Санкт-Петербургский государственный университет

Овечкин Григорий Владимирович

Выпускная квалификационная работа

Хэширование кукушки в телекоммуникационных сетях

Уровень образования: магистратура
Направление и код: 01.04.01 «Математика»
Основная образовательная программа ВМ.5832.2022 «Современная математика»

Научный руководитель:
преподаватель, факультет
математики и компьютерных
наук СПбГУ,
доктор ф.-м. наук, профессор,
Петров Федор Владимирович

Рецензент:
инженер ключевых проектов,
ООО «Техкомпания Хуавэй»,
доктор ф.-м. наук,
Максименко Александр Николаевич

Санкт-Петербург
2024 год

Содержание

1 Введение

Хеширование кукушки — одна из наиболее практичных структур данных. Она имеет функционал словаря: постоянное время вставки в среднем, постоянное время удаления и время поиска в худшем случае. Одним из других важных свойств является высокий коэффициент полезного использования памяти. Оригинальный алгоритм вставляет элемент в течение $\Omega(\log n)$ времени в худшем случае, что не соответствует требованиям в некоторых практических задачах. Однако существует несколько так называемых деаморти-зированных версий, которые сокращают время вставки в наихудшем случае до $O(1)$, немного увеличивая время выполнения двух других операций. Еще одной полезной модификацией кукушкиного хеширования является фильтр кукушки — динамическая структура данных, позволяющая отвечать на запросы принадлежности элемента множеству. Фильтр кукушки наследует все преимущества классического кукушкиного хеширования. Он эффективнее по памяти, чем фильтр Блума, и, кроме того, в отличие от фильтра Блума, может работать не только как множество ключей, но и как словарь, т.е. может хранить значения.

В этой работе мы проводим обзор известных теоретических результатов, связанных с кукушкиным хешированием, реализуем несколько алгоритмов, адаптированных к различным задачам, и сравниваем производительность кукушкиного хеширования и других алгоритмов в различных сценариях, связанных с маршрутизацией в телекоммуникационных сетях.

1.1 Динамические словари

Как уже было сказано, хеширование кукушки принадлежит классу структур данных, называемом динамическими словарями. Динамический словарь - это одна из фундаментальных структур данных, которая хранит пары ключ-значение и поддерживает три основные операции: вставку, удаление и поиск. Точное определение можно найти, например, здесь [2]. В телекоммуникациях динамические словари являются одним из самых базовых алгоритмов и составляют важную часть функционала таких устройств, как маршрутизаторы и коммутаторы. Современные магистральные сети обеспечивают скорость, измеряемую в Тбит/с, что означает, что сетевые устройства должны обрабатывать миллиарды пакетов в секунду. В то же время таблицы с информацией о пересылке достаточно велики для хранения в сверхбыстрой кэш-памяти. Все это предъявляет серьезные требования к алгоритмам, используемым в сетевых устройствах. Поиск адреса пакета в базе данных устройства должен выполняться за несколько тактов процессора и выполнять только одно обращение к памяти.

Методы, основанные на хешировании, и, в частности, кукушкины хеш-таблицы, часто используются для реализации таких баз и выполнения быстрых операций поиска. Мы рассмотрим различные алгоритмы и проведем моделирование, которое показывает преимущества кукушкиного хеширования, а также сформулируем несколько хорошо известных теоретических результатов, которые показывают пределы применения кукушкиного хеширования.

В статьях по кукушкиному хешированию авторы обычно изучают только этапы вставки, так называемый статический случай. Типичные вопросы, которые они поднимают, таковы: сколько случайных ключей может быть вставлено до первого сбоя или какова вероятность вставки n случайных ключей в таблицу с m

записями. На практике структуры данных являются динамическими объектами, и мы не должны ограничивать наше изучение только статическим случаем. В нескольких статьях было предложено рассмотреть все последовательности запросов ограниченной длины (динамический случай) и доказать свойства, которые выполняются с высокой вероятностью. Моделирование, особенно в динамическом случае, не может охватить какое-либо достаточно большое подмножество последовательностей запросов. Однако оно позволяет нам сравнивать разные алгоритмы или один и тот же алгоритм с разными параметрами.

Основой для нашей модели вычислений является целочисленная RAM. Но целые числа, хранимые в ячейках, которые можно прочитать/записать за одно обращение к памяти ограничены некоторой константой. Более того, в отличие от классических результатов с оценками в терминах O-больших нас больше будут интересовать конкретные константы в главных членах асимптотик.

2 Хеширование кукушки: математическое описание

Начнем с общего описания задачи о словаре. Пусть $\mathcal{K}$ — множество всевозможных ключей, $\mathcal{V}$ — множество всевозможных значений. Записью будем называть пару $r = (k, v)$ из ключа и значения. Словарь H представляет собой черный ящик с тремя операциями:

- Операция Insert(H, k, v) должна вставлять запись (k, v) в словарь H,
- Операция Delete(H, k) должна удалять запись, содержащую ключ k в своем первом поле, из словаря H,
- Операция Lookup(H, k) должна возвращать значение, соответствующее ключу k, если k был вставлен ранее и не был удален после последней вставки. В противном случае он должен возвращать нулевое значение, которое эквивалентно сообщению о том, что такого ключа в таблице нет.

Далее мы будем рассматривать только хеш-таблицы, как представителей класса словарей. Обычно в хеш-таблицах используются хеш-функции, которые являются функциями из множества $\mathcal{K}$ в множество $[m] = \{0, 1, \ldots, m-1\}$ для некоторого числа $m \in \mathbb{N}$. Обозначим это множество всех таких функций для данных $\mathcal{K}$ и m через $\mathcal{H}^m_{\mathcal{K}}$. Хеш-функции сами по себе являются чрезвычайно интересными объектами, однако они лежат за пределами области нашего исследования. Далее мы уточним некоторые важные свойства хеш-функций, которые будем использовать.

