ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет информатики, математики и компьютерных наук

**Программа подготовки бакалавров по направлению
01.03.02 Прикладная математика и информатика**

Антонов Илья Витальевич

КУРСОВАЯ РАБОТА

Задача о кратчайшем пути

Научный руководитель
к.ф.-м.н., доцент

_____

Е.К. Бацына

Нижний Новгород, 2022

Оглавление

Введение

В настоящее время известно множество алгоритмических задач на графах. Одной из наиболее известных является задача о кратчайшем пути. Она возникает во многих сферах человеческой деятельности и имеет значимое практическое приложение. Например, эта задача появляется в современных системах GPS-навигации, когда необходимо построить оптимальный маршрут из одной точки в другую. Эти системы используют граф следующим образом: вершинами обозначаются перекрестки дорог, ребрами - участки дорог между перекрестками, а также каждому ребру задается значение, называемое весом ребра. Весом ребра могут выступать различные параметры: длина дороги, время затрачиваемое на проезд по дороге, стоимость проезда по дороге и прочее. Алгоритмы поиска кратчайшего пути позволяют успешно решать данную задачу.

На настоящий момент существует множество алгоритмов поиска кратчайшего пути в графах. Наиболее распространенным и популярным является алгоритм Дейкстры, разработанный в 1959 году нидерландским ученым Эдсгером Вибе Дейкстрой. Этот алгоритм позволяет находить кратчайший путь от заданной вершины до всех вершин в графе без ребер отрицательного веса. Рассмотрению алгоритма Дейкстры и его модификаций, направленных на уменьшение времени работы алгоритма, посвящена данная курсовая работа.

1. Постановка задачи

Сформулируем задачу о кратчайшем пути. Дан неориентированный граф $G = (V, E)$, где $V = \{1, \ldots, n\}$ - множество вершин графа, помеченных натуральными числами от 1 до n, $E \subset V \times V$ - множество ребер графа. На множестве ребер определена положительная вещественнозначная функция $c: E \to \mathbb{R}^+$, значение которой $c(e)$ для конкретного ребра $e \in E$ называется весом ребра. Также заданы вершины $start \in V$ и $finish \in V$. Требуется установить существует ли путь из вершины $start$ до вершины $finish$, а также, если он существует, найти такой путь $u_1, \ldots, u_k$, где $start = u_1$, $finish = u_k$ и $(u_i, u_{i+1}) \in E$ при $i = \overline{1, k-1}$, для которого значение функции $\sum_{i=1}^{k-1} c(u_i, u_{i+1})$ минимально.

Предполагается, что граф задается одним из следующих способов:

1. Матрицей расстояний $(m_{i,j})$, $i = \overline{1, n}$, $j = \overline{1, n}$, где n - количество вершин в графе, определенной по правилу:

$$m_{i,j} = \begin{cases} c(i,j), \text{ если } (i,j) \in E \\ \infty, \text{ если } (i,j) \notin E \end{cases}$$

   Для компьютерного хранения матрицы расстояний, вместо ∞ используются числа, которые заведомо не могут быть весами ребер в поставленных условиях.

2. Множеством ребер $\{(i, j, c(i,j)) : (i,j) \in E\}$, $i = \overline{1, n}$, $j = \overline{1, n}$, где n - количество вершин в графе.

2. Описание алгоритма Дейкстры

Пусть дана постановка задачи из гл. 1. Алгоритм Дейкстры находит кратчайшие пути от заданной стартовой вершины до всех вершин графа, в которые существует путь из стартовой вершины. В случае, когда требуется найти кратчайший путь до определенной финишной вершины, в момент ее посещения можно сделать досрочное завершение алгоритма. При этом кратчайший путь до этой вершины уже будет найден. Введем следующие обозначения:

- V - множество вершин графа, где $|V| = n \in \mathbb{N}$
- E - множество ребер графа, где $|E| = m \in \mathbb{N}$
- $start \in V$ - стартовая вершина
- $finish \in V$ - финишная вершина
- $c[i, j]$ - вес ребра $(i, j) \in E$.
- $d[i]$ - оценка расстояния от $start$ до $i \in V$
- $pred[i]$ - вершина из которой пришли в вершину $i \in V$
- U - множество посещенных вершин

Работа алгоритма Дейкстры строится следующим образом. Сначала вершине $start$ устанавливается оценка кратчайшего пути $d[start] := 0$, другим же вершинам устанавливается оценка $d[v] := \infty$. Для всех вершин инициализируем $pred[v] := \infty$, где ∞ обозначает, что у вершины на данном этапе нет предшественника. Далее начинается итерационный процесс, который будет продолжаться пока в $V \setminus U$ есть вершины. Опишем алгоритм его работы. Из $V \setminus U$ извлекается вершина x с минимальной оценкой $d[x]$. Эта вершина добавляется в множество посещенных вершин U. Для всех ребер исходящих из x в $v \in V \setminus U$ проводим процесс называемый релаксацией ребер. Если $d[x] + c[x, v] < d[v]$, то изменяем значение $d[v] := d[x] + c[x, v]$, а также устанавливаем вершине v вершину предшественника x, присваивая

$pred[v] := x$. Затем переходим к следующему шагу итерационного процесса. По окончании работы алгоритма, длина кратчайшего пути будет находиться в $d[finish]$. Если он существует, то значение $d[finish]$ будет отлично от ∞. При решении различных практических задач важно знание пути, длина которого минимальна. Для его восстановления удобно использовать рекурсивную процедуру. Воспользуемся тем, что в $pred[v]$ лежит номер вершины из которой мы пришли в вершину v. Заведем список $path$, в котором будем хранить путь. Операция $path.push\_back(x)$ добавляет число x в конец списка $path$. Ниже приведен псевдокод процедуры, восстанавливающей кратчайший путь. Предполагается, что перед выполнением процедуры список $path$ - пустой и $d[end] \neq \infty$. Для запуска процедуры 1 и 2 параметрами передаются массив $pred$ и список $path$, третьим параметром номер вершины end.

Алгоритм 1 Алгоритм восстановления пути

```
1: function RestorePath(pred, path, v)
2:     if pred[v] ≠ ∞ then
3:         RestorePath(pred, path, pred[v])
4:         path.push_back(v)
5:     else
6:         path.push_back(v)
7:         return
8:     end if
9: end function
```

После завершения работы процедуры RestorePath, в списке $path$ в правильном порядке с начала списка лежат вершины, по которым проходит кратчайший путь. Теперь приведем псевдокод алгоритма Дейкстры.

Алгоритм 2 Алгоритм Дейкстры

```
1: function DIJKSTRA(start, finish, V, E, c)
2:     U ← ∅
3:     for v ∈ V do
4:         d[v] ← ∞
5:         pred[v] ← ∞
6:     end for
7:     d[start] ← 0
8:     while V \ U ≠ ∅ do
9:         x ← ∞, d_min ← ∞
10:        for v ∈ V \ U do
11:            if d[v] ≤ d_min then
12:                d_min ← d[v]
13:                x ← v
14:            end if
15:        end for
16:        U ← U ∪ {x}
17:        for v смежные для x do
18:            if v ∉ U and d[x] + c[x, v] < d[v] then
19:                d[v] ← d[x] + c[x, v]
20:                pred[v] ← x
21:            end if
22:        end for
23:    end while
24:    return d[finish]
25: end function
```

3. Модификации алгоритма Дейкстры

В данном разделе будут рассмотрены модификации алгоритма Дейкстры, направленные на улучшение его реализации по времени работы.

3.1. Использование списков смежности

Сложность алгоритма Дейкстры сильно зависит от структуры данных, на которой он построен. При подсчете асимптотики алгоритма необходимо обращать внимание на сложность двух основных операций: извлечения вершины с минимальной оценкой расстояния и релаксацию ребер. Обычно для реализации U используют массив булевского типа данных, то есть $U[i] \in \{0, 1\}$ при $i \in V$, где значение 1 принимается в случае, если вершина была посещена, а значение 0 в противном случае. Для нахождения вершины с минимальной оценкой расстояния в наивной реализации (см. гл. 2) используют линейный перебор всех вершин графа с использованием массива U и дополнительной проверкой не посещенных вершин на минимальное значение.

Извлечение вершины с минимальной оценкой расстояния при таком подходе занимает $\mathcal{O}(n)$. Всего в графе n вершин, поэтому с учетом стоимости операции для одной итерации, суммарная сложность выполнения операции извлечения вершин $\mathcal{O}(n^2)$. Релаксация одного ребра занимает $\mathcal{O}(1)$, потому что она требует ограниченное количество операций, работающих за $O(1)$. Остается понять, как можно прорелаксировать все ребра и какое время это займет.

Существует два подхода как это можно сделать. Первый подход заключается в использовании матрицы расстояний. Сила и одновременно слабость такого подхода заключается в том, что данная матрица хранит отношения фиксированной вершины со всеми вершинами графа. То есть, чтобы прорелаксировать все ребра графа, надо перебрать все возможные пары вершин, и посмотреть есть ли между ними ребра. Сложность релаксации ребер при таком подходе $\mathcal{O}(n^2)$, а значит сложность наивной реализации алгоритма Дейкстры с использованием матрицы расстояний $\mathcal{O}(2n^2)$.

Однако возникает вопрос: как можно избежать перебора всех возможных пар вершин для релаксации ребер, и ограничиться только перебором всех ребер. Это рассуждение приводит нас к второму подходу. Для каждой вершины $i \in V$ будем хранить список $\{j \in V : (i, j) \in V\}$ вершин смежных ей. Чтобы выбрать для релаксации ребра, исходящие из i вершины, достаточно проитерироваться по данному списку. При таком подходе в процессе релаксации один раз возникают ребра, которые действительно содержатся в графе, и сложность релаксации всех ребер составляет $\mathcal{O}(m)$. Таким образом, сложность наивной реализации алгоритма Дейкстры с использованием списков смежности $\mathcal{O}(n^2 + m)$, что не хуже чем сложность алгоритма с использованием матрицы расстояний $\mathcal{O}(2n^2)$.

