Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
GRADUATION THESIS

Разработка самоподстраивающейся динамической сети на основе списка с пропусками

Обучающийся / Student Надуткин Федор Максимович
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group M34381
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

Обучающийся/Student

Документ подписан
Надуткин Федор Максимович
17.05.2023

(эл. подпись/ signature)

Надуткин Федор Максимович
(Фамилия И.О./ name and surname)

Руководитель ВКР/ Thesis supervisor

Документ подписан
Аксенов Виталий Евгеньевич
17.05.2023

(эл. подпись/ signature)

Аксенов Виталий Евгеньевич
(Фамилия И.О./ name and surname)

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ / OBJECTIVES FOR A GRADUATION THESIS

Обучающийся / Student Надуткин Федор Максимович
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group M34381
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Тема ВКР/ Thesis topic Разработка самоподстраивающейся динамической сети на основе списка с пропусками
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

Основные вопросы, подлежащие разработке / Key issues to be analyzed
Требуется изучить проблематику самоподстраивающихся конкурентных сетей, разработать и реализовать свои решения на основе адаптивной версии списка с пропусками - SplayList.
Конкретные подзадачи:
1. Реализация статической сети на основе SkipList, как основной объект для сравнения.
2. Реализация адаптивных сетей на основе SplayList.
3. Сравнение производительности алгоритмов адаптивных и статических.
4. Изучение возможности частичной адаптивности (когда обновление происходит с определённой вероятностью) и реализация алгоритмов, использующих эту идею.
5. Сравнение производительности частично адаптивных алгоритмов с адаптивными и статическими вариантами сетей.

Форма представления материалов ВКР / Format(s) of thesis materials:
Программный код, презентация, пояснительная записка

Дата выдачи задания / Assignment issued on: 31.01.2023

Срок представления готовой ВКР / Deadline for final edition of the thesis 22.05.2023

Характеристика темы ВКР / Description of thesis subject (topic)

Тема в области фундаментальных исследований / Subject of fundamental research: да /

yes
Тема в области прикладных исследований / Subject of applied research: нет / not

СОГЛАСОВАНО / AGREED:

Руководитель ВКР/ Thesis supervisor

Документ подписан
Аксенов Виталий Евгеньевич
16.05.2023

(эл. подпись)

Аксенов Виталий Евгеньевич

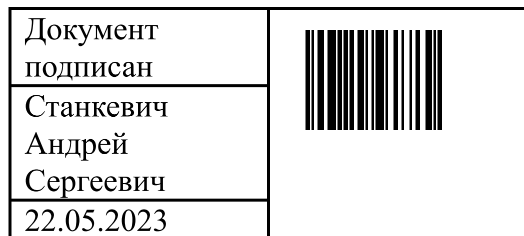
Задание принял к исполнению/ Objectives assumed BY

Документ подписан
Надуткин Федор Максимович
16.05.2023

(эл. подпись)

Надуткин Федор Максимович

Руководитель ОП/ Head of educational program


Документ подписан
Станкевич Андрей Сергеевич
22.05.2023

(эл. подпись)

Станкевич Андрей Сергеевич

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
SUMMARY OF A GRADUATION THESIS

Обучающийся / Student Надуткин Федор Максимович
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group M34381
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Тема ВКР/ Thesis topic Разработка самоподстраивающейся динамической сети на основе списка с пропусками
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
DESCRIPTION OF THE GRADUATION THESIS

Цель исследования / Research goal
Разработка и реализация самоподстраивающейся конкурентной сети на основе списка с пропусками
Задачи, решаемые в ВКР / Research tasks
1. Реализация статической сети на основе SkipList, как основной объект для сравнения. 2. Реализация адаптивных сетей на основе SplayList. 3. Сравнение производительности алгоритмов адаптивных и статических. 4. Изучение возможности частичной адаптивности (когда обновление происходит с определённой вероятностью) и реализация алгоритмов, использующих эту идею. 5. Сравнение производительности частично адаптивных алгоритмов с адаптивными и статическими вариантами сетей. 6. Разработка и описание конкурентных версий алгоритмов.
Краткая характеристика полученных результатов / Short summary of results/findings
1. SimpleSplayListNet даёт лишь небольшой выигрыш на часто повторяющихся запросах и даёт существенный проигрыш на редких запросах. 2. TreeSplayListNet больший прирост производительности, однако сильно нагружает head, что может стать узким местом нашей программы. 3. Структуры ParentChildNet дают существенный прирост производительности, причём чем меньше величина листа, тем лучше структура себя использует. Однако стоит выверять баланс, ведь чем меньше длина интервала, тем больше

соединений приходится на один сервер. 4. Добавление вероятностного обновления улучшает показатели алгоритма на редко повторяемых запросах, но почти нивелируется, когда запросы повторяются часто.

Обучающийся/Student

(эл. подпись/ signature)

Надуткин Федор Максимович

(Фамилия И.О./ name and surname)

Руководитель ВКР/ Thesis supervisor

(эл. подпись/ signature)

Аксенов Виталий Евгеньевич

(Фамилия И.О./ name and surname)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 5
1. Базовые структуры и определения 9
1.1. Базовая Модель. 9
1.2. SkipListNet 10
1.3. SplayList. 11
1.4. Выводы к главе 1 15
2. Структуры на основе SplayList 16
2.1. SimpleSplayListNet 16
2.2. TreeSplayListNet 19
2.3. Выводы к главе 2 25
3. Структуры, не идущие назад при поиске. 29
3.1. LeftRightSplayNet 29
3.2. ParentChildNet 29
3.3. Выводы к главе 3 36
4. Вероятностные структуры 38
4.1. ProbabilitySimpleSplayListNet. 38
4.2. ProbabilityTreeSplayListNet 38
4.3. ProbabilityFrontTreeSplayListNet 40
4.4. Выводы к главе 4 43
5. Конкурентность 44
5.1. Выводы к главе 5 45
6. Тестирование на реальных данных 46
7. Итоги 47
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 49

ВВЕДЕНИЕ

Зачастую, когда речь заходит об организации коммуникационной сети, используются статические структуры, не зависящие от идущего в ней трафика, такие как например толстые деревья или сильно связанные разреженные графы (экспандеры) [4, 11]. Однако, как показывают исследования, шаблоны коммуникации в центрах обработки данных обладают значительной пространственной и временной структурой , то есть трафик бывает всплесками, а матрицы трафика несбалансированы [9, 13, 15, 17]. Это показывает неэффективное использование ресурсов в статических структурах. В отличие от этого сети основанные на запросах и модифицирующихся в зависимости от них, например, на основе реконфигурируемых оптических технологий, могут самонастраиваться, чтобы лучше обслуживать основные потоки в сети, устанавливая прямые связи между часто общающимися машинами в дата-центре [5, 7–10, 16].