Зафиксируем множество ключей $\mathcal{K}$ и множество значений $\mathcal{V}$. Кукушкина хеш-таблица (или хеш-таблица кукушки) будет хранить пары ключ-значение из множества $\mathcal{K} \times \mathcal{V}$. Таким образом, определим кукушкину хеш-таблицу T как кортеж $(m, l, d, s, \{h_i\}_{i=1}^d, \{T_i\}_{i=1}^d, A_I, A_D, A_L)$, где $m, l, d, s \in \mathbb{N}$, $h_i \in \mathcal{H}^m_{\mathcal{K}}$ для $1 \le i \le d$, и каждая таблица T_i есть массив размера $m \times l$, ячейки которого либо содержат одну запись, либо пусты. Будем говорить, что таблицы имеют m строк длины l. Число s обозначает максимальное число итераций во внешнем цикле вставки. Алгоритмы поиска A_L, вставки A_I и удаления A_D приведены ниже (1, 2, 3, соответственно). Внимание: сообщение о переполненности кукушкиной хеш-таблицы не означает, что все ячейки всех ее таблиц заняты.

Нетрудно заметить, что временная сложность функций поиска и удаления не зависит от количества вставленных в таблицу ключей, поэтому их сложность равна $O(1)$. Обычно после неудачной вставки

Algorithm 1 Table Lookup

Require: Таблица T и ключ k.

Ensure: Возвращает соответствующее значение, если ключ k содержится в таблице T, или $\perp$ (ничего) в другом случае.

```
a_i ← h_i(k) for 1 ≤ i ≤ d
i ← 0
while i < d do
    j ← 0
    while j < l do
        if T_i[a_i][j] is not empty and k == T_i[a_i][j].key then
            return T_i[a_i][j].value
        end if
    end while
end while
return ⊥
```

Algorithm 2 Table Insert

Require: Таблица T, ключ k и значение v.

Ensure: Вставляет запись (k, v) или возвращает сообщение о переполненности таблицы.

```
q ← 0                                              ▷ номер итерации вставки
f ← −1                                             ▷ запрещенный индекс таблицы
while q < s do
    a_i ← h_i(k) for 1 ≤ i ≤ d
    i ← 0
    while i < d do
        j ← 0
        while j < l do
            if T_i[a_i][j] is empty then
                T_i[a_i][j] ← (k, v)
                return inserted                    ▷ вставка прошла успешно
            end if
        end while
    end while
    p ← choose([d]\{f})    ▷ выбрать номер следующий таблицы согласно некоторому правилу
    t ← choose([l])        ▷ выбрать в запись внутри строки согласно некоторому правилу
    k' ← T_s[a_s][t]
    T_p[a_p][t] ← k
    k ← k'
end while
return overfull                                    ▷ таблица переполнена
```

Algorithm 3 Table Delete

Require: Таблица T и ключ k.

Ensure: Удаляет соответствующую запись, если ключ k представлен в таблице T, или возвращает сообщение об отсутствии данного ключа.

```
a_i ← h_i(k) for 1 ≤ i ≤ d
i ← 0
while i < d do
    j ← 0
    while j < l do
        if T_i[a_i][j] is not empty and k == T_i[a_i][j].key then
            T_i[a_i][j] ← empty
            return deleted                          ▷ соответствующая запись удалена
        end if
    end while
end while
return not matched                                  ▷ ключ в таблице отсутствует
```

следует перехеширование всей таблицы с использованием новых хеш-функций. Но это операция заведомо требует $O(n)$ операций, что непозволительно много в нашей ситуации.

Таким образом, остается две взаимосвязанные проблемы, которые требуют решения. Первая проблема — это, возможно, слишком большое количество итераций во внешнем цикле, и вторая — это высокая вероятность отказа, (невозможность вставки) алгоритма. Решению этих вопросов было посвящено множество работ, обзор которых приведен в следующей главе.

3 Краткая классификация различных версий хеширования кукушки

Существует множество различных версий алгоритмов хеширования кукушки. Алгоритмы могут иметь:

1. разное количество d таблиц и соответствующих им хеш-функции (в классической версии $d = 2$);
2. разные алгоритмы вставки (два распространенных подхода — поиск в ширину и метод случайного блуждания);
3. разное число шагов вставки s;
4. дополнительные структуры для хранения записей (например, тайник или очередь);
5. разные варианты хеш-функций (некоторые из них чисто теоретические).

Также различные версии могут различаться по области значений хеш-функций: в одних случаях каждая хеш-функция бьет в свою таблицу, в других — в одну общую таблицу. Описание выше придерживается

первого подхода так как он более удобен с точки зрения реализации алгоритма, но некоторые авторы предпочитают использовать второй. Разница между двумя подходами не столь существенна и ее анализ не входит в область наших интересов в данной работе. Перейдем к конкретным примерам кукушкиных хеш-таблиц. Во всех примерах, если не сказано иное, будем считать, что значения хеш-функций $h_1(k), h_2(k), \ldots, h_d(k)$ распределены равномерно и независимо.

3.1 Классическая версия

Р. Пах и Ф. Ф. Родлер ввели понятие хеширования кукушки и доказали некоторые его свойства в работе [1]. Первая версия алгоритма имеет $d = 2$ таблицы размера $\lceil (1+\varepsilon)n \rceil \times 1$, а для доказательства оценки временной сложности вставки хеш-функции достаточно выбирать из 2-универсального семейства (см. параграф 5). Вставка в данной версии выполняется за $O(log(n))$ элементарных операций. В случае $d = 2$ таблиц единственная неоднозначность в алгоритме вставки имеется только в самом начале, можно зафиксировать одну из таблиц и всегда сначала идти в нее. Проблема перехеширования здесь упоминается, но авторы используют амортизированный анализ временной сложности алгоритма вставки, который в нашем случае недопустим. В телекоммуникационных сетях наиболее важным является то, сколько времени операция выполняется в худшем случае, а не в среднем.

3.2 Хеширование кукушки с большим количеством таблиц d

Классическая версия кроме описанных выше проблем имеет еще один существенный недостаток, для хранения n записей она требует $2(1+\varepsilon)n$ ячеек, т.е. коэффициент использования памяти чуть меньше 50%. Простейший способ решить эту проблему — увеличить длину строк. Но слишком длинные строки нельзя считывать за одну операцию. Например, если в слово помещается 128 бит информации, а ключ и значения суть 32-битные целые числа, то нет смысла делать строки длины больше 2. А вот увеличивать число хеш-функций d и, соответственно, число возможных позиций для вставки каждого ключа можно гораздо свободнее.