К тому же, формирование списков смежности оптимальнее по времени чем формирование матрицы расстояний. Оно занимает время $\mathcal{O}(n + m)$, в то время как матрица расстояний формируется за $\mathcal{O}(n^2 + m)$. Также списки смежности выгоднее по памяти. Их размер $\mathcal{O}(n + m)$, в то время как размер матрицы расстояний $\mathcal{O}(n^2)$.

3.2. Прерывание алгоритма Дейкстры

Поставленная в гл. 1 задача предполагает нахождение кратчайшего пути от стартовой вершины до финишной. Однако, алгоритм Дейкстры находит кратчайшие пути от стартовой вершины до всех достижимых вершин графа. Предлагается принцип, базирующийся на следующем соображении. Используя тот факт, что в момент посещения вершины до неё найден кратчайший путь, можно завершать иттерационный процесс алгоритма Дейкстры в момент посещения финишной вершины. Такая модификация не ухудшит время работы алгоритма, а в случае, когда кратчайший путь достигается не в вершине с самой большой оценкой кратчайшего пути - улучшит время работы. Для её реализации достаточно после извлечения вершины с минимальной оценкой кратчайшего пути добавить проверку на достижение финишной вершины:

Алгоритм 3 Проверка на достижение финишной вершины

```
1: if x == finish then
2:     break
3: end if
```

3.3. Модификации, основанные на приоритетных очередях

Подход к выбору вершин с минимальной оценкой кратчайшего пути, изложенный в гл. 2, хорошо применим на плотных графах, для которых $m \approx n^2$. Однако, в случаях когда граф является разреженным с $m = \mathcal{O}(n)$ ребрами, перебор всех вершин становится далеко не оптимальным. Разреженные графы возникают в различных областях практики. Например, граф дорог, построенный по карте местности, является планарным. Для таких графов известно, что количество ребер в них $\mathcal{O}(n)$, а следовательно они разреженные, что еще раз подтверждает необходимость оптимизации алгоритма Дейкстры.

Одним из ключевых направлений в таких случаях является использование различных приоритетных очередей для извлечения вершин. Сформулируем подход к использованию приоритетных очередей, а также рассмотрим различные их варианты.

Помимо обозначений из гл. 2, нам потребуется определить приоритетную очередь. Обозначаться она будет символом Q. Операция $Q.push(i, j)$ будет добавлять вершину $j \in V$ с приоритетом i. Операция $Q.pop()$ будет извлекать из $Q \neq \emptyset$ вершину $j \in V$ с i минимальным приоритетом во всей очереди Q. Теперь, используя приоритетную очередь, модифицируем алгоритм Дейкстры. Объявим пустую приоритетную очередь Q. Перед началом работы алгоритма добавим вершину $start$ с приоритетом 0 в Q. Вместо условия $V \setminus U \neq \emptyset$ сделаем условие $Q \neq \emptyset$. Оно отражает ту же логику: пока есть вершины к рассмотрению, продолжаем иттерационный процесс. Теперь вместо поиска вершины с минимальной оценкой кратчайшего пути в $V \setminus U$ мы будем извлекать вершину из Q, используя операцию $Q.pop()$. Эта замена сохраняет то же самое значение - роль приоритета вершины $v \in V$ играет значение $d[v]$. Чтобы Q пополнялась вершинами, после релаксации ребер добавляем в Q вершины с обновленными значениями $d[v]$, используя операцию $q.push(d[v], v)$. Однако это вызывает новую проблему, а именно: после нескольких процедур релаксации значение $d[v]$ для какой-либо вершины v может обновляться, а следовательно в приоритетной очереди одна и та же вершина может появляться несколько раз с различными приоритетами. Чтобы лишний раз не проводить релаксации, которая уже были совершены, мы

будем проверять была ли ранее посещена вершина. Если была, то перейдем к следующей итерации. Такой подход является самым простым и оптимальным. Интуитивно хочется проверять дублирование вершины в приоритетной очереди после ее добавления, но на практике большинство приоритетных очередей либо не поддерживают такую функцию, либо это достаточно трудоемко по времени. Приведем псевдокод алгоритма Дейкстры с использованием приоритетной очереди.

Алгоритм 4 Алгоритм Дейкстры на приоритетной очереди

```
1:  function DIJKSTRA(start, finish, V, E, c)
2:      U ← ∅
3:      Q ← ∅
4:      for v ∈ V do
5:          d[v] ← ∞
6:          pred[v] ← ∞
7:      end for
8:      d[start] ← 0
9:      Q.push(0, start)
10:     while Q ≠ ∅ do
11:         x ← Q.pop()
12:         if x ∈ U then
13:             continue
14:         end if
15:         U ← U ∪ {x}
16:         for v смежные для x do
17:             if v ∉ U and d[x] + c[x, v] < d[v] then
18:                 d[v] ← d[x] + c[x, v]
19:                 pred[v] ← x
20:                 Q.push(d[v], v)
21:             end if
22:         end for
23:     end while
24:     return d[finish]
25: end function
```

3.3.1. Приоритетная очередь, построенная на d-куче

d-кучи, где $d \in \mathbb{N}$, $d > 1$, являются одной из наиболее удобных структур данных для построения приоритетных очередей. В их основе лежит d-арное дерево, для реализации которого на $size$ элементов используют массив $data[0 \dots size - 1]$. Значение $data[i]$ называется ключем вершины с индексом i, где $i < size$. Под ключом и добавляемыми элементами мы будем понимать пару $(d[v], v)$. Операции сравнения пар будут работать по первому элементу пары. В этом дереве для вершины с индексом i потомками являются вершины с индексами из множества:

$$child[i] = \{j \in \{i \cdot d + 1, \dots, i \cdot d + d\} : j < size\}$$

Это дерево подчиняется свойству неубывающей кучи, то есть

$$\forall i < size \; \forall j \in child[i] : data[i] \; \leq \; data[j]$$

k-м уровнем d-арного дерева при $k \geq 0$ называется множество вершин:

$$D_k = \left\{ i \in \left\{ \left(\sum_{0 \leq j < k} d^j \right), \left(\sum_{0 \leq j < k} d^j \right) + 1, \dots, \left(\sum_{0 \leq j < k} d^j \right) + d^k - 1 \right\} : i < size \right\}$$

Номером последнего уровня d-арного дерева называется k такое, что $D_k \neq \emptyset$ и $D_{k+1} = \emptyset$. Все уровни d-арного дерева в куче, кроме, может быть, последнего, заполнены полностью. Последний уровень дерева заполнен слева направо, в смысле индексов. То есть, если i - наименьший индекс вершины на последнем уровне, то на последнем уровне будут вершины с индексами $i, \dots, size-1$.

Так как данная структура данных имеет свойство неубывающей кучи, то элемент с минимальным ключом, при условии того, что куча не пустая это $data[0]$. Остается разобраться с тем, как ввести операции $push$ и pop, которые сохраняют все свойства d-кучи. В дальнейшем, для удобства описания данных операций будем предполагать, что известно число $size$ и массив $data$ бесконечный. На практике же надо контролировать выход за границы массива и при необходимости увеличивать количество выделенной под него памяти средствами выбранного для реализации языка программирования.

Для реализации функции $push$ воспользуемся следующими фактами: для любой вершины с индексом $i \geq 1$ - индекс родительской вершины

$\lfloor (i-1)/d \rfloor$, вершина с индексом 0 является корнем d-арного дерева и родительской вершиной не обладает. Пусть add элемент, добавляемый в кучу, $size$ - текущий размер кучи. Работа функции $push$ строится следующим образом. Добавляем add в конец массива и увеличиваем значение $size$ на 1. Такое добавление сохраняет структуру d-арного дерева. Но после него нарушается структура неубывающей кучи. Исправим это путем просеивания дерева снизу вверх. Положим $i := size - 1$. Пока $i > 0$ и $data[i] < data[\lfloor (i-1)/d \rfloor]$, то есть нарушено свойство неубывающей кучи, меняем значения $data[i]$ и $data[\lfloor (i-1)/d \rfloor]$, а зачем присваиваем новое значение $i := \lfloor (i-1)/d \rfloor$. Такое просеивание восстанавливает структуру неубывающей кучи.

Действительно, если $data[i] < data[\lfloor (i-1)/d \rfloor]$, то неубывание нарушается лишь в одном месте. Так как данная структура - куча, то:

$$\forall j \in child[\lfloor (i-1)/d \rfloor],\ j \neq i : data[\lfloor (i-1)/d \rfloor] \leq data[j]$$

После обмена значений $data[i]$ и $data[\lfloor (i-1)/d \rfloor]$ восстанавливается свойство неубывания, т.к. в терминах до обмена значений:

$$data[i] < data[\lfloor (i-1)/d \rfloor] \leq data[j]$$

Алгоритм 5 Алгоритм добавления элемента в d-кучу

```
1: function PUSH(add, data, size)
2:     data[size] ← add
3:     i ← size++
4:     while i > 0 and data[i] < data[⌊(i − 1)/d⌋] do
5:         swap(data[i], data[⌊(i − 1)/d⌋])
6:         i ← ⌊(i − 1)/d⌋
7:     end while
8: end function
```

Для реализации функции pop, воспользуемся тем, что данная структура - куча и в $data[0]$ минимальное значение. Сохраним его, чтобы в дальнейшем вернуть. Поменяем значения $data[0]$ и $data[size-1]$, затем уменьшим $size$ на 1. Теперь из кучи извлечен минимальный элемент, но нарушено неубывание. Проведем процедуру просеивания кучи сверху вниз. Положим

$i := 0$. Если существует $j \in child[i]$ такой, что $data[j] < data[i]$ - меняем значения $data[i]$ и $data[j]$. После такой операции, у i вершины относительно всех потомков восстанавливается свойство неубывающей кучи, но при этом оно, возможно, нарушится у j вершины. Положим $i := j$ и продолжим данный процесс пока существует $j \in child[i]$ такой, что $data[j] < data[i]$. Приведем псевдокод данной процедуры:

Алгоритм 6 Алгоритм извлечения элемента из d-кучи

```
1:  function POP(data, size)
2:      min ← data[0]
3:      swap(data[0], data[−− size])
4:      i ← 0
5:      while true do
6:          j ← i
7:          for k = 1; k ≤ d; ++k do
8:              if i · d + k < size and data[i · d + k] < data[j] then
9:                  j ← i · d + k
10:             end if
11:         end for
12:         if i == j then
13:             break
14:         else
15:             swap(data[i], data[j])
16:             i ← j
17:         end if
18:     end while
19:     return min
20: end function
```

Время выполнения операции *push* и *pop* пропорционально высоте d-арного дерева, которая ограничена сверху значением $\lceil \log_d size \rceil$. Следовательно сложность операций *push* и *pop* составляет $\mathcal{O}(\log size)$. Обычно на практике используют значения d равные $2, 4$ и 8. При использовании d-куч в алгоритме Дейкстры релаксация ребра занимает $\mathcal{O}(\log n)$, так как при ре-

лаксации осуществляется добавление вершины в приоритетную очередь за $\mathcal{O}(\log n)$. Всего в графе m ребер, поэтому сложность релаксации всех ребер $\mathcal{O}(m \log n)$. Извлечение вершины из приоритетной очереди происходит за $\mathcal{O}(\log n)$, поэтому суммарно вершины будут извлекаться $\mathcal{O}(n \log n)$ раз. То есть сложность алгоритма Дейкстры с использованием d-куч составляет $\mathcal{O}(n \log n + m \log n)$.

3.3.2. Приоритетная очередь, построенная на фибоначчиевой куче

Майкл Фридман и Роберт Тарьян разработали и предложили использовать в качестве приоритетной очереди для алгоритма Дейкстры структуру данных, называемую фибоначчиевой кучей. Фибоначчиева куча строится на основе биномиальных деревьев. Биномиальное дерево ранга k обозначается B_k и определяется рекуррентно: дерево B_0 содержит один узел, дерево B_k при $k \geq 1$ состоит из двух биномиальных деревьев B_{k-1}, связанных вместе таким образом, что корень одного дерева B_{k-1} является потомком корня другого дерева B_{k-1}. Каждый узел в таком дереве содержит следующие поля: $left$ и $right$ - ссылки на левого и правого братьев, $child$ - ссылка на левого потомка, $degree$ - степень вершины, key - ключ вершины. Аналогично разд. 3.3.1 под ключом вершины и добавляемыми элементами мы будем понимать пару $(d[v], v)$. Операции сравнения пар будут работать по первому элементу пары.

Такая куча состоит из набора биномиальных деревьев. Для того, чтобы связать эти деревья в одну структуру, их корни считаются братьями и образуют двусвязный циклический список, называемый списком корней. Во всех деревьях ключ любого узла не больше ключа любого из его потомков. Для проведения различных операций доступ к фибоначчиевой куче осуществляется через ссылку на корень с минимальным ключом, который будем обозначать как min, также будем хранить $size$ - количество элементов в куче.

Чтобы построить алгоритмы над кучей нам потребуется вспомогательная операция $MergeLists$, которая объединяет корневые списки двух различных фибоначчиевых куч. Пусть $first$ и $second$ - корни из различных куч.

Алгоритм 7 Алгоритм объединения корневых списков

```
1: function MergeLists(first, second)
2:     l ← first.left
3:     r ← second.right
4:     second.right ← first
5:     first.left ← second
6:     l.right ← r
7:     r.left ← l
8: end function
```

Для добавления к куче узла с ключом key, к списку корней кучи присоединяется одноэлементное дерево B_0 с ключом корня key. Такая функция $push$ работает за $\mathcal{O}(1)$.

Алгоритм 8 Алгоритм добавления узла в фибоначчиеву кучу

```
1: function PUSH(min, key, size)
2:     new_node.key ← key
3:     new_node.degree ← 0
4:     new_node.left ← new_node
5:     new_node.right ← new_node
6:     if min == null then
7:         min ← new_node, size ← 1
8:     else
9:         MergeList(min, new_node)
10:        if new_node.key < min.key then
11:            min ← new_node
12:        end if
13:        size++
14:    end if
15: end function
```

Для построения функции удаления из фибоначчиевой кучи, нам потребуется функция $Consolidate$, которая собирает из всех деревьев корневого списка - деревья разных рангов. Согласно [1] максимальный ранг дерева в куче на $size$ элементов составляет $D = \lfloor \lg size \rfloor$. Создадим массив $A[0 \dots D] \leftarrow null$, где $A[i]$ - ссылка на корень дерева ранга i. Начнем перебирать корни всех деревьев. Пусть текущее дерево ранга i. Если $A[i] == null$, то в $A[i]$ помещаем ссылку на текущее дерево. В противном случае, объединяем текущее дерево с деревом из $A[i]$ путем добавления корня дерева с большим ключом корня в список потомков дерева с меньшим ключом корня, а затем удаляем из $A[i]$ ссылку на старое дерево. Получается дерево ранга $i+1$. Для него повторяем вышеописанные действия до тех пор, пока не получим дерево с рангом, которого не было в массиве A.

Алгоритм 9 Алгоритм Consolidate

```
1: function CONSOLIDATE(min, size)
2:     A[0 . . . D] ← null
3:     for каждый корень current в списке корней do
4:         x ← current
5:         d ← current.degree
6:         while A[d] ≠ null do
7:             y ← A[d]
8:             if x.key > y.key then
9:                 swap(x, y)
10:            end if
11:            x.degree + +
12:            y.left ← y
13:            y.right ← y
14:            if x.child ≠ null then
15:                MergeList(x.child, y)
16:            else
17:                x.child ← y
18:            end if
19:            A[d] ← null
20:            d + +
21:        end while
22:        A[d] ← x
23:    end for
```

После выполнения вышеприведенной части алгоритма *Consolidate* сформированы биномиальные деревья различных рангов. Ссылки на их корни лежат в массиве $A[0 \ldots D]$. Остается связать корни в двусвязный циклический список и определить минимальный элемент, по ссылке на который будет осуществляться доступ к куче.

Алгоритм 10 Алгоритм Consolidate(продолжение)

```
24:     min ← null
25:     for i = 0; i ≤ D; i++ do
26:         if A[i] ≠ null then
27:             A[i].left ← A[i]
28:             A[i].right ← A[i]
29:             if min == null then
30:                 min ← A[i]
31:             else
32:                 MergeList(min, A[i])
33:                 if A[i].key < min.key then
34:                     min ← A[i]
35:                 end if
36:             end if
37:         end if
38:     end for
39: end function
```

Как нам известно, доступ к фибоначчиевой куче осуществляется через ссылку на корень с минимальным ключом, обозначаемый как min. Чтобы удалить минимальный элемент из кучи, корневой список его потомков через ссылку на левого потомка добавляют в корневой список фибоначчиевой кучи, затем минимальный элемент отрезают от кучи путем изменения ссылок у его правого и левого братьев. На этом этапе производят прорежение деревьев, то есть формируют в куче биномиальные деревья различных рангов. Для этого используют рассмотренную выше функцию $Consolidate$. Помимо прорежения деревьев, она устанавливает в min ссылку на минимальный корень. В [1] показано, что за счет использования $Consolidate$ амортизированное время работы функции pop составляет $\mathcal{O}(\log size)$.

Алгоритм 11 Алгоритм извлечения узла с минимальным ключом

```
1: function POP(min, size)
2:     if size > 0 then
3:         removable ← min
4:         key ← removable.key
5:         if removable.child ≠ null then
6:             MergeList(removable, removable.child)
7:         end if
8:         l ← removable.left
9:         r ← removable.right
10:        l.right ← r
11:        r.left ← l
12:        min ← removable.right
13:        size − −
14:        if size == 0 then
15:            min ← null
16:        else
17:            Consolidate()
18:        end if
19:        return key
20:    else
21:        return error
22:    end if
23: end function
```

При использовании фибоначчиевых куч в алгоритме Дейкстры релаксация одного ребра происходит за $\mathcal{O}(1)$, так как сложность добавления вершины в фибоначчиеву кучу составляет $\mathcal{O}(1)$. Релаксация всех ребер занимает $\mathcal{O}(m)$. Сложность извлечения вершины из приоритетной очереди $\mathcal{O}(\log n)$, поэтому извлечение всех вершин происходит за $\mathcal{O}(n \log n)$. Сложность алгоритма Дейкстры при таком подходе $\mathcal{O}(n \log n + m)$.

3.3.3. Приоритетная очередь, построенная на двухуровневых корзинах

Если вес любого ребра в графе измеряется целым неотрицательным числом, то в таком случае для построения приоритетной очереди можно использовать структуру данных - двухуровневую корзину. Её работа базируется на следующем факте: если известен C - максимальный вес ребра в графе, то в момент посещения вершины v с оценкой кратчайшего пути $d[v]$, в очереди приоритетами могут быть значения $d[v], d[v]+1, \ldots, d[v]+C$. Таких значений $C+1$ штук. Известно, что в графе с n вершинами максимальная длина пути равна $lenght = (n-1)C$. Разобьем множество $\{0, 1, \ldots, lenght\}$ на k не пересекающихся подмножеств $B_0 = \{0, ..., r-1\}$, $B_1 = \{r, \ldots, 2r-1\}$, …, $B_i = \{ir, \ldots, (i+1)r-1\}$, …, $B_k = \{kr, \ldots, lenght\}$, длины $r = \lceil\sqrt{C+1}\rceil$ для всех B_i, кроме, возможно, B_k. Зададим массив корзин первого уровня $b[0 \ldots k]$. В корзину $b[i]$ попадают вершины $v \in V$ для которых $d[v] \in B_i$. Внутри каждой корзины первого уровня будет массив корзин второго уровня $b[i].c[0, \ldots, r-1]$, где ячейка $b[i].c[j]$ это список $v \in V$ для которых $d[v] = j + ir$. В переменной $size$ будем хранить количество вершин во всех корзинах первого уровня, аналогично в переменной $b[i].size$ будем хранить количество вершин в i корзине первого уровня. Для удобства при извлечении вершин на первом уровне заведем указатель $pointer$, такой что $\forall i < pointer : b[i].size = 0$, который будет указывать либо на корзину первого уровня в которой лежит вершина v с минимальным $d[v]$, либо на последнюю корзину из которой была извлечена минимальная вершина. Аналогичный указатель $b[i].pointer$ будет и на втором уровне корзин.