Этот диплом исследует алгоритмическую проблему самоподстраивающихся сетей. Большинство существующих подстраивающихся сетей используют решения на основе splay деревьев [6, 12, 16]. Когда от структуры требуется распределённость, то используют конкурентную структуру `Cbtree` [14], поддерживающую баланс в зависимости от запросов к её вершинам. В последовательной реализации этой структуры скорость выполнения запросов совпадает со сложностью выполнения в статических структурах данных, заранее заготовленных с учётом заданых запросов. В то же время распределённая реализация `Cbtree` становится достаточно трудной в реализации из-за сложности перебалансировки. Более того, поддержание точной статистики и баланса могут негативно сказаться на производительности самой структуры, поэтому авторы предполагают вариант с ограниченным параллелизмом, при котором перебалансировка делегируется одному потоку. Однако, подобный вариант ставит нас в зависимость от этого потока, и при его зависании останавливается вся перебалансировка системы, что также может повлечь за собой просадки в производительности.

В этой работе мы обращаемся к другой адаптивной распределённой структуре `SplayList` [18] и делаем на её основе свою самоподстравивающуюся сеть `SplayListNet`, а также её версии с различными оптимизациями. Отдалённо наша структура может напоминать сеть, сделанную на основе

`SkipList` [1], как например `SkipListNet`. Однако фундаментальное отличие заключается в том, что в `SkipListNet` высота элемента выбирается не случайно, а в зависимости от числа обращений от него или к нему, поэтому элементы, от которых или к которым чаще всего идут запросы, поднимаются выше, а как следствие доступ к ним становится быстрее.

Гарантии: После m операций амортизированная скорость запроса между вершинами x и y становится равной $\mathcal{O}(\log \frac{m}{f(x_1,x_2)} + \log \frac{m}{f(y_1,y_2)})$, где x_1/x_2 — число раз, когда x было началом/концом запросом, y_1/y_2 — число раз, когда y было началом/концом запроса, а $f(x_1,x_2)$ — некоторая возрастающая функция. Также к каждой версии `SplayListNet` будет предложена её конкурентная реализация, для простоты основанная на блокировках.

Проделанная работа:

В первой главе представлена модель конкурентных коммуникационных сетей, основная терминология, `SplayList` на основе которого будут строиться разрабатываемы в статье структуры, а также описание и реализация `SkipListNet` с которой мы впоследствии будем сравнивать наши алгоритмы и эти методы сравнения.

Во второй главе представлены базовые `routing` структуры на основе `SplayList`, они все ходят в `head` при посылке запроса. Такими структурами являются простейшая реализация адаптивной сети, `SimpleSplayListNet`, и версия адаптивной сети, `TreeSplayListNet`, ищущая общие вершины на пути к `head`. Также в главе предоставлены доказательство их работы, замеры скорости их работы и сравнение.

В третьей главе приведены структуры, которым уже не нужно идти в `head` для посылки запроса. Описаны две структуры: 1) структура, `LeftRightSplayNet`, которая умеет посылать запросы из одной половины в другую, и 2) структурка, `ParentChildNet`, представляющая из себя дерево из `LeftRightSplayNet` со структурами из главы 2 в листьях. Также проведено сравнение алгоритмов в зависимости от того какая структура используется в листьях и порога начиная с которого строится лист.

В четвёртой главе приведены вероятностные алгоритмы, они проводят обновление лишь с определённой вероятностью, а не после каждого запроса. Также приведены модификации, использующие тот факт, что им не нужно обновлять структуру, такие как например

`ProbabilityFrontTreeSplayListNet`. Также доказана скорость их работы и проведены сравнения с алгоритмами, описаными в раних главах.

В пятой (финальной) главе описано, как реализовывать конкурентные версии данных алгоритмов. Также показано какие модификации стоит сделать и почему.

В главе "Итоги" подведены основные итоги дипломной работы.

Используемая терминология:

1) `Адптивный(ая)` - самоподстраивающийся(аяся).
2) `Статический(ая)` - постоянный(ая).
3) `value` - значение.
4) `SkipList` - вероятностная структура данных, основанная на нескольких параллельных отсортированных связных списках с эффективностью, сравнимой с двоичным деревом (порядка $\mathcal{O}(\log n)$ среднее время для большинства операций). Подробнее можно прочитать здесь [1]
5) `head` - начальная вершина структур на основе `SkipList`, обычно не хранит никаких значений и ключей, используется для удобства работы со структурой.
6) `tail` - конечная вершина структур на основе `SkipList`, обычно не хранит никаких значений и ключей, используется для удобства работы со структурой.
7) `SplayList` - адаптивная структура данных, на основе `SkipList`. Подробно описана [18].
8) `send` - операция посылки запроса из одной вершины структуры в другую, реализуется в `routing` структурах (подробнее о них в главе 1).
9) `find` - операция поиска вершины в структуре, реализуется в `search` структурах (подробнее о них в главе 1).
10) `descend` - спуск, операция применяющаяся в адаптивных структурах при выполнении определённых условий. Подробнее описана в главе 1.3.
11) `ascend` - подъём, операция применяющаяся в адаптивных структурах при выполнении определённых условий. Подробнее описана в главе 1.3.
12) `Инвариант` — свойство, остающееся неизменным при преобразованиях определённого типа.
13) `Трафик` - информация передающаяся по коммуникационной сети.
14) `MaxLevel` - наивысший уровень структуры.