В работе [4] Д. Фотакис, Р. Пах, П. Сандерс и П. Спиракис доказали, что $d = O(\log(1/\varepsilon))$ хеш-функций достаточно для хранения n записей в $(1+\varepsilon)n$ ячейках. В таком случае, время поиска будет константным в худшем случае, а время вставки будет константным в среднем. В этой работе используется алгоритм вставки на основе метода поиска в ширину. Авторы доказали следующую теорему.

Теорема 4.1 *Для каждого положительного числа $\varepsilon < 1/5$ и целого числа $d \geq 5 + 3\log(1/\varepsilon)$ алгоритм вставки требует $(1/\varepsilon)^{O(\ln d)}$ операций в среднем. Более того, алгоритм выполняет не более $o(n)$ шагов, прежде чем найдет свободную ячейку с вероятностью $1 - O(1/n)$.*

Похожий результат доказали Н. Фунтулакис, К. Панайоту и А. Штегер в работе [3] для вставки случайным блужданием. Их оценка на используемую память близка к теоретическому пределу. О нем и о константе c_d^* будет сказано в параграфе 4. В этой версии хеширования кукушки используются единая таблица размера $m \times 1$.

Теорема 4.2 *Для $d \geq 3$ положим $c(d) = \frac{d+\log(d-1)}{(d-1)\log(d-1)}$. Тогда для любого числа $\zeta > 0$ и числа $\varepsilon \in (0,1)$ справедливо следующее. Для любого числа $d \geq 3$ и набора из $n = \lfloor (1-\varepsilon)c_d^* m \rfloor$ ключей с вероятностью*

$1 - o(1)$ *каждый из элементов будет вставлен в таблицу за время* $O(\log^{2+c+\zeta} n)$.

3.3 Хеш-таблица с тайником

Наиболее естественный алгоритм вставки состоит в том, чтобы наложить априорное ограничение в размере $\alpha \log n$ на количество удалений, и если после $\alpha \log n$ итераций не было найдено пустой ячейки, записать текущий элемент в специальное место, называемое тайником. А. Кирш, М. Митценмахер и У. Видер (см. [10]) показали, что использование тайника даже размера $O(1)$ сильно уменьшает вероятность перехеширования всей таблицы. Для случая $d = 2$ и большего числа таблиц верны следующие теоремы.

Теорема 4.3 *Для каждого целого числа* $s \geq 1$ *существует константа* $\alpha > 0$ *такая, что размер тайника* S *после вставки* n *элементов удовлетворяет соотношению* $\mathbb{P}(S \geq s) = O(n^{-s})$.

Теорема 4.4 *Для любых чисел* $c > 0$ *и* $\varepsilon \in (0, 0.2)$ *существует целое число* d *такое, что для каждого целого числа* $s \geq 1$ *верно соотношение* $\mathbb{P}\{S \geq s\} = O(n^{1-cs})$ *при* $n \to \infty$. *Более того,* d *можно выбрать среди целых чисел не больших* $3\log(1/\varepsilon) + O(1)$, *где асимптотика берется при* $\varepsilon \to 0$.

3.4 Деамортизированная версия

Амортизированный анализ, применяемый к кукушкиному хешированию, работает не во всех ситуациях. Как уже было сказано в нашем случае производительность в наихудшем случае имеет решающее значение. А. Кирш и М. Митценмахер в статье [9] предложили применить деамортизацию к кукушкиному хешированию с использованием методов на основе очередей. Ю.Арбитман, М.Наор и Г.Сегев в своей работе используют подход Кирша и Митценмахера и совершенствуют его, доказывая некоторые оценки для наихудшего случая. Псевдокод новой операции вставки приведен ниже (см. 4).

Пусть $\varepsilon > 0$ и $L \in \mathbb{Z}_{\leq 0}$. Будем рассматривать кукушкино хеширование с 2 таблицами T_0 и T_1, состоящих из $m = \lceil (1 + \varepsilon)n \rceil$ одноэлементных строк, и очереди с максимальным размером L. Хеш-функции будут выбираться случайно и независимо из $p(n)$-независимого семейства хеш-функций, где p — некоторый полином.

Определение. Последовательность запросов называется *n-ограниченной*, если в любой момент времени структура данных содержит не более n записей.

Теорема 4.5 *Для любого многочлена* $p(n)$ *и числа* $\varepsilon > 0$ *существует число* L *такое, что для кукушкиного хеширования с параметрами* ε *и* L *выполняется следующее. Для любой* n*-ограниченной последовательности запросов с вероятностью* $1 - 1/p(n)$ *ни одна вставка не закончится переполнением.*

4 Теоретические оценки снизу на использование памяти

Проблему вставки элементов с помощью алгоритма кукушки можно переформулировать в терминах ориентируемости гиперграфов. И этот вопрос изучался в работах разных авторов (см. [7], [14], [15], [16], [17]). Рассмотрим семейство d-однородных (каждое ребро инцидентно k вершинам) гиперграфов на m вершинах. Гиперграф из этого семейства является *l-ориентируемым*, если для каждого ребра мы можем выбрать одну из его вершин, так что каждая вершина выбирается в общей сложности не более l раз. Если

Algorithm 4 Queued Table Insert

Require: Таблица T, ключ k и значение v.