Добавление вершины v с приоритетом $d[v]$ в такую структуру осуществляется довольно просто. Нужно найти для $d[v]$ сначала индекс корзины первого уровня по формуле $i = \lfloor d[v]/r \rfloor$, затем в этой корзине разместить номер вершины v в позицию $b[i].c[d[v] - ir]$, и обновить указатели $pointer$, $b[i].pointer$, если добавляемая вершина по приоритету меньше вершин которые уже содержатся в корзинах на 1 или 2 уровнях, либо если корзины 1 или 2 уровня до добавления были пустыми.

Алгоритм 12 Алгоритм добавления в двухуровневую корзину

```
1: function PUSH(v, d[v], b, size, r)
2:     i ← ⌊d[v]/r⌋
3:     b[i].c[d[v] − ir].push_back(v)
4:     if size == 0 or i < pointer then
5:         pointer ← i
6:     end if
7:     if b[i].size == 0 or d[v] − ir < b[i].pointer then
8:         b[i].pointer ← d[v] − ir
9:     end if
10:    size + +, b[i].size + +
11: end function
```

Если $size \neq 0$, то для извлечения вершины v с минимальным $d[v]$, используя $pointer$, находим непустую корзину $b[i]$ первого уровня, а в ней используя $b[i].pointer$, находим непустую корзину второго уровня.

Алгоритм 13 Алгоритм извлечение минимума из двухуровневой корзины

```
1: function POP(b, size)
2:     if size ≠ 0 then
3:         while b[pointer].size == 0 do
4:             pointer + +
5:         end while
6:         while b[pointer].c[b[pointer].pointer] == ∅ do
7:             b[pointer].pointer + +
8:         end while
9:         v = b[pointer].c[b[pointer].pointer].pop_back()
10:        size − −, b[i].size − −
11:        return v
12:    else
13:        return error
14:    end if
15: end function
```

В целях экономии памяти можно не создавать k корзин первого уровня. Достаточно держать в памяти $\lceil\sqrt{C+1}\rceil+1$ зацикливающихся корзин первого уровня, каждая из которых содержит $\lceil\sqrt{C+1}\rceil$ корзин второго уровня.

Для извлечения вершины из такой структуры нужно среди $\lceil\sqrt{C+1}\rceil+1$ корзин первого уровня справа от корзины с индексом *pointer* найти непустую, а затем в ней среди $\lceil\sqrt{C+1}\rceil$ корзин второго уровня, найти корзину, содержащую вершину. Таким образом, извлечение вершины из такой структуры данных занимает $\mathcal{O}(2\sqrt{C+1})$. Добавление вершины, очевидно, происходит за $\mathcal{O}(1)$. Таким образом, алгоритм Дейкстры с двухуровневыми корзинами работает за $\mathcal{O}(m+2n\sqrt{C+1})$.

3.3.4. Гибрид двухуровневых корзин и 4-кучи

В [2] упомянуто, что двухуровневые корзины эффективны, если максимальный вес ребра в графе C - небольшое число. Если C - большое число, то диапазоны размера $\sqrt{C+1}$ достаточно широки и вероятность попадания хотя бы одной вершины в каждую из корзин первого уровня велика. А значит плотность корзин первого уровня, заполненных вершинами, большая. Следовательно, нахождение непустой корзины первого уровня с высокой долей вероятности занимает небольшое относительно верхней оценки $O(\sqrt{C+1})$ для этой операции время. Проблемы при больших C возникают на этапе поиска вершин на втором уровне корзин. В графах с большими весами ребер маловероятна ситуация, при которой разница между оценками кратчайшего пути мала. Значит, плотность вершин в корзинах второго уровня небольшая и их поиск становится достаточно долгим. Часто его сложность доходит до практически худшего случая $O(\sqrt{C+1})$. Однако хочется использовать такую структуру и при больших C. В данной работе предлагаем создать гибрид двухуровневых корзин и 4-кучи.

На первом уровне двухуровневых корзин мы сохраним ту же структуру. На втором уровне, вместо $\lceil\sqrt{C+1}\rceil$ элементного массива будет использована 4-куча. При добавлении элемента сначала будет находиться диапазон корзины первого уровня, в который он попадает, а затем в этой корзине элемент будет размещаться в 4-кучу. Таким образом добавление работает за $\mathcal{O}(1+\log n) = \mathcal{O}(\log n)$. Извлечение будет аналогично извлечению в двухуровневых корзинах, только теперь на втором уровне, мы будем извлекать элемент из 4-кучи. Суть данного улучшения состоит в том, что в этой структуре сложность поиска на втором уровне корзин зависит не от размера массива $\lceil\sqrt{C+1}\rceil$, а лишь от фактического количества вершин, попавших в корзину. Такое извлечение работает за $\mathcal{O}(\sqrt{C+1}+\log n)$, так как сначала за $\mathcal{O}(\sqrt{C+1})$ находим непустую корзину первого уровня, а в ней за $\mathcal{O}(\log n)$ извлекаем элемент из 4-кучи.

Таким образом, сложность алгоритма Дейкстры с нашей структурой данных составляет $\mathcal{O}(n\log n + n\sqrt{C+1} + m\log n)$. Заметим, что данная асимптотика похожа на асимптотику алгоритма с d-кучей (см. разд. 3.3.1), однако за счет того, что у нас $\lceil\sqrt{C+1}\rceil + 1$ куч в корзинах первого уровня,

то максимальная наполняемость любой кучи в нашей модификации должна быть меньше чем при подходе из разд. 3.3.1, а значит и операции с кучами в нашей модификации должны работать быстрее.

Для эффективности по памяти при реализации, как и в разд. 3.3.3 на первом уровне будем хранить $\lceil\sqrt{C+1}\rceil+1$ зацикливающихся корзин, каждой из которых соответствует своя 4-куча.

Заметим, что отказавшись от идеи хранения корзин второго уровня, мы сняли ограничение на целочисленность для веса ребер. При выборе корзины первого уровня достаточно проверять попадает ли вершина в определенный диапазон.

4. Сравнительный анализ модификаций алгоритма Дейкстры

4.1. Сравнение времени работы алгоритма Дейкстры при использовании матрицы расстояний и списков смежности

Для сравнения времени работы алгоритма Дейкстры при использовании матрицы расстояний и списков смежности на языке программирования C++ были написаны две наивные реализации алгоритма Дейкстры. Одна реализация использовала матрицу расстояний, другая списки смежности. Было проведено измерение времени работы алгоритма на неориентированных графах для которых количество ребер $m \approx 3n$. На основе этих измерений был получен график (рис. 4.1).

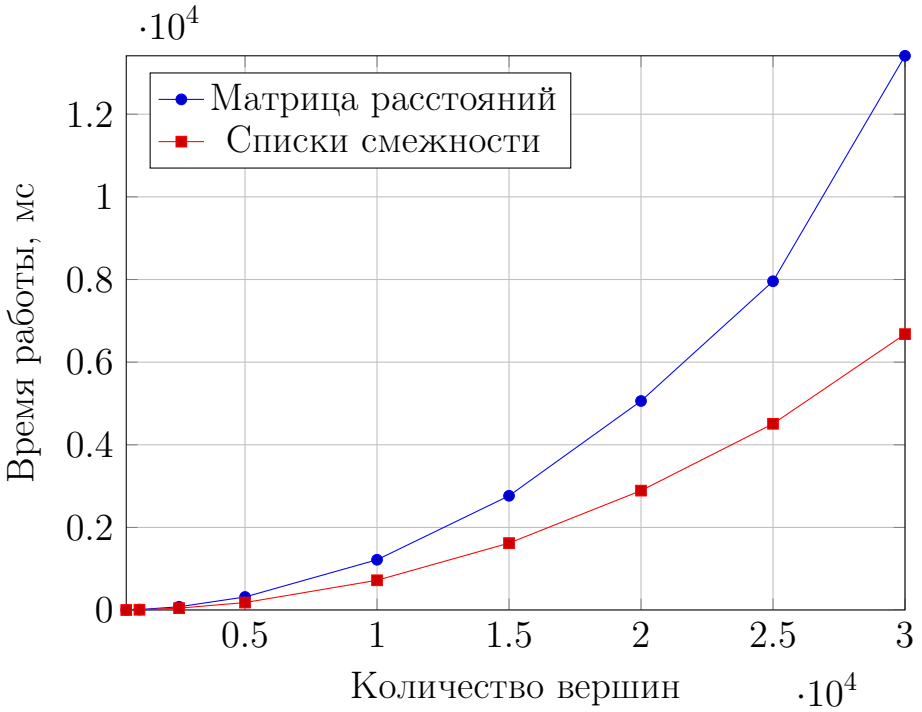

Рис. 4.1. График сравнения времени работы алгоритма Дейкстры при использовании матрицы расстояний и списков смежности на графах с $m \approx 3n$ ребрами

Уже при 15000 вершинах реализация с матрицей расстояний работает в $1,7$ раза дольше, чем реализация с списками смежности. А при 30000 вершинах эти значения различаются в 2 раза. Вышеприведенный график и соображения, описанные в разд. 3.1, подтверждают целесообразность использования списков смежности вместо матрицы расстояний в реализации алгоритма Дейкстры.