15) `Синхронизация` - Инвариант — свойство, остающееся неизменным при преобразованиях определённого типа.
16) `Синхронная структура` - структура, выполняющая запросы последовательно.
17) `Конкурентная структура` - структура, способная выполнять несколько запросов одновременно.
18) `lock/лок` - блокировка ресурса при которой с ресурсом может работать только тот, кто взял данную блокировку.
19) `deadLock` - ситуация, когда несколько объектов имеют блокировку на разные ресурсы, и для продолжения работы им требуются ресурс на который не взят `lock`.
20) `Статическая нагрузка` — нагрузка, величина, направление и точка приложения которой изменяются во времени незначительно.

ГЛАВА 1. БАЗОВЫЕ СТРУКТУРЫ И ОПРЕДЕЛЕНИЯ

1.1. Базовая Модель

Нам дана коммуникационая сеть — routing структура, соединяющая множество вершин $V = \{v_1, ..., v_n\}$ и множество запросов $\sigma = (\sigma_1, \sigma_2, ..., \sigma_m)$ между вершинами: $\sigma_i = (s_i, f_i)$, где s_i — стартовая вершина, f_i — конечная вершина, $s_i < f_i$, а m может быть неограничено. Запросы в нашей системе заранее не известны и генерируются случайно (если не задано никаких других ограничений, например на повторение запроса k раз). Когда запрос посылается, он проходит определённый путь из вершин “передающих” сообщение от s_i до f_i, таким образом σ_i может пройти путь $s_i, v_{i_1}, v_{i_2}, ..., f_i$ прежде чем достигнет конечной точки.

Определение 1. Пусть $\sigma_i = (s_i, f_i)$, *временем запроса* t_i будем называть число обновлений + число рёбер, через которые прошло сообщение из s_i прежде чем достигнуть f_i.

Определение 2. Пусть дана сеть S и множество запросов $\sigma = (\sigma_1, \sigma_2, ..., \sigma_m)$. **Общим временем работы** T будем называть суммарное время всех запросов. $T = \sum\limits_{i=0}^{m} t_i$.

Определение 3. Пусть дана сеть S и множество запросов $\sigma = (\sigma_1, \sigma_2, ..., \sigma_m)$. **Амортизированным временем работы** T_A будем называть общее время работы делённое на количество всех запросов. $T_A = \frac{1}{m} \cdot \sum\limits_{i=0}^{m} t_i$.

Сравнение алгоритмов будет проводиться по T_A для одинаковых множеств запросов. Алгорим сравнения:

1) Берётся массив запросов (массив одинаков для всех).
2) Выбирается количество раз $repeats$, которое запрос будет повторяться (некая степень 2).
3) В структуре $repeats$ раз посылается каждый запрос.
4) Считается T_A.
5) Строится график зависимости T_A от $repeats$.

Для алгоритмов, зависящих и меняющих другие параметры (например вероятность обновления), алгоритм проверяет для всех доступных значений.

1.2. SkipListNet

`SkipListNet` является базовым алгоритмом для статической коммуникационной сети. Все последующие сравнения будут проводиться именно с ним, чтобы выяснить выигрыш/проигрыш в производительности той или иной структуры.

Сама структура по построению идентична обычному `SkipList` [1]. Каждый уровень представляет из себя односвязанный список с началом в head и концом в tail в основе которого лежат объекты класса `Node` у которых есть следующие поля:

- `key`, ключ, хранящийся в данной вершине;
- `value`, значение, хранящееся в данной вершине;
- `next`, ссылка на следующую вершиину, на каждом уровне, где есть наша вершина;
- `topLevel`, максимальный уровень на котором есть данная вершина.

Пример класса приведён в листинге 1.

Листинг 1 – класс Node

```
class Node<K: Comparable<K>, V>(
    val key: K?,
    var value: V?,
    var next: List<Node<K, V>> = ArrayList(),
    var topLevel: Int = 0)
```

Посылка запроса $\sigma_i = (s_i, f_i)$ осуществляется посредством операции `send`, принимающей на вход стартовую вершину и ключ конечной точки, а также функцию, которую нужно будет осуществить по достижении f_i.

Теперь приступим к описанию алгоритма. Пример работы приведён на рисунке 1, а псевдокод доступен в листинге 2. используешь $s_i < f_i$ Алгоритм состоит из двух фаз:

1) Фаза `Up`. До тех пор пока мы не нашли вершину, чей `key` на текущем будет не меньше, чем f_i (или не дошли до конца).
   (1) Идём вправо.
   (2) В текущей вершине поднимаемся на уровень выше.
2) Фаза `Down`. Когда мы нашли подходящую вершину повторяем операцию `find` из обычного `SkipList`.

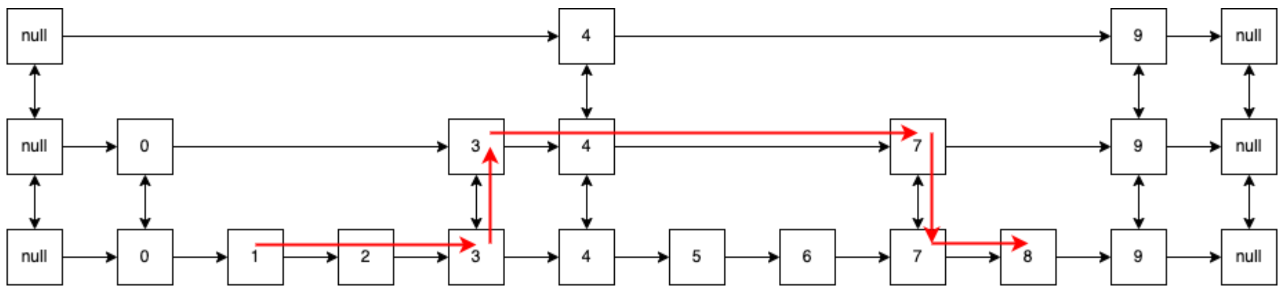


Рисунок 1 – Алгоритм посылки запроса в SkipListNet из вершины с key = 1 в вершину с key = 8.