Ensure: Вставляет запись (k, v) или возвращает сообщение о переполненности таблицы.

```
q ← 0                                                  ▷ номер итерации вставки
f ← −1                                                 ▷ запрещенный индекс таблицы
while q < s do
    a_i ← h_i(k) for 1 ≤ i ≤ d
    i ← 0
    while i < d do
        j ← 0
        while j < l do
            if T_i[a_i][j] is empty then
                T_i[a_i][j] ← (k, v)
                return inserted                        ▷ вставка прошла успешно
            end if
        end while
    end while
    p ← choose([d]\{f})        ▷ выбрать номер следующий таблицы согласно некоторому правилу
    t ← choose([l])            ▷ выбрать в запись внутри строки согласно некоторому правилу
    k′ ← T_s[a_s][t]
    T_p[a_p][t] ← k
    T.queue.pushback(k′)
    k ← T.queue.front
    T.queue.popfront
end while
return overfull                                        ▷ таблица переполнена
```

$d = 2$ и $l = 1$, то мы имеем обычное представление об ориентируемости графов. Рассмотрим случайный k-однородный гиперграф $H_{m,n,d}$ с m вершинами и n d ребрами, равномерно распределенный на множестве всех k-однородных гиперграфов с m вершинами и n ребрами. Тогда верна следующая теорема.

Теорема 5.1 *Для любых $d \geq 2$ и $l \geq 1$ существует константа $c^*_{d,l}$ такая, что при $m \to \infty$*

$$\mathbb{P}(H_{md,\lfloor cmd \rfloor,d} \text{ — } l\text{-ориентируем }) \to \begin{cases} 0, & \text{если } c > c^*_{d,l} \\ 1, & \text{если } c < c^*_{d,l} \end{cases}.$$

Remark. Константа $c^*_{d,l}$ может быть вычислена явно. Пусть ξ^* — это единственное решение уравнения

$$kl = \frac{\xi^* Q(\xi^*, l)}{Q(\xi^*, l+1)}, \quad \text{где } Q(x, y) = 1 - e^{-x} \sum_{j<y} \frac{x^j}{j!}.$$

Тогда

$$c^*_{d,l} = \frac{\xi^*}{kQ(\xi^*, l)^{k-1}}.$$

Remark. В некоторых случаях существует эффективный алгоритм для нахождения этой ориентации.

$c^*_{2,1} = 2$

5 Семейства хеш-функций

Хеш-функции являются важной частью алгоритмов работы с хеш-таблицами. Чем более случайным является поведение хеш-функций, тем более сильные свойства алгоритмов работы с хеш-таблицами могут быть доказаны.

Существует несколько определений случайности поведения хеш-функций. Семейство $\mathcal{H}$ называется (μ, k)-универсальным, если для любых k различных ключей $x_1, \ldots, x_n \in \mathcal{K}$ и k значений $y_1, \ldots, y_k \in [m]$ при выборе хеш-функции h из $\mathcal{H}$ случайным равномерным образом выполняется неравенство

$$\mathbb{P}(h(x_1) = y_1, \ldots, h(x_k) = Y_k) \leq \frac{\mu}{m^d}.$$

Если $\mu = 1$, то все неравенства становятся равенствами и мы будем называть такое семейство $\mathcal{H}$ k-независимым.

В данной работе мы будем использовать полиномиальное хеширование. Пусть k целое положительное число. Рассмотрим семейство многочленов над простым полем $\mathbb{F}_p$ (для некоторого большого простого числа p) степени k. Выберем случайный многочлен. Чтобы захешировать ключ, мы должны представить его в виде остатка по модулю p и вычислить многочлен в данной точке. Такое хеширование позволяет генерировать хеш-функции из k-независимого семейства для любого k.

6 Описание алгоритмов и тестовых сценариев

Алгоритм хеширования кукушки реализован на языке программирования C++ и протестирован в различных сценариях. Во всех сценариях тестирование не прекращается после первого переполнения, поскольку нас интересует, какая доля всех записей хранится в таблице и какая точность ответов на запросы поиска.

6.1 Список тестируемых алгоритмов

В этом разделе мы более подробно рассмотрим различия в версиях реализованных алгоритмов.

Хеширование кукушки. Базовая версия кукушкиной хеш-таблицы состоит из d хеш-таблиц, содержащих m строк по l записей в каждой. Функция вставки ограничена s шагами перехода. Если для одной записи хеш-таблицы требуется M бит, то пространственная сложность хеширования кукушки равна $d \cdot k \cdot l \cdot M$ бит, временная сложность функций поиска и удаления равна $O(d)$, а временная сложность вставки равна $O(d \cdot s)$.

Хеширование кукушки с тайником. Надстройку в виде тайника очень легко добавить. Она может быть реализована в различных формах. Их можно разделить на две большие части: программные и аппаратные. Одной из любопытных реализаций является ассоциативная память (CAM). CAM - это аппаратный вариант ассоциативного контейнера, такого как set или dictionary. Она почти такая же быстрая, как обычная оперативная память, но намного дороже. В нашем алгоритме тайник — это дополнительный список, который имитирует CAM. Единственный параметр, который нас интересует, — его длина. Тайник вступает в игру, когда все обычные шаги по вставке израсходованы, и на последнем шаге есть выбитый элемент. Тогда этот элемент вставляется в тайник. Для корректной обработки элементов из тайника необходимо обновить функции поиска и удаления. Вообще, тайник может быть добавлен в качестве постфикса к любой структуре данных, в которую не все элементы могут быть вставлены за определенное количество итераций. Обычно размер тайника составляет не более одного процента от размера всей таблицы.

Хеширование кукушки с очередью. В отличие от тайника, очередь — это префиксная структура. Она играет роль буфера. Все выбитые элементы вставляются в очередь, а затем в порядке FIFO они вставляются в основную таблицу таким образом, чтобы общее количество шагов было меньше или равно s. Вновь прибывшие элементы вставляются сразу в таблицу, минуя очередь. Это дополнение может быть полезно в таких структурах данных, где количество шагов при вставке может значительно варьироваться. Как и тайник, очередь может быть реализована аппаратно, но это еще сложнее. Однако, чисто программная реализация очереди также интересна. Один из вопросов заключается в том, может ли очень маленькая очередь (например, постоянного размера) значительно улучшить базовый алгоритм. Частично ответ был дан в подпараграфе 3.4.