4.2. Сравнение времени работы алгоритма Дейкстры при использовании приоритетных очередей и без их использования

Для сравнения времени работы алгоритма Дейкстры при использовании приоритетных очередей и без их использования на языке программирования C++ были реализованы две модификации алгоритма Дейкстры на основе списков смежности. Первая использовала наивную реализацию, вторая использовала приоритетную очередь - двоичную кучу. Были проведены две серии испытаний. Первая серия проводилась на неориентированных графах для которых $m \approx 3n$. По её результатам построен график (рис. 4.2).

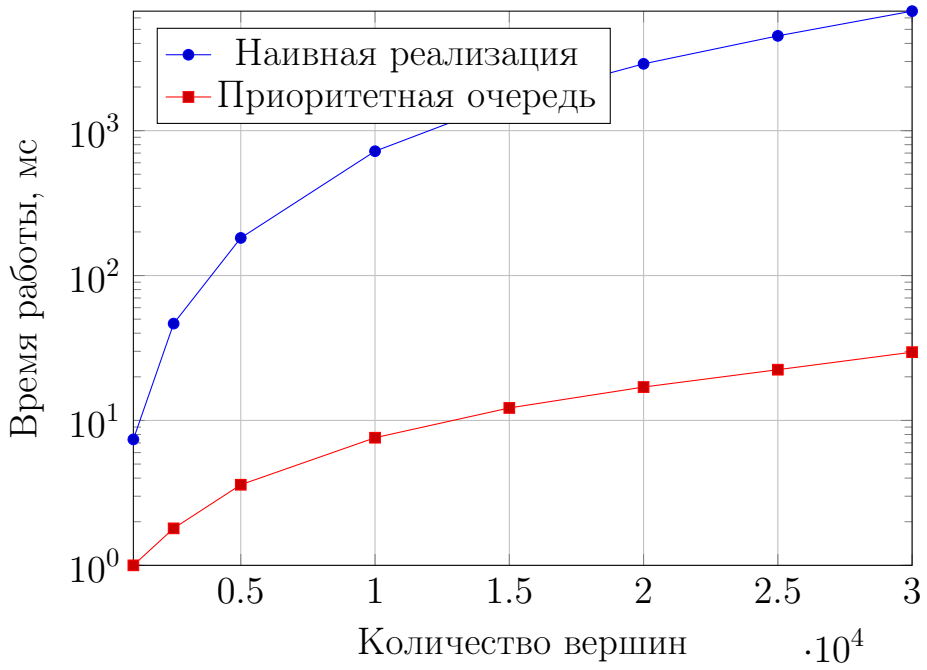

Рис. 4.2. График сравнения времени работы алгоритма Дейкстры при использовании приоритетной очереди и без её использования на графах с $m \approx 3n$ ребрами

Из (рис. 4.2) видно, что для разреженных графов, к классу которых относятся исследуемые, использование приоритетных очередей значительно экономит время. Например, при 30000 вершинах наивная реализация работает в 230 раз дольше, чем модификация основанная на приоритетной очереди - двоичной куче.

Вторая серия испытаний проводилась на неориентированных полных графах, для которых количество ребер равно C_n^2. По её результатам был построен график (рис. 4.3).

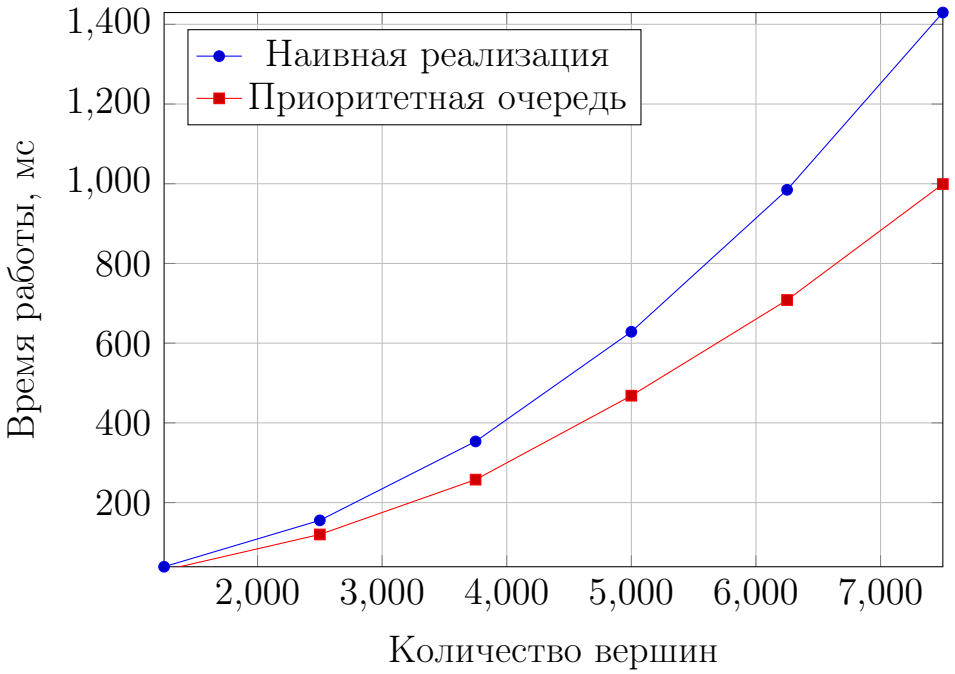

Рис. 4.3. График сравнения времени работы алгоритма Дейкстры при использовании приоритетной очереди и без её использования на полных графах

В начале разд. 3.3 было упомянуто, что наивная реализация алгоритма Дейкстры оптимальна на графах с $m \approx n^2$ ребрами. Из (рис. 4.3) видно, что, в принципе, это суждение верно, однако приоритетная очередь, основанная на двоичной куче, оказывается более оптимальной для таких графов, и, например, в полном графе с 7500 вершинами работает в $1,43$ раза быстрее.

Исходя из (рис. 4.2), (рис. 4.3) и вышеприведенного анализа, можно сделать вывод, что при реализации алгоритма Дейкстры наиболее целесообразно использовать приоритетные очереди.

4.3. Сравнение времени работы алгоритма Дейкстры при использовании различных приоритетных очередей

Для сравнения времени работы алгоритма Дейкстры с различными приоритетными очередями на языке программирование C++ был реализован алгоритм Дейкстры на основе списков смежности с следующими структурами данных:

1. Двухуровневые корзины
2. Гибрид двухуровневых корзин и 4-кучи (далее - гибрид)
3. 8-куча
4. 4-куча
5. 2-куча
6. Фибоначчиева куча

Так как время работы некоторых структур данных, а именно двухуровневых корзин и гибрида зависит от C - максимального веса ребра, то целесообразно при тестировании помимо количества вершин и ребер учитывать и этот параметр.

4.3.1. Сравнение для графов с количеством ребер $O(n)$

Для графов с количеством ребер $O(n)$ было проведено 2 серий испытаний. В первой серии испытаний исследовалось время работы алгоритма Дейкстры с приоритетными очередями на неориентированных графах, для которых n - количество вершин варьируется от 50000 до 150000, количество ребер $m \approx 30n$. Также исследовалось, что будет с временем работы алгоритма, если веса ребер находятся в пределах: 1-10000, 10000-1000000, 1000000-100000000. На основании результатов первой серии испытаний были построены следующие графики (рис. 4.4), (рис. 4.5), (рис. 4.6).

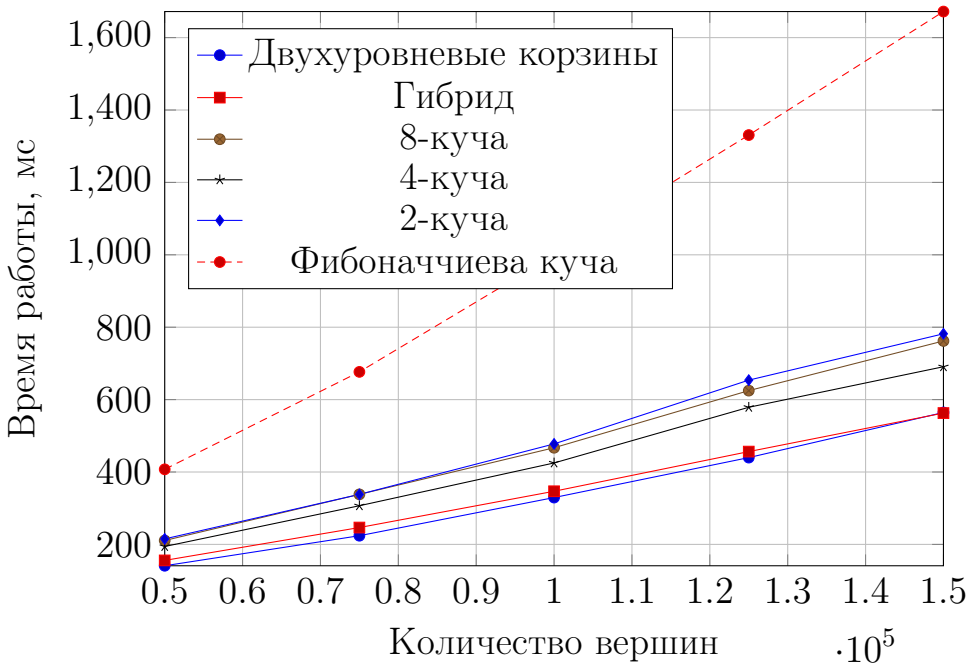

Рис. 4.4. График сравнения времени работы алгоритма Дейкстры при использовании приоритетных очередей на графах с $m \approx 30n$ вершинами и весами ребер в пределах 1-10000