Листинг 2 – Посылка запроса в SkipListNet

```
fun send(start: Node<K, V>, finish: K, function: (V, V) -> Unit) {

    var current = start
    var currentLevel = start.topLevel

    //region High
    while (current.next[currentLevel].key != null && current.next[currentLevel
        ].key < finish) {
        current = current.next[currentLevel]
        currentLevel = current.topLevel
    }
    //endregion

    //region Down
    for (index in currentLevel downTo 0) {
        while (current.next[index].key != null && current.next[index].key <=
            finish) {
            current = current.next[index]
        }

        if (current.key == finish) {
            function(start.value, current.value)
        }
    }
    //endregion

    throw RuntimeException("Request between ${start.key} and $finish not
        succeeded")
}
```

1.3. SplayList

Прежде чем переходить к описанию алгоритмов на основе `SplayList` [18], нам стоит её описать. Это `search` структура на основе которой мы будем делать `routing` структуры.

Определение 4. *Поддеревом вершины u на высоте h* называются все вершины v для которых выполнено $u.key < v.key < u.next[h].key$.

`SplayList` представляет из себя множество связанных списков, состоящих из объектов класса `SplayNode`, упорядоченных по возрастанию (если не заданно какого-то другого порядка). Объекты класса `SplayNode` имеют следующие поля.

1) `key`, ключ, хранящийся в вершине;
2) `value`, значение, хранящееся в вершине;
3) `next`, ссылка на следующую вершиину, на каждом уровне, где есть вершина;
4) `previous`, ссылка на предыдущую вершину, на каждом уровне, где есть эта вершина;
5) `topLevel`, максимальный уровень, на котором есть эта вершина;
6) `selfHits`, число запросов к данной `SplayNode`;
7) `hits`, массив, где на каждом уровне h представлена сумма `selfHits` вершин из поддерева нашей вершины на высоте h.

Пример класса приведён в листинге 3.

Листинг 3 – Класс SplayNode

```
class SplayNode<K : Comparable<K>, V>(
    val key: K?,
    var value: V?,
    var next: List<SplayNode<K, V>> = ArrayList(),
    var previous: MutableList<SplayNode<K, V>> = ArrayList(),
    var topLevel: Int = 0,
    var selfHits: Int = 0,
    var hits: MutableList<Int> = ArrayList())
```

Основным отличием класса `SplayList` от `SkipList` является распределение вершин по уровням. Все операции, которые проводятся со `SplayList` делятся на 2 фазы:

1) Поиск вершины и выполнение самой операции (Прямой проход);
2) Обновление значений (Обратный проход).

Во время обратного прохода обновляемые вершины могут как подняться, так и опуститься, если выполняются определённые условия. Сформулируем эти условия.

Пусть дана структура `SplayList` и вершина u. Введём функцию $hits(u,h) = u.hits[h] + u.selfHits$. Пример кода в листинге 4:

Листинг 4 – Код getHits(node, height)

```
fun getHits(node: SplayNode<K, V>, height: Int): Int {
    if (height > node.topLevel) {
        return node.selfHits
    }
    return node.hits[height] + node.selfHits
}
```

Условие спуска: Дана структура `SplayList` S с максимальным уровнем H. Если вершина u не является `head`, то существует $v = u.previous[u.topLevel]$. Обозначим $u.topLevel$ как h, в случае, если выполняется условие:

$$hits(u,h) + hits(v,h) \leqslant \frac{m}{2^{H-h}}$$

говорят, что u удовлетворяет условию спуска. Пример спуска приведён в листинге 5:

Листинг 5 – Код descend

```
fun descend(currentNode: SplayNode<K, V>,
            previousNode: SplayNode<K, V>) {
    val currentHeight = currentNode.topLevel
    val descendingCondition = M / (2.0.pow(MaxLevel - currentHeight))

    val currentHits=getHits(currentNode, currentHeight)
    val previousHits=getHits(previousNode,currentHeight)
    if (currentHits + previousHits <= descendingCondition) {
        val nextNode = currentNode.next[currentHeight]
        previousNode.hits[currentHeight] += currentHits

        previousNode.next[currentHeight] = currentNode
            .next.removeAt(currentHeight)
        nextNode.previous[currentHeight] = currentNode
            .previous.removeAt(currentHeight)
        currentNode.hits.removeAt(currentHeight)
        currentNode.topLevel--
    }
}
```

Условие подъёма: Дана структура `SplayList` S с максимальным уровнем H и вершина u_0. Пусть v - такая вершина, что $u_0.topLevel < v.topLevel$,

$u_0.key < v.key$, $v.key$ — минимален (если u_0 не является `head` или `tail`, то такая вершина существует). Пусть $u_{1...k}$ - все такие вершины, что $u_0.topLevel = u_1.topLevel = ... = u_k.topLevel$ и $u_0.key < u_1.key < ... < u_k.key < v.key$. Обозначим $u.topLevel$ как h, если выполнено условие

$$\sum_{i=0}^{k} hits(u_i, h) > \frac{m}{2^{H-h-1}}$$

то говорят, что u_0 удовлетворяет условию подъёма.

Замечание 1. Рассмотрим вершину u, такую что $\exists$ $p = u_0.previous[u_0.topLevel] : p.topLevel > u_0.topLevel$. В статье [18] приводится доказательство, что только u_0 может соответствовать условию подьёма. Таким образом, $\sum_{i=0}^{k} u_i.hits[h] = p.hits[h+1] - p.hits[h]$. Пример подъёма показан на листинге 6.

