -> 8 [label=5, color=red];
    6 -> 7 [label=3, color=red];
    6 -> 9 [label=8];
    7 -> 9 [label=2, color=red];
    8 -> 7 [label=1];
    8 -> 9 [label=3];
}
```

Сообщение в консоль:

```
Results written to: .\output\graph.dot
Shortest distances from vertex 0:
Vertex 0: 0
Vertex 1: 4
Vertex 2: 2
Vertex 3: 4
Vertex 4: 9
Vertex 5: 5
Vertex 6: 5
Vertex 7: 8
Vertex 8: 10
Vertex 9: 10
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была изучена теоретическая основа и реализован быстрый вариант алгоритма Дейкстры для поиска кратчайших путей в ориентированных взвешенных графах. Программа разработана с использованием языка C++ и применяет приоритетную очередь, что позволило достичь асимптотической сложности $O(ElogV)$, подтверждающей эффективность алгоритма при обработке графов средней и высокой плотности.

В процессе работы были рассмотрены структура и принципы функционирования алгоритма, проведён анализ его временной сложности, а также реализованы функции чтения данных из входного файла, визуализации результатов в формате Graphviz и вывода кратчайших расстояний на экран.

Разработанное программное решение прошло проверку на ряде тестовых примеров, что подтвердило корректность его работы и соответствие заявленным требованиям. Программа успешно находит кратчайшие пути между вершинами и формирует корректное графическое представление результата.

Полученные результаты могут быть использованы при решении практических задач маршрутизации, оптимизации сетевых путей, логистики и других областей, где требуется определение кратчайших маршрутов в графовых структурах.

Выполнение курсовой работы позволило закрепить знания в области алгоритмов и структур данных, а также развить навыки проектирования и реализации программ на языке C++.

СПИСОК ЛИТЕРАТУРЫ

Список литературы

1. Doxygen Manual. — URL: https://www.doxygen.nl/manual/index.html (дата обр. 02.11.2025) ; [Электронный ресурс].
2. Graphviz Documentation. — URL: https://graphviz.org/documentation/ (дата обр. 30.10.2025) ; [Электронный ресурс].
3. priority_queue — C++ Reference. — URL: https://en.cppreference.com/w/cpp/container/priority_queue (дата обр. 28.10.2025) ; [Электронный ресурс].
4. Pseudocode // Built In. — URL: https://builtin.com/data-science/pseudocode (дата обр. 05.11.2025) ; [Электронный ресурс].
5. Алгоритм Дейкстры / Википедия. — URL: https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D1%8B (дата обр. 26.10.2025) ; [Электронный ресурс].
6. Алгоритм Дейкстры // Algorithmica. — URL: https://ru.algorithmica.org/cs/shortest-paths/dijkstra/ (дата обр. 27.10.2025) ; [Электронный ресурс].
7. Алгоритм Дейкстры // Prog-Cpp. — URL: https://prog-cpp.ru/deikstra/ (дата обр. 01.11.2025) ; [Электронный ресурс].
8. Алгоритмы: построение и анализ : [пер. с англ.] / Т. Х. Кормен [и др.]. — 2-е изд. — Москва : Вильямс, 2019. — С. 1328.