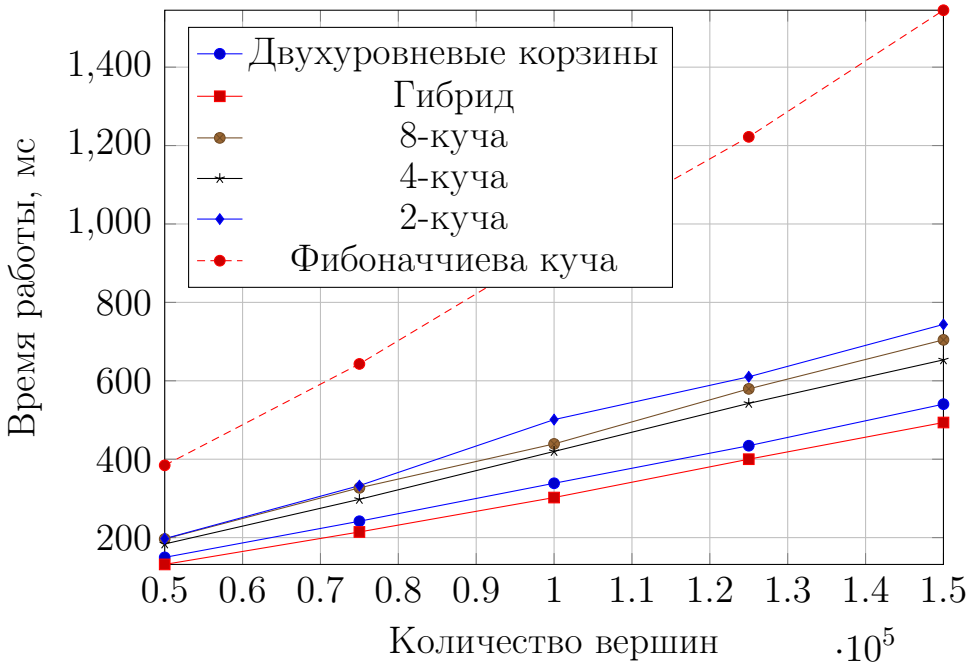

Рис. 4.5. График сравнения времени работы алгоритма Дейкстры при использовании приоритетных очередей на графах с $m \approx 30n$ вершинами и весами ребер в пределах 10000-1000000

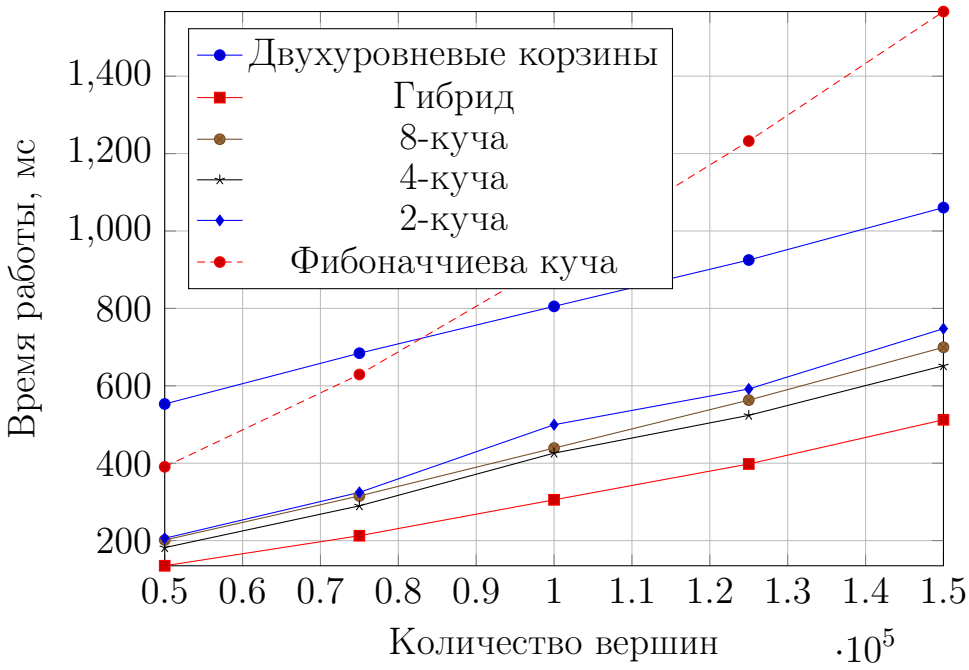

Рис. 4.6. График сравнения времени работы алгоритма Дейкстры при использовании приоритетных очередей на графах с $m \approx 30n$ вершинами и весами ребер в пределах 1000000-10000000

В второй серии испытаний измерялось время работы алгоритма Дейкстры с приоритетными очередями на неориентированных графах, для которых n - количество вершин варьируется от 100000 до 1000000, количество ребер $m \approx 20n$ при весах ребер в диапазонах 1-10000, 10000-1000000, 1000000-100000000. На основании результатов второй серии испытаний были построены следующие графики (рис. 4.7), (рис. 4.8), (рис. 4.9).

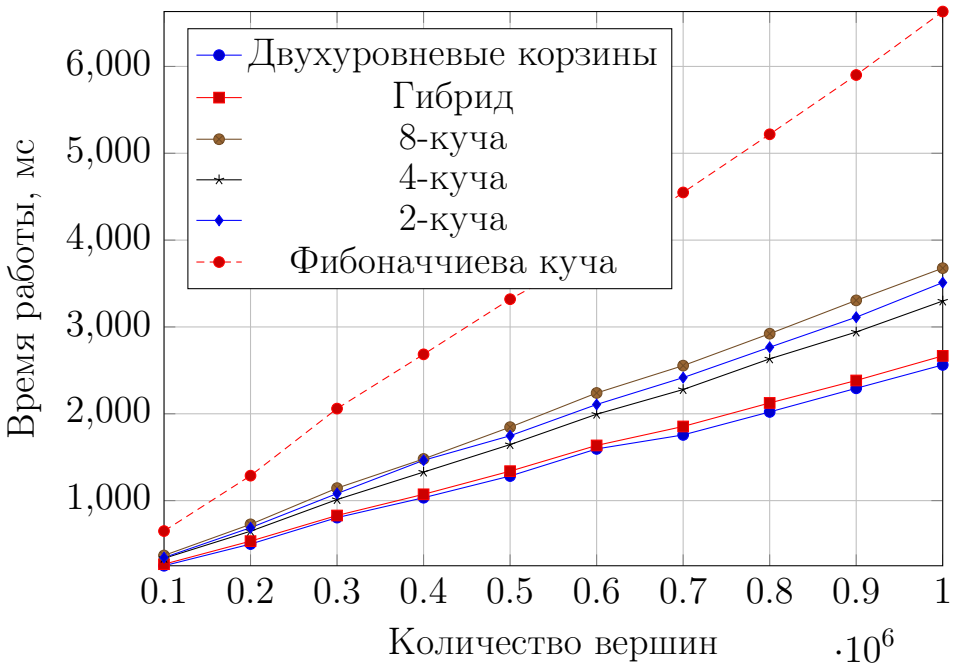

Рис. 4.7. График сравнения времени работы алгоритма Дейкстры при использовании приоритетных очередей на графах с $m \approx 20n$ вершинами и весами ребер в пределах 1-10000

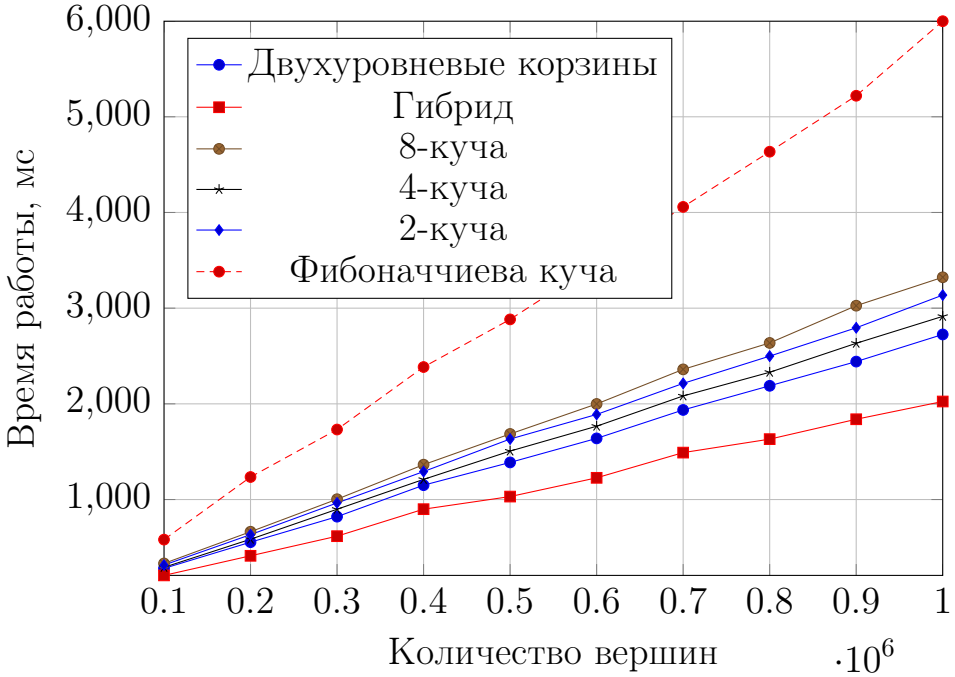

Рис. 4.8. График сравнения времени работы алгоритма Дейкстры при использовании приоритетных очередей на графах с $m \approx 20n$ вершинами и весами ребер в пределах 10000-1000000