Листинг 6 – Код ascend

```
fun ascend(currentNode: SplayNode<K, V>, previousNode: SplayNode<K, V>) {
    val currentHeight = currentNode.topLevel
    val ascendingCondition = M / (2.0.pow(MaxLevel - currentHeight - 1))

    while (previousNode.topLevel > currentHeight
    && previousNode.hits[currentHeight + 1] - previousNode.hits[currentHeight]
        > ascendingCondition) {
        val nextNode = previousNode.next[currentHeight + 1]

        currentNode.next.add(next)
        currentNode.previous.add(previous)
        currentNode.topLevel++

        nextNode.previous[currentHeight + 1] = currentNode
        previousNode.next[currentHeight + 1] = currentNode

        currentNode.hits.add(previous.hits[currentHeight + 1] - previousNode.
            hits[currentHeight] - current.selfHits)
        previousNode.hits[currentHeight + 1] = previousNode.hits[currentHeight]
        currentHeight = currentNode.topLevel
        ascendingCondition = M / (2.0.pow(maxLevel - currentHeight - 1))
    }
}
```

Замечание 2. В случае, если $u_0.topLevel = H$, нужно увеличить H на 1, поэтому перед тем как проводить обновление стоит проверить, что $\log m \geqslant H$,

и если это условие выполняется, увеличить H. Пример роста уровня в листинге 7.

Листинг 7 – Новый уровень

```
fun newLevel() {
    if (2.0.pow(MaxLevel) <= M) {
        val level = head.topLevel

        head.hits.add(head.hits[level])
        head.next.add(tail)
        tail.previous.add(head)
        head.topLevel++
        tail.topLevel++
        MaxLevel++
    }
    return
}
```

1.4. Выводы к главе 1

В данной главе были:

1) Введена основная терминология, которая будет использована во всей дальнейшей работе.
2) Описан алгоритм сравнения routing структур.
3) Представлена routing структура `SkipListNet` с которой будут проводиться основные сравнения в работе.
4) Показан и объяснён алгоритм `SplayList`, который будет использоваться в качестве основы для всех дальнейших алгоритмов.

ГЛАВА 2. СТРУКТУРЫ НА ОСНОВЕ SPLAYLIST

2.1. SimpleSplayListNet

Это простейшая вариация реализации сети на основе `SplayList`. Сама структура абсолютно идентична обычному `SplayList`, единственное отличие заключается только в том, что вместо операции `find`, мы должны выполнить операцию `send`. Его основная идея в том, что мы посылаем запрос $\sigma_i = (s_i, f_i)$ из s_i в корневую вершину, а затем из корневой вершины в f_i. Опишем этот алгоритм:

1) Нас просят выполнить $send(s_i, f_i)$.
2) Увеличиваем $s_i.selfHits$ и m на 1 и совершаем обратный проход от s_i до `head` с обновлением всех значений. Пример кода в листинге 8.
3) Осуществляем поиск `SplayNode` со значением f_i.
4) Увеличиваем $f_i.selfHits$ и m на 1 и совершаем обратный проход от f_i до `head` с обновлением всех значений.

Листинг 8 – Код update

```
fun addHits(node: SplayNode<K, V>, start: Int) {
    for (level in start..node.topLevel) {
        node.hits[level]++
    }
}

fun update(updated: SplayNode<K, V>) {
    var currentNode = updated
    currentNode.selfHits++
    newLevel()

    while (currentNode != head) {
        val currentHeight = currentNode.topLevel
        val previousNode = currentNode.previous[currentHeight]
        addHits(previousNode, currentHeight)

        ascend(currentNode, previousNode)
        descend(currentNode, previousNode)
        currentNode = previousNode
    }
}
```

Рассмотрим следующий пример.

1) Предположим у нас есть структура `SimpleSplayListNet` (Рисунок 2) S с максимальным уровнем $H = 4$ (нумерация с 0) и $m = 10$. Мы хотим послать запрос из вершины 2 в вершину 8.
2) (Рисунок 3) Мы увеличиваем счётчик вершины 2 до двух, проверяем, что $\log 11 < 4$ (правда). Поднимаемся на её максимальную уровень вершины (1) и находим её $previous[1] == 1$.
   (1) Максимальный уровень обеих вершин одинаков, согласно замечанию 1 мы не можем поднять вершину выше.
   (2) $hits(2, 1) + hits(1, 1) = 3 > \frac{11}{2^{4-1}} \Rightarrow$ вершина не опускается.
   (3) Переходим к вершине 1.
3) (Рисунок 4). Мы находимся на высоте 1. Поднимаемся $1.topLevel == 1$ и находим $1.previous[1] == head$. Обновляем все $head.hits$ для высоты выше 1.
   (1) Проверяем $hits(head,1) + hits(1,1) = 3 > \frac{11}{2^{4-1}} \Rightarrow$ вершина не опускается.
   (2) 3 - первая вершина, что $3.topLevel > 1.topLevel$. Проверяем сумму всех вершин от 1 на высоте 1 $hits(1,1) + hits(2,1) = 1 + 2 = 3 > \frac{11}{2^{4-2}} \Rightarrow$ вершина должна подняться на новый уровень.
   (3) После того как вершина 1 поднялась на высоту 2 $head.hits[2]$ стало равно 1, $1.hits[2]$ стало равно 2, остальные $head.hits$ остались прежними.
   (4) Если теперь мы проверим условие подъёма для 1, то получим $hits(1, 2) + hits(3, 2) = 4 < \frac{11}{2^{4-3}}$
4) После того как мы дошли до `head`, мы начинаем поиск вершины 8 (Рисунок 5), операция идентична поиску, что осуществлён в `SplayList`. Когда мы дошли до вершины 8, мы можем спокойно передать сообщение.
5) Когда мы осуществляем обратный проход, ситуация с вершинами 8, 7 (Рисунки 6 и 7) схожа с описаными выше пунктами. Мы также увеличиваем счётчик $8.selfHits$ на 1, проверяем, что $\log m < H$, и поднимаем вершину в случае, если выполняется условие подъёма.