Алгоритм основанный на d-левом хешировании. Следующий алгоритм не принадлежит семейству кукушкиных хешований. Он рассматривается в качестве некоторой точки отсчета и нужен для того, чтобы показать сильные стороны хеширования кукушки. Одной из модификаций классической схемы хеш-таблиц с использованием списков было хеширование по принципу "разрешать ничьи влево"(см. [13] и [12]). Вкратце, предположим, что хеш-таблицы расположены одна за другой слева направо. Для каждого нового элемента имеется несколько строк в разных таблицах, куда его можно поместить. Из этих строк надо выбрать наименее заполненную, а если таких несколько, то самую левую, и вставить новый элемент в нее. Это уменьшит длину самой большой цепочки с $\log n / \log \log n(1 + o(1))$ по сравнению до

$\log\log n/(d\Phi_d) + O(1)$. Мы будем использовать модифицированную версию этого алгоритма. В одну строку каждой таблицы не может быть вставлено больше, чем l элементов, и если все строки для некоторого ключа полностью заполнены, то алгоритм вставки возвращает сообщение об ошибке. Псевдокод функции вставки приведен ниже в алгоритме 5.

Algorithm 5 d-Left Table Insert

Require: Таблица T, ключ k и значение v.

Ensure: Вставляет запись (k, v) или возвращает сообщение о переполненности.

```
a_i ← h_i(k) for 1 ≤ i ≤ d
if T_i[a_i] is full for all i then
    return overfull                          ▷ все возможные строки заполнены
end if
i ← choose(1, ..., n)                       ▷ выбираем согласно правилу описанному выше
T_i[a_i].pushback(k, v)
return inserted
```

6.2 Список тестовых сценариев

Всего есть два тестовых сценария: статический и динамический. В обоих сценариях используется алгоритм генерации новых ключей для вставки в таблицу. Мы используем два алгоритма генерации ключей: случайный и последовательный. В первом способе каждый новый ключ это псевдослучайное число (ПСЧ), полученное с использованием встроенного генератора ПСЧ. Во втором способе только первый ключ получен с использованием генератора ПСЧ, а каждый следующий ключ получается из предыдущего увеличением на один. В каждом из сценариев мы будем сравнивать обычное хеширование кукушки, деамортизированный вариант с использованием очереди и d-левое хеширование. В каждом тестовом примере все хеш-таблицы будут иметь одинаковый размер.

Статический сценарий. В статическом сценарии в хеш-таблицу вставляется n ключей, а затем подается n запросов поиска с этими же ключами. Способ генерации ключей указан в таблице параметров в параграфе 7 для каждого теста. Алгоритм тестирования выглядит так:

1. Сгенерировать n ключей;
2. Выбрать хеш-функцию для каждой хеш-таблицы;
3. Запустить алгоритм на тестовом примере;
4. Вычислить заполненность таблицы — отношение числа вставленных элементов к общему числу запросов вставки;
5. Повторить предыдущие шаги N раз;
6. Посчитать распределения заполненности таблицы;

7. Сравнить их между различными алгоритмами.

Динамический сценарий. В динамическом сценарии в хеш-таблицу изначально вставляется n ключей, а затем подается последовательность из $q - n$ запросов. Тип каждого запроса генерируется случайно и независимо с вероятностями 0.9 для поиска и по 0.05 для вставки и удаления. Исключение составляют случаи, когда число вставленных элементов минус число удаленных элементов слишком маленькое или слишком большое. Тогда следующая операция обязательна будет вставкой или удалением, соответственно. Для операции вставки новый ключ генерируется случайно или последовательно в зависимости от тестового примера. Для операции поиска и удаления ключ выбирается случайно равномерно из множества тех ключей, которые должны присутствовать в таблице на данный момент. Фактическое количество вставленных и еще не удаленных ключей есть процесс схожий со случайным блужданием с барьерами. Мы выражаем значение барьера в процентах от загрузки хеш-таблицы. Эти значения указаны в таблице параметров. Алгоритм тестирования в динамическом сценарии выглядит так:

1. Сгенерировать последовательность из q запросов как описано выше;
2. Выбрать хеш-функцию для каждой хеш-таблицы;
3. Запустить алгоритм на тестовом примере;
4. Вычислить как меняется со временем заполненность хеш-таблицы и точность ответов на запросы поиска;
5. Сравнить эти статистики для разных алгоритмов.

7 Компьютерное моделирование

Целью данной работы является сравнение различных подходов к построению кукушкиной хеш-таблицы, а также сравнение их всех с другими известными алгоритмами построения хеш-таблиц в различных сценариях.

В разделе 6 мы описали тестовые сценарии и алгоритмы хеш-таблиц без каких-либо числовых значений. Здесь мы приводим конкретные значения всех параметров для каждого тестового примера. Хеширование кукушки с очередью во всех тестах имеет очередь максимальной длины 10. Также во всех тестах хеш-функции выбирались из семейства полиномиальных хеш-функций степени 4. Во всех динамических тестах нижний барьер числа элементов равен 0.99, а верхний барьер равен 1.

7.1 Результаты

Для статических тестов приведены графики распределения заполненности хеш-таблиц. По оси абсцисс отложена заполненность, а по оси ординат — частота, т.е. количество случаев, когда заполненность хеш-таблицы лежала в определенном интервале ширины 0.005. Для динамических тестов приведены графики зависимости заполненности (верхний график) и точности (нижний график) от времени, т.е. количества

Название теста	N	n	d	m	l	s	генерация ключей
Статический тест 1	1000	10^5	2	2	$1,25 \cdot 10^4$	4	случайная
Статический тест 2	1000	10^5	2	2	$1,25 \cdot 10^4$	4	последовательная
Статический тест 3	100	$2 \cdot 10^5$	4	2	$1,25 \cdot 10^4$	4	случайная
Статический тест 4	100	$2 \cdot 10^5$	4	2	$1,25 \cdot 10^4$	4	последовательная

Рис. 1: Таблица параметров для статических тестовых примеров

Название теста	n	q	d	m	l	s	генерация ключей
Динамический тест 1	10^5	10^7	2	$1,25 \cdot 10^4$	4	4	случайная
Динамический тест 2	10^5	10^7	2	$1,25 \cdot 10^4$	4	4	последовательная
Динамический тест 3	$2 \cdot 10^5$	10^7	4	$1,25 \cdot 10^4$	4	10	случайная
Динамический тест 4	$2 \cdot 10^5$	10^7	4	$1,25 \cdot 10^4$	4	10	последовательная
Динамический тест 5	10^5	10^7	2	$1,25 \cdot 10^4$	4	6	случайная
Динамический тест 6	10^5	10^7	2	$1,25 \cdot 10^4$	4	3	случайная