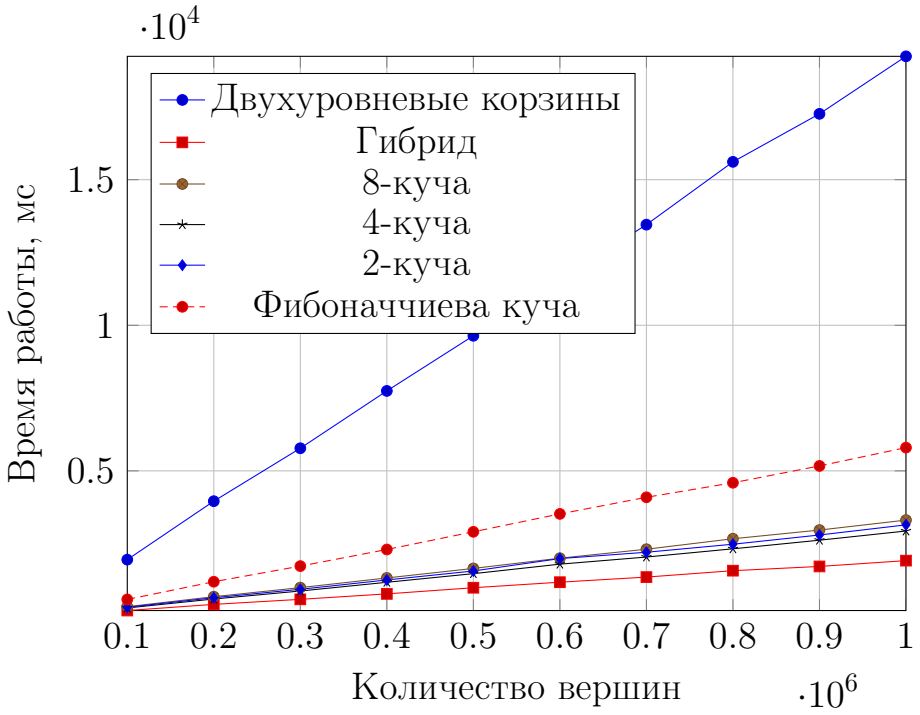

Рис. 4.9. График сравнения времени работы алгоритма Дейкстры при использовании приоритетных очередей на графах с $m \approx 20n$ вершинами и весами ребер в пределах 1000000-100000000

Проанализируем результаты двух серий испытаний, которые проводились на графах с $O(n)$ ребрами. Из всех графиков видно, что 8-кучи, 4-кучи, 2-кучи, фибоначчиевы кучи и гибрид (при весе ребер из диапазона 1-100000000) устойчивы к изменению веса ребер.

Из 8, 4, 2-куч, по данным наших тестов, лучшей является 4-куча, затем идет 2-куча, работающая примерно на 5-7% дольше, чем 4-куча. За ней следует 8-куча, работающая на 12-14% дольше, чем 4-куча. Из результатов испытаний, становится понятно, что использование фибоначчиевых куч с наилучшей асимптотикой для алгоритма Дейкстры $\mathcal{O}(n \log n + m)$ на практике с разреженными графами нецелесообразно. Фибоначчиевы кучи работают в 2 раза дольше чем 4-кучи.

Из (рис. 4.4) и (рис. 4.7) видно, что наилучшей структурой данных при весе ребер от 1 до 10000 являются двухуровневые корзины. Немного им уступает гибрид, работая примерно на 4% дольше. Отрыв двухуровневых корзин от 4-кучи составляет 22-37%.

Из (рис. 4.5) и (рис. 4.8) видно, что наилучшей структурой данных при весе ребер от 10000 до 1000000 является гибрид, двухуровневые корзины же всё еще лучше d-куч. Гибрид работает быстрее на 38-43% чем 4-куча.

Из (рис. 4.6) и (рис. 4.9) видно, что наилучшей структурой данных при весе ребер от 1000000 до 100000000 все также является гибрид, который работает на 27-53% быстрее, чем 4-куча. Разница во времени существенно ощущается с ростом количества вершин. Двухуровневые корзины в свою очередь сильно деградируют, и на (рис. 4.9) сильно уступают даже фибоначчиевой куче. С увеличением количества вершин этот отрыв становится всё больше и больше, и на 1 млн вершин двухуровневые корзины работают в 3,31 раза дольше, чем фибоначчиева куча, которая как нами выше установлено далеко не лучший вариант на практике.

4.3.2. Сравнение для графов с количеством ребер $O(n^2)$

Для графов с количеством ребер $O(n^2)$ была проведена одна серия испытаний. В ней измерялось время работы алгоритма Дейкстры с приоритетными очередями на полных графах. Для рассматриваемых графов количество вершин варьировалось от 1250 до 7500. Аналогично разд. 4.3.1 было исследовано время работы алгоритма при весе ребер в диапазонах: 1-10000, 10000-1000000, 1000000-100000000. На основании этой серии испытаний были построены графики (рис. 4.10), (рис. 4.11), (рис. 4.12).

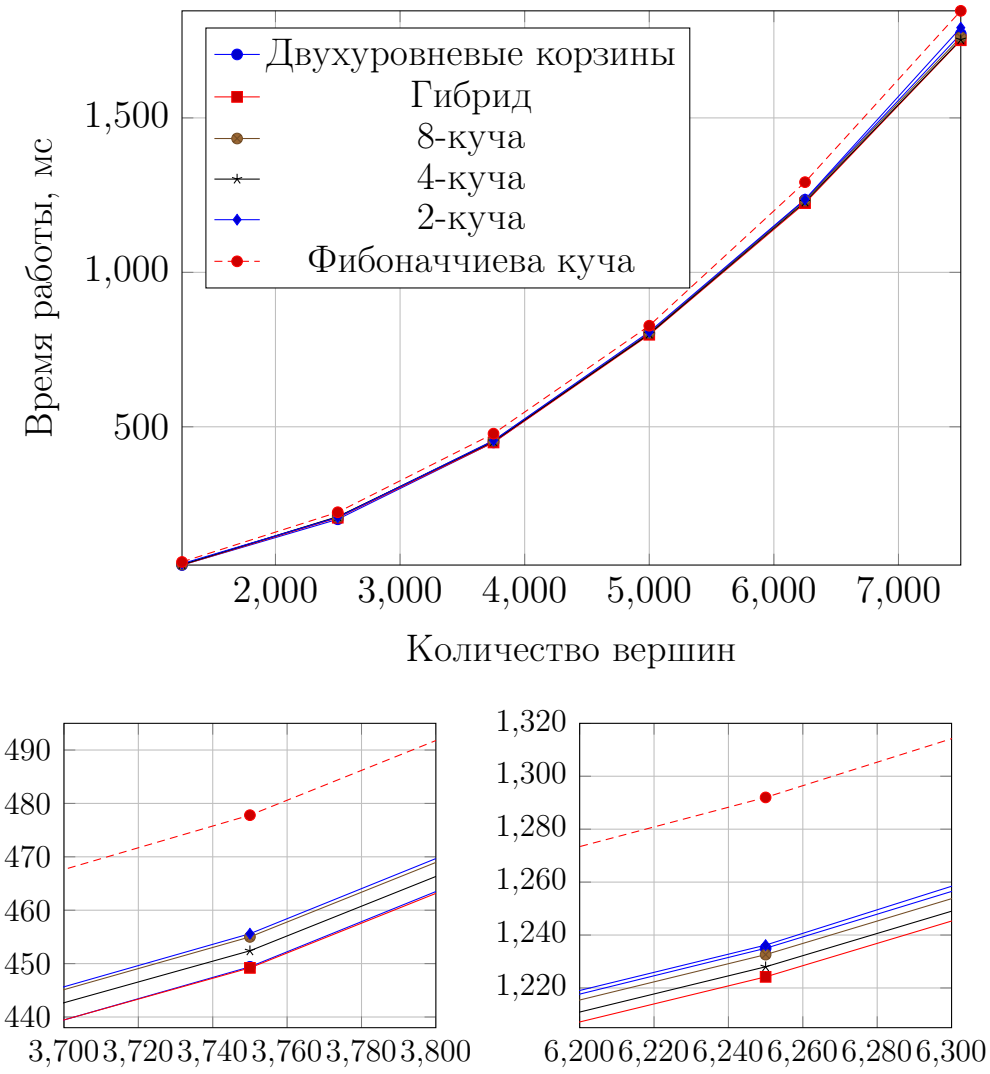

Рис. 4.10. Графики сравнения времени работы алгоритма Дейкстры при использовании приоритетных очередей на полных графах с весами ребер в пределах 1-10000

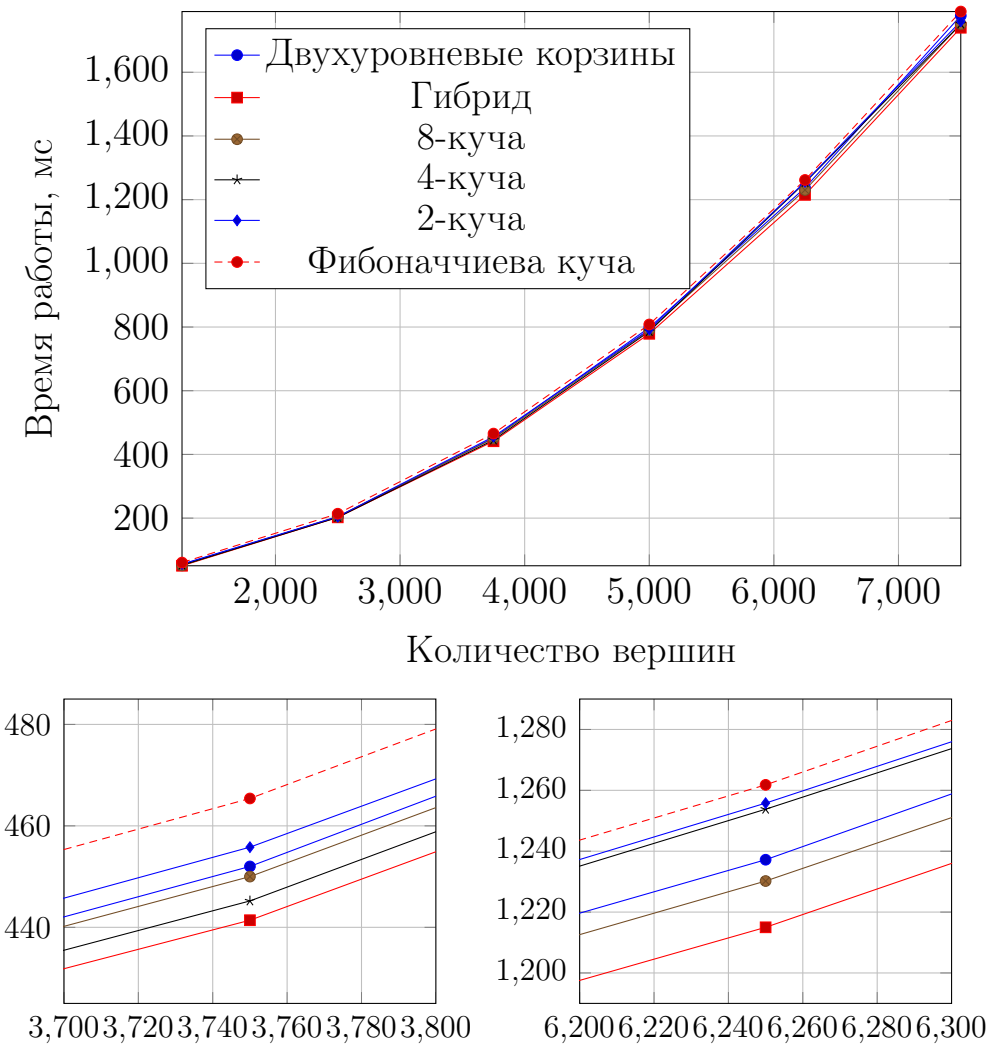

Рис. 4.11. График сравнения времени работы алгоритма Дейкстры при использовании приоритетных очередей на полных графах с весами ребер в пределах 10000-1000000