6) При переходе от 6 к 4, можно заметить, что $hits(6,2) + hits(4,2) = 3 = \frac{12}{2^{4-2}} \Rightarrow$ вершина 6 должна быть опущена, при этом $4.hits[2] = 2$ (Рисунок 8).
7) Ну и наконец мы доходим от вершины 4 до `head`, проведя вся должные обновления. Операция `send` закончена.

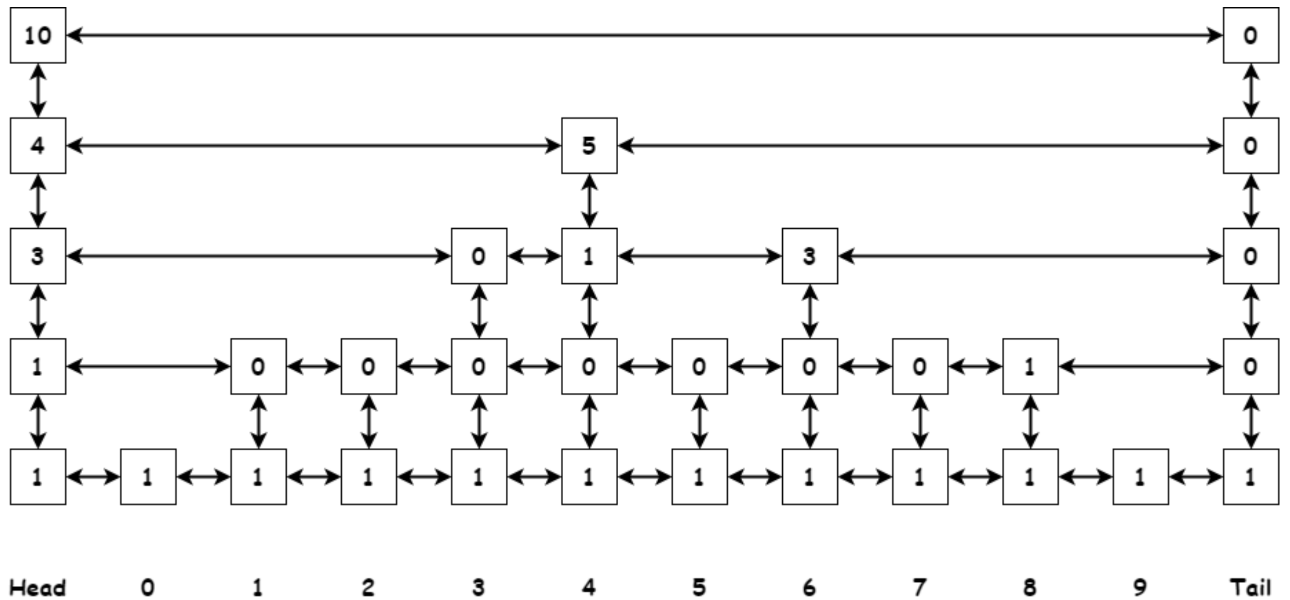


Рисунок 2 – Изначально данная нам `SimpleSplayListNet`

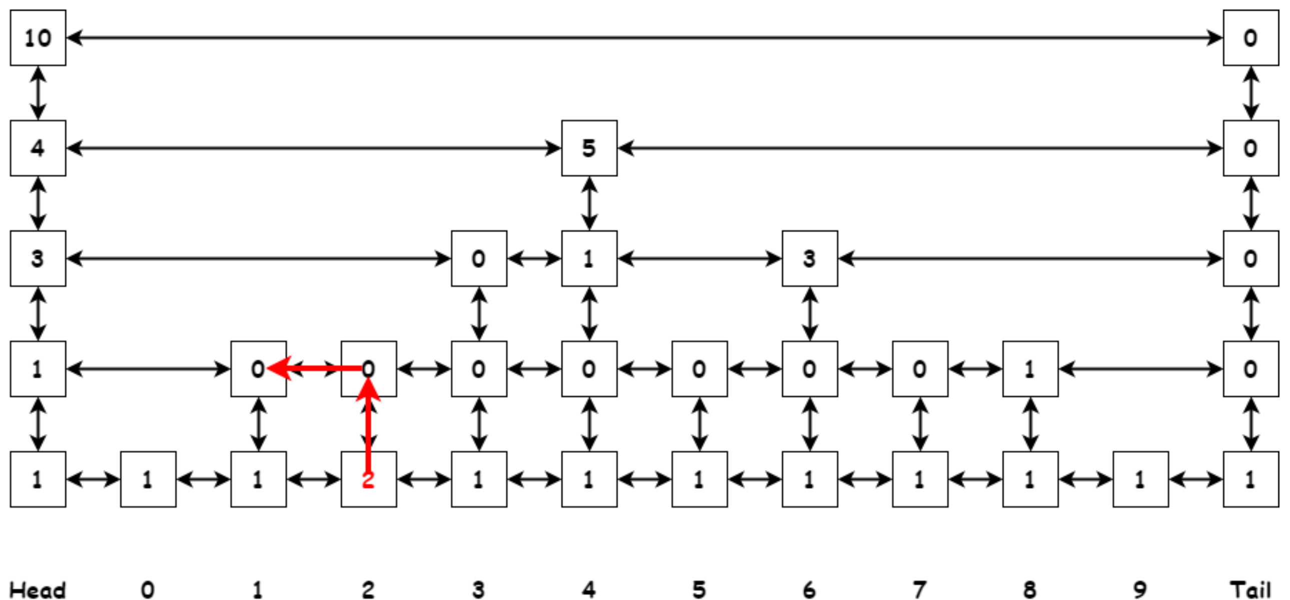


Рисунок 3 – Переход от вершины 2 к вершине 1

Сравнение этого алгоритма со SkipListNet (Рисунок 9) показывает неудовлетворительные результаты. Мы начинаем получать выигрыш только начи-