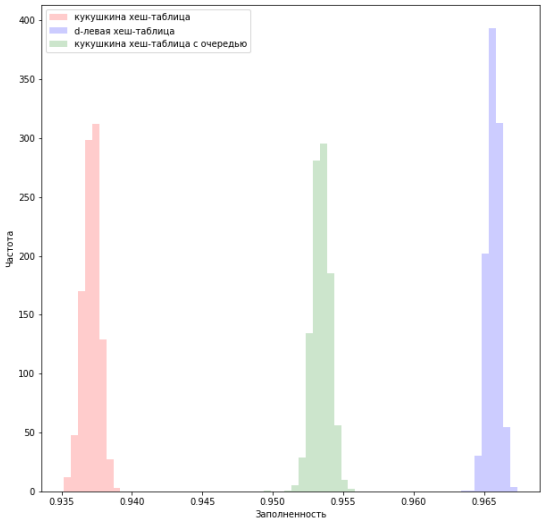
Рис. 2: Таблица параметров для динамических тестовых примеров

уже обработанных запросов. По оси абсцисс отложены порядковый номер вставки или удаления на верхнем графике и порядковый номер запроса поиска на нижнем графике. По оси ординат отложены заполненность (отношение размера исследуемой хеш-таблицы к размеру идеального словаря) и частота на верхнем и нижнем графиках соответственно.

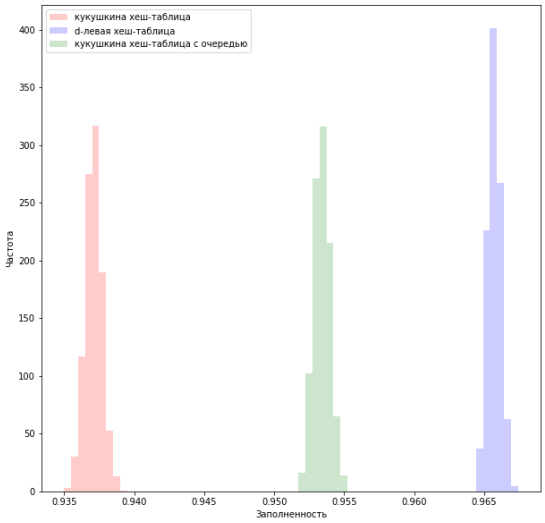
На графиках динамических тестов видно, что кукушкино хеширование ведет себя лучше в динамике, чем d-левое хеширование. Особенно хорошо это видно на графике 6. До начала динамической части последовательности запросов заполненность d-левой хеш-таблицы была больше, а в процессе опустилась ниже заполненности кукушкиной хеш-таблицы. Это наблюдение подтверждает тот факт, что решая практические задачи нельзя фокусироваться только на статическом тестовом сценарии. Некоторые алгоритмы раскрывают весь свой потенциал именно в динамике.

Проведенные тесты подтверждают хорошо известные результаты о кукушкином хешировании. Заполненность таблицы улучшается при увеличении числа итераций и при увеличении числа подтаблиц.

Отдельно стоит отметить поведение хеш-таблиц при последовательной генерации ключей и использования хеш-функций из 2-независимого семейства (см. графики 8). Оказывается, что 2-независимости недостаточно для того, чтобы поведение исследуемых алгоритмов было похожим на их поведения на случайных данных. Как было сказано в начале данной работы вопросы выбора хеш-функций не будут рассматриваться детально. Отметим только, что задача поиска «хорошего» класса хеш-функций исследовалась в работах многих авторов (см. [18] и [19]).