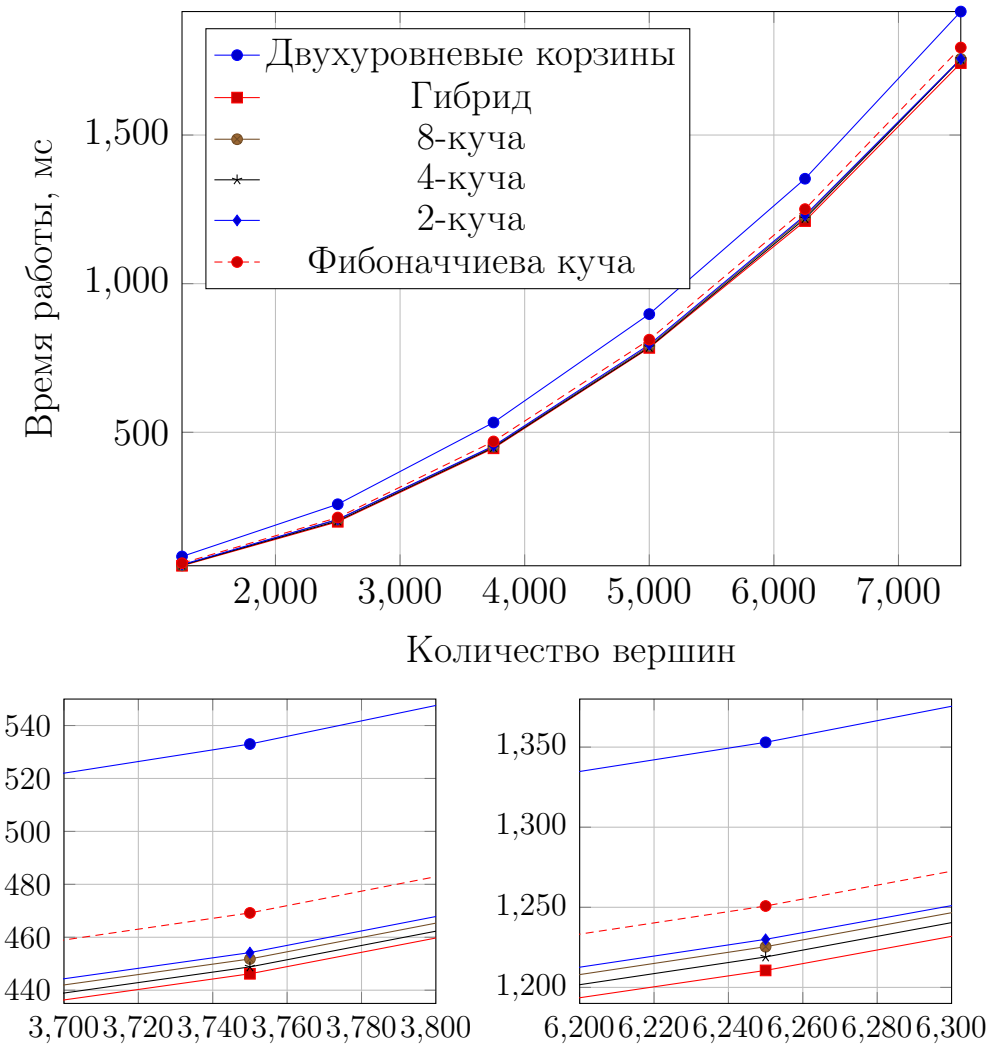

Рис. 4.12. График сравнения времени работы алгоритма Дейкстры при использовании приоритетных очередей на полных графах с весами ребер в пределах 1000000-100000000

Как видно из (рис. 4.10), в случае, когда веса ребер находятся в диапазоне 1-10000, все модификации имеют относительно схожее время работы. Лучшее время работы показывает модификация, основанная на гибриде. За гибридом следует модификация с 4-кучей, которая работает на 1% дольше гибрида, затем идут модификации с 2-кучей и 8-кучей, которые на 2% дольше гибрида. Интересное поведение показывают двухуровневые корзины. Например, при 3750 вершинах эта модификация имеет время работы схожее с гибридом, а уже при 6250 вершинах время работы двухуровневых корзин схоже с 2-кучей. Модификация с фибоначчиевой кучей работает в среднем на 8% дольше гибрида.

Анализируя случай, когда веса ребер из диапазона 10000-1000000 (рис. 4.11), можно заметить ту же тенденцию что и на (рис. 4.10). Лучшей модификацией все также остается гибрид, при этом остальные модификации работают на 1-7% дольше.

Из (рис. 4.12) следует, что при весе ребер из диапазона 1000000-100000000 модификация с двухуровневыми корзинами, как и в разд. 4.3.1 деградирует, и работает примерно на 25% дольше других модификаций с приоритетными очередями. Однако, остальные приоритетные очереди в этих условиях сохраняют ту же тенденцию, что и при весе ребер в диапазонах 1-10000 и 10000-1000000.

Заметим, что на полных графах хорошо раскрывается фибоначчиева куча, которая, как мы знаем из разд. 4.3.1, в испытаниях на графах с $O(n)$ ребрами работала в среднем в 2 раза дольше чем 4-куча. В испытаниях, проведенных на графах с количеством ребер $O(n^2)$, фибоначчиева куча работала всего лишь на 6-8% дольше гибрида, который как нами было установлено выше является лучшей модификацией в этой серии испытаний.

Таким образом, из результатов наблюдений, становится ясно, что для алгоритма Дейкстры на полных графах можно выбрать любую вышеупомянутую приоритетную очередь, кроме двухуровневых корзин в описанных выше условиях.

Заключение

В данной работе был изложен алгоритм Дейкстры, а также основные его модификации, позволяющие сократить время работы алгоритма. Были подробно даны описания и алгоритмы построения приоритетных очередей, таких как d-кучи, фибоначчиева куча, двухуровневые корзины. На основе двухуровневых корзин и 4-кучи была предложена приоритетная очередь - гибрид двухуровневых корзин и 4-кучи (далее - гибрид), с которой алгоритм Дейкстры работает за $\mathcal{O}(n \log n + n\sqrt{C+1} + m \log n)$, где n - количество вершин, m - количество ребер, C - максимальный вес ребра в графе. Все алгоритмы, представленные выше, были реализованы на языке программирования C++. Для этих алгоритмов было проведено тестирование. Результатом тестирования и теоретических рассуждений являются следующие рекомендации к использованию алгоритма Дейкстры: при реализации лучше использовать списки смежности, чем матрицу расстояний, также лучше реализовывать алгоритм Дейкстры с приоритетными очередями и применять завершение алгоритма Дейкстры, если финишная вершина достигнута.

Если алгоритм Дейкстры используется на графах с $O(n)$ целочисленными ребрами маленького веса (1-10000), то целесообразно использовать модификацию построенную на двухуровневых корзинах. В остальных же случаях, то есть при весе ребер в диапазоне (10000-100000000) или при ребрах не целочисленного веса, для графов с количеством ребер $O(n)$ предпочтительнее использовать модификацию построенную на гибриде, которая, по результатам тестирования, работает на 27-53% быстрее чем модификация с 4-кучей. Отметим, что 4-куча является, по результатам наших наблюдений, самой быстрой структурой не зависящей от параметра C. Однако, можно также использовать гибрид и на графах с целочисленными ребрами маленького веса (1-10000). Проигрыш по времени относительно двухуровневых корзин будет составлять приблизительно 4% времени работы модификации с двухуровневыми корзинами.

Если алгоритм Дейкстры используется на графах с $O(n^2)$ ребрами, то для реализации можно выбрать практически любую из вышеописанных при-

оритетных очередей. Неэффективность относительно других модификаций, в нашей серии испытаний, показала лишь модификация с двухуровневыми корзинами, при весе ребер в диапазоне (1000000-100000000).

Таким образом, предложенная в данной работе, структура данных - гибрид двухуровневых корзин и 4-кучи успешно себя зарекомендовала при тестировании как на графах с количеством ребер $O(n)$, так и на графах с количеством ребер $O(n^2)$.

Список использованной литературы

1. *Cormen T. H.*, *Leiserson C. E.*, *Rivest R. L.*, *Stein C.* Introduction to Algorithms. — MIT Press, 2009. — С. 505—526.
2. *Goldberg A. V.*, *Silverstein C.* Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets // Network Optimization. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1997. — С. 292—327.
3. *Dijkstra E. W.* A note on two problems in connexion with graphs // Journal of Chemical Information and Modeling. — 1959. — Т. 55. — С. 269—271.
4. *Алексеев В. Е.*, *Таланов В. А.* Графы и алгоритмы : учебное пособие. — Москва : ИНТУИТ, 2016. — С. 170—183.

Приложение

https://github.com/antonov1lya/CoursePaper