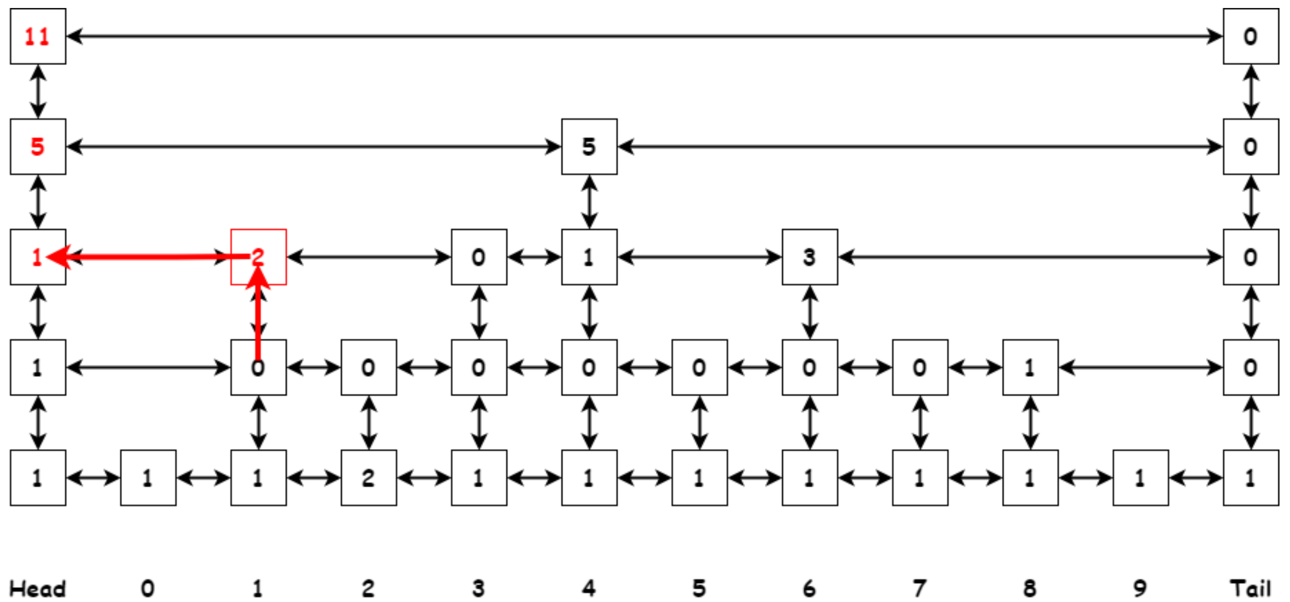


Рисунок 4 – Подъём вершины 1 и переход от вершины 1 к вершине head

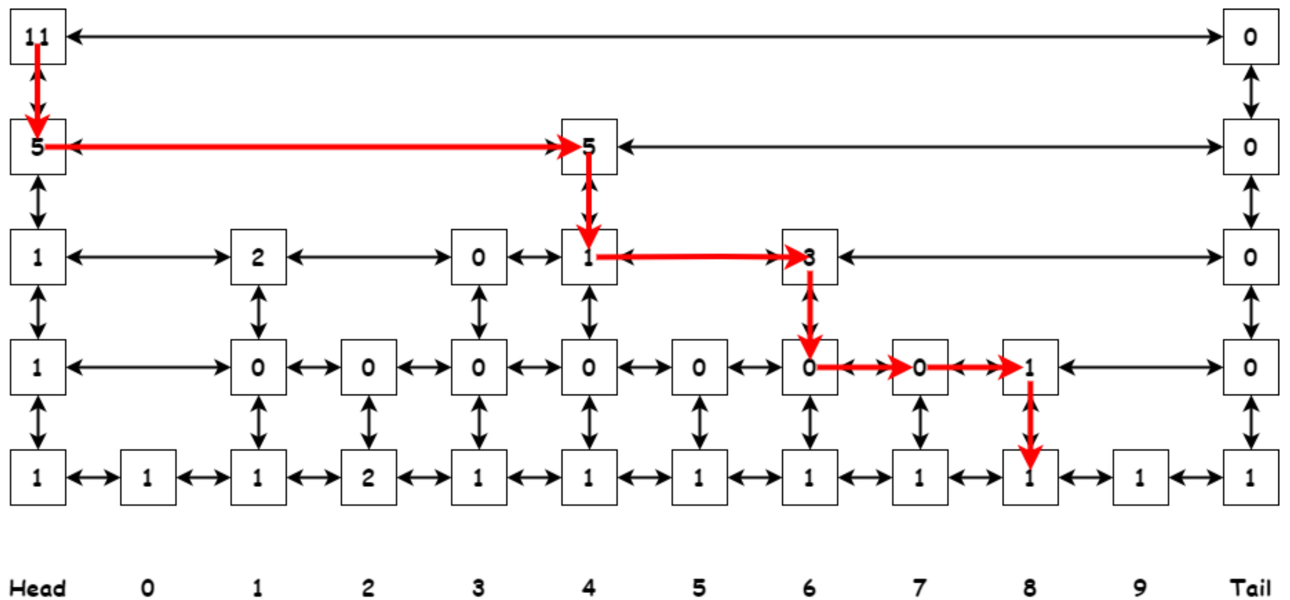


Рисунок 5 – Поиск вершины 8

ная повторять запрос более 500 раз, но даже такой выигрыш нельзя назвать достаточно существенным.

2.2. TreeSplayListNet

Для ускорения работы попробуем добавить оптимизацию и ходить не к голове, а к наименьшему общему предку.