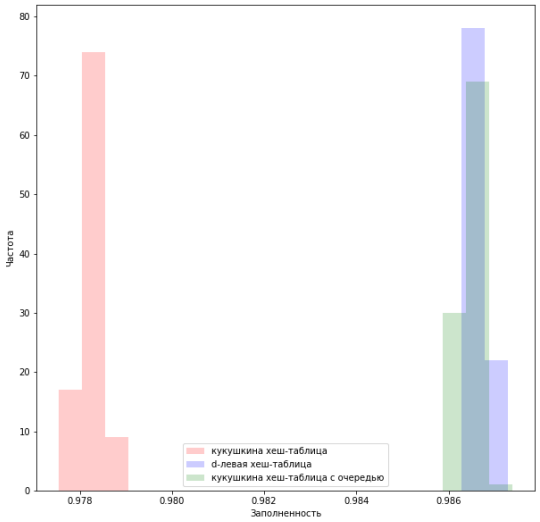


(a) Статический тест 1

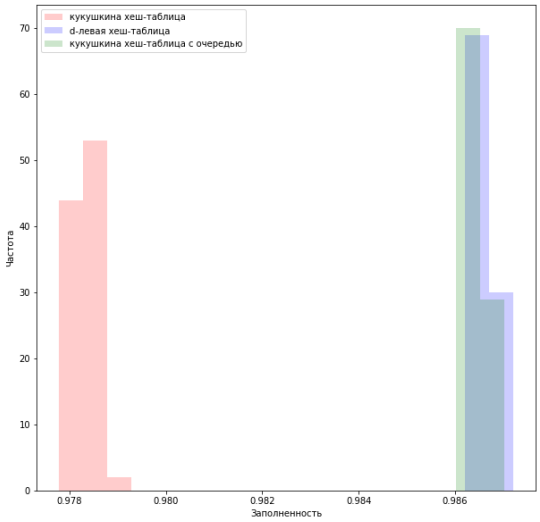


(b) Статический тест 2

Рис. 3: Статические тесты. $d = 2$

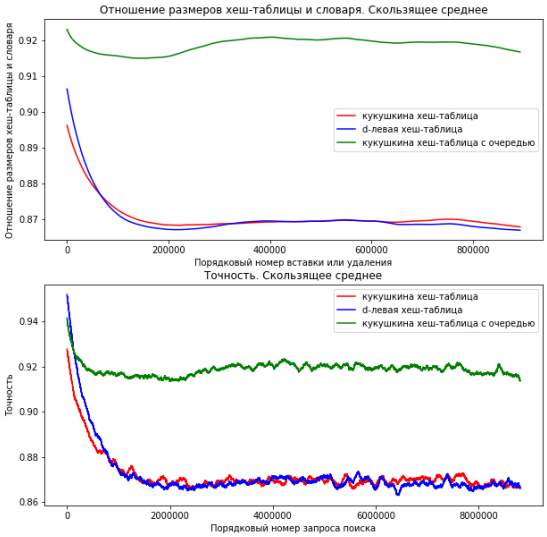


(a) Статический тест 3

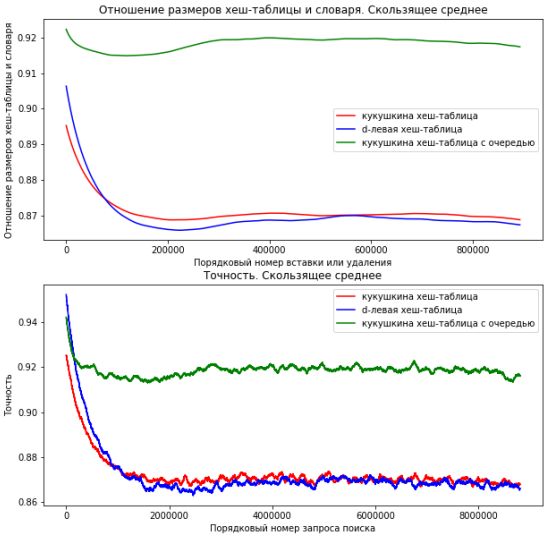


(b) Статический тест 4

Рис. 4: Статические тесты. $d = 4$

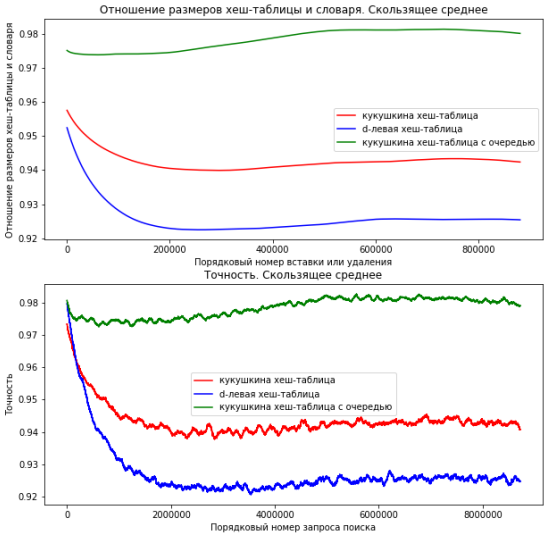


(a) Динамический тест 1

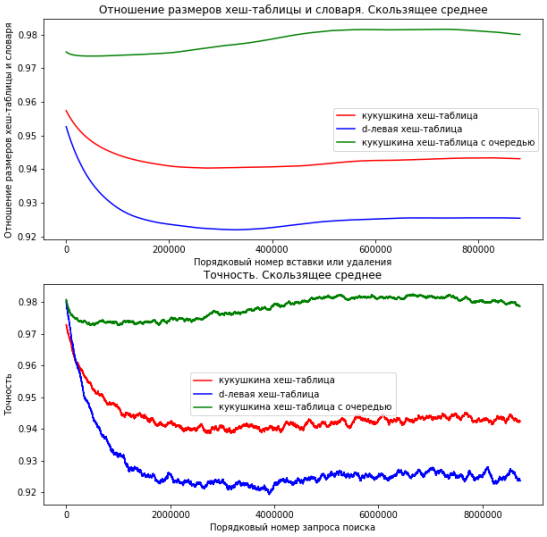


(b) Динамический тест 2

Рис. 5: Динамические тесты. $d = 2$

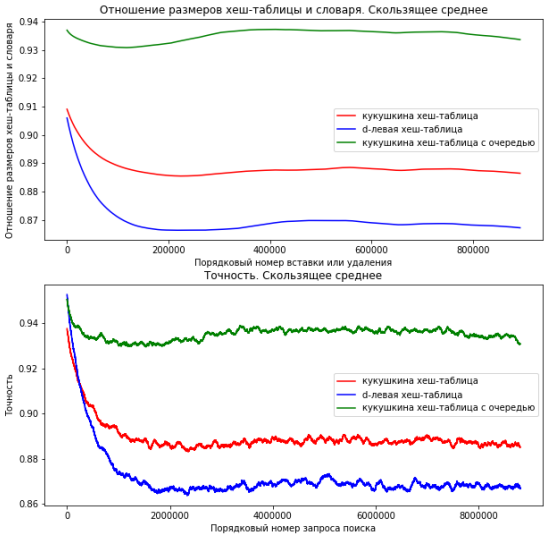


(a) Динамический тест 5

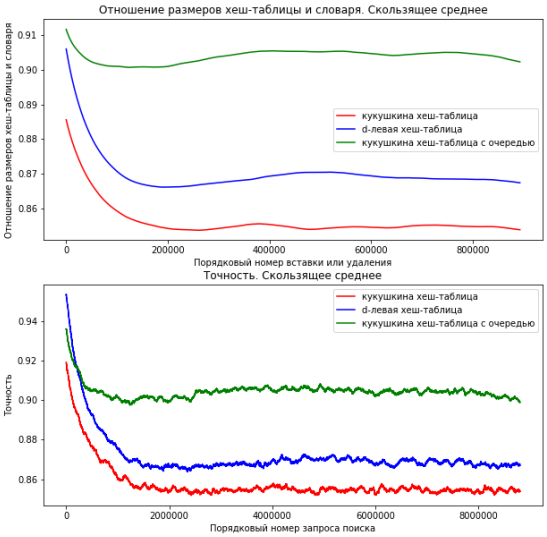


(b) Динамический тест 6

Рис. 6: Динамические тесты. $d = 2$

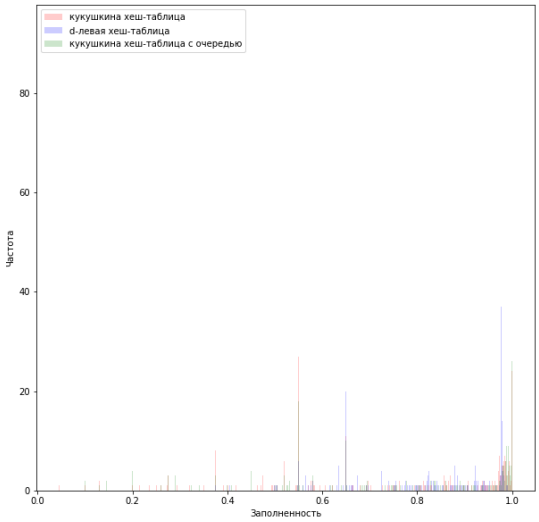


(a) Динамический тест 5

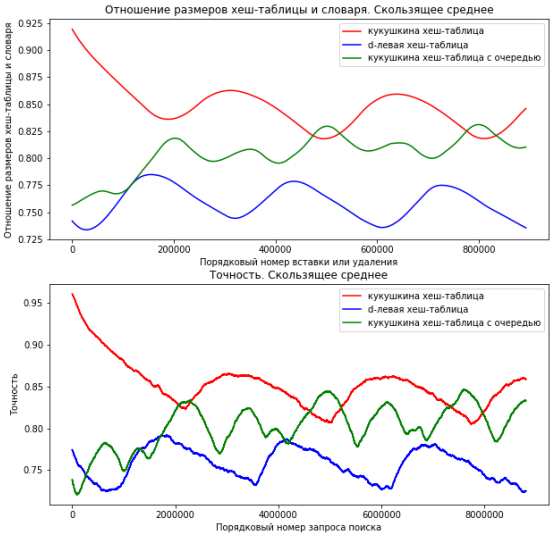


(b) Динамический тест 6

Рис. 7: Динамические тесты. $d = 4$



(a) Статический тест 1



(b) Динамический тест 1

Рис. 8: Полиномиального хеширования степени 1 недостаточно

8 Заключение

Проведенное исследование показывает, что хеширование кукушки является подходящим алгоритмом для телекоммуникационных сетей. Эта структура данных имеет высокий коэффициент полезного использования памяти, операция поиска записи по ключу может быть выполнена за одно обращение к памяти при условии хранения разных таблиц на разных физических носителях со своими устройствами чтения/записи. Также хеширование кукушки более устойчиво к частым изменениям (удалениям старых элементов и вставкам новых), чем, например, d-левое хеширование. Отдельно стоит отметить достаточную гибкость в подстройке кукушкиной хеш-таблицы для конкретных задач. В зависимости от длины считываемого слова можно изменять количество элементов в строке. Нужный компромисс между временем вставки одного элемента и коэффициентом полезного использования памяти достигается выбором подходящего числа итераций внешнего цикла вставки. А компромисс между сложностью архитектуры устройства и использованием памяти достигается выбором нужного числа таблиц.

Список литературы

[1] R. Pagh, F. F. Rodler. Cuckoo hashing. *BRICS Report Series. RS-01-32.* 2001

[2] R. Pagh. On the Cell Probe Complexity of Membership and Perfect Hashing. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01).* ACM Press, New York, 2001

[3] N. Fountoulakis, K. Panagiotou, A. Steger. On the Insertion Time of Cuckoo Hashing. *arXiv:1006.1231v3*

[4] D. Fotakis, R. Pagh, P. Sanders, P. Spirakis. Space Efficient Hash Tables with Worst Case Constant Access Time. 2003, 271–282. 10.1007/3-540-36494-3_25.

[5] K. Mehlhorn. Data structures and algorithms. 1, Sorting and searching. *Springer-Verlag, Berlin, 1984.*

[6] O. Gabber, Z. Galil. Explicit constructions of linear-sized superconcentrators. *J. Comput. System Sci.*, **22**(3):407–420, 1981.

[7] N. Fountoulakis, M. Khosla, K. Panagiotou. The Multiple-orientability Thresholds for Random Hypergraphs. *arXiv:1309.6772.*

[8] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Math.* **6** (2), 306–318.

[9] A. Kirsch, M. Mitzenmacher. Using a Queue to De-amortize Cuckoo Hashing in Hardware. *In Proceedings of the 45th Annual Allerton Conference on Communication, Control, and Computing*, pages 751–758, 2007.

[10] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *In Proceedings of the 16th Annual European Symposium on Algorithms*, pages 611–622, 2008.

[11] Y. Arbitman, M. Naor, G. Segev. De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results. *arXiv:0903.0391v1.*

[12] Y. Azar, A. Z. Broder, A. R. Karlin, E. Upfal. Balanced Allocations. *SIAM J. Comput.*, **29** (1): 180–200, 1999.

[13] B. Vöcking. How asymmetry helps load balancing. *Journal of the ACM* **50**(4):568–589, 2003.

[14] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink. Tight thresholds for cuckoo hashing via XORSAT. *In Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP 2010)*, volume **6198** of Lecture Notes in Computer Science, pages 213–225. 2010.

[15] D. Fernholz and V. Ramachandran. The k-orientability thresholds for $G_{n,p}$. *In Proceedings of the 18th annual ACM-SIAM symposium on Discrete algorithms (SODA 2007)*, pages 459–468, 2007.

[16] A. Frieze and P. Melsted. Maximum matchings in random bipartite graphs and the space utilization of cuckoo hash tables. *Random Structures & Algorithms*, **41**(3):334–364, 2012.

[17] P. Gao and N. C. Wormald. Load balancing and orientability thresholds for random hypergraphs. *In Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC 2010)*, pages 97—104, 2010.

[18] M. Thorup, Y. Zhang. Tabulation-Based 5-Independent Hashing with Applications to Linear Probing and Second Moment Estimation. *SIAM Journal on Computing*, **41**(2):293-331, 2012.

[19] M. Mitzenmacher, S. Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. *In Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 2008, San Francisco, California, USA, January 20-22, 2008.