Определение 5. Пусть даны вершины u_1 и u_2, $u_1.key < u_2.key$. Наименьший общий предок вершин u_1, u_2 — это вершина v, для которой выполняются следующие условия:

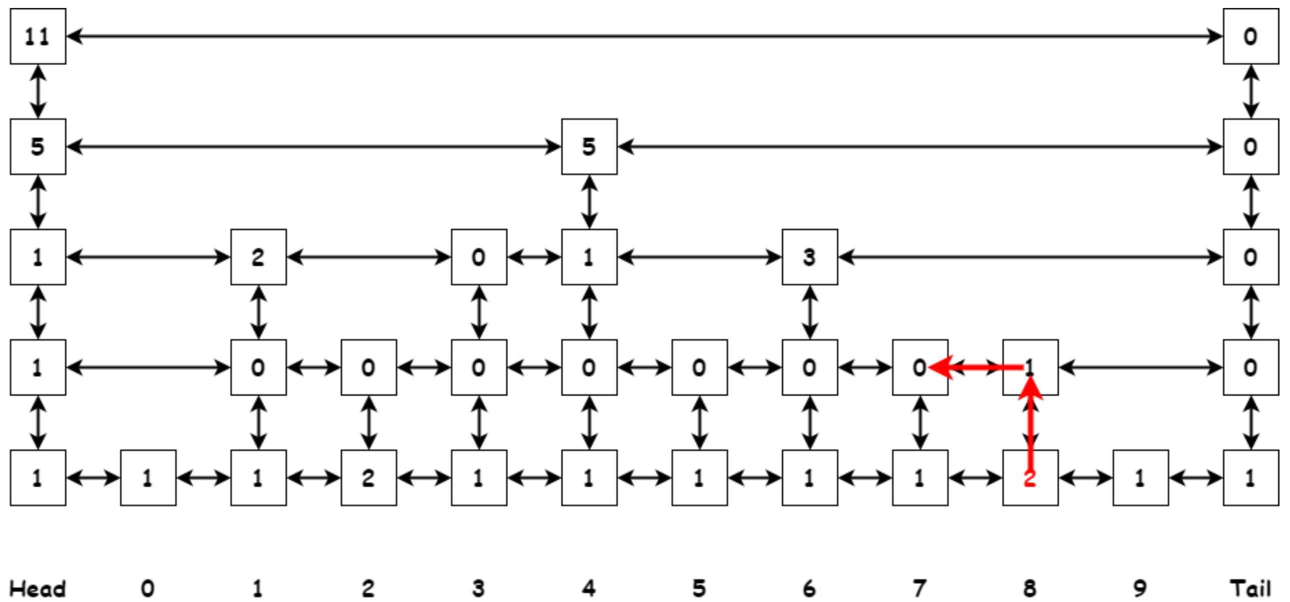


Рисунок 6 – Обратный проход от 8 к 7

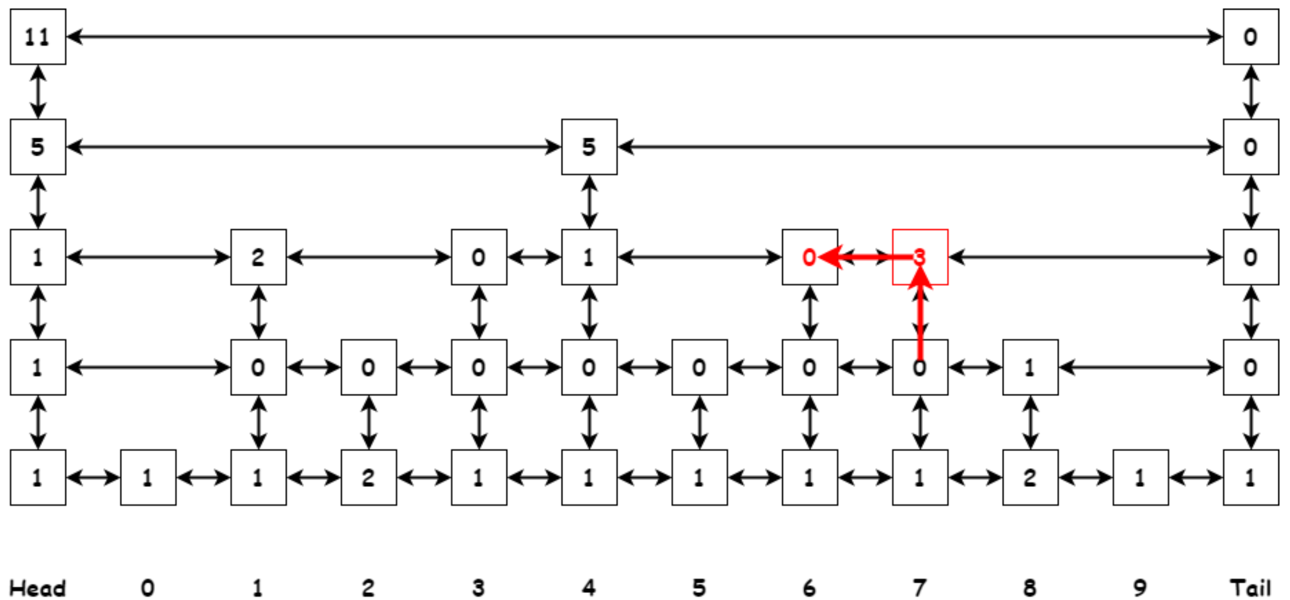


Рисунок 7 – Обратный проход от 7 к 6

1) $v.topLevel \geqslant u_1.topLevel$ и $v.topLevel \geqslant u_2.topLevel$
2) существует h: u_1 находится в поддереве v на высоте h или $u_1 = v$
3) существует h: u_2 находится в поддереве v на высоте h или $v.next[h] = u_2$

В этом алгоритме мы в первую очередь будем идти не к `head`, а к минимальному общему предку обеих вершин.

Тогда алгоритм `send` получится следующим:

1) Делаем `send` также как он сделан в `SimpleSplayListNet`, только вместо походов к голове, идём к наибольшему общему предку s_i и f_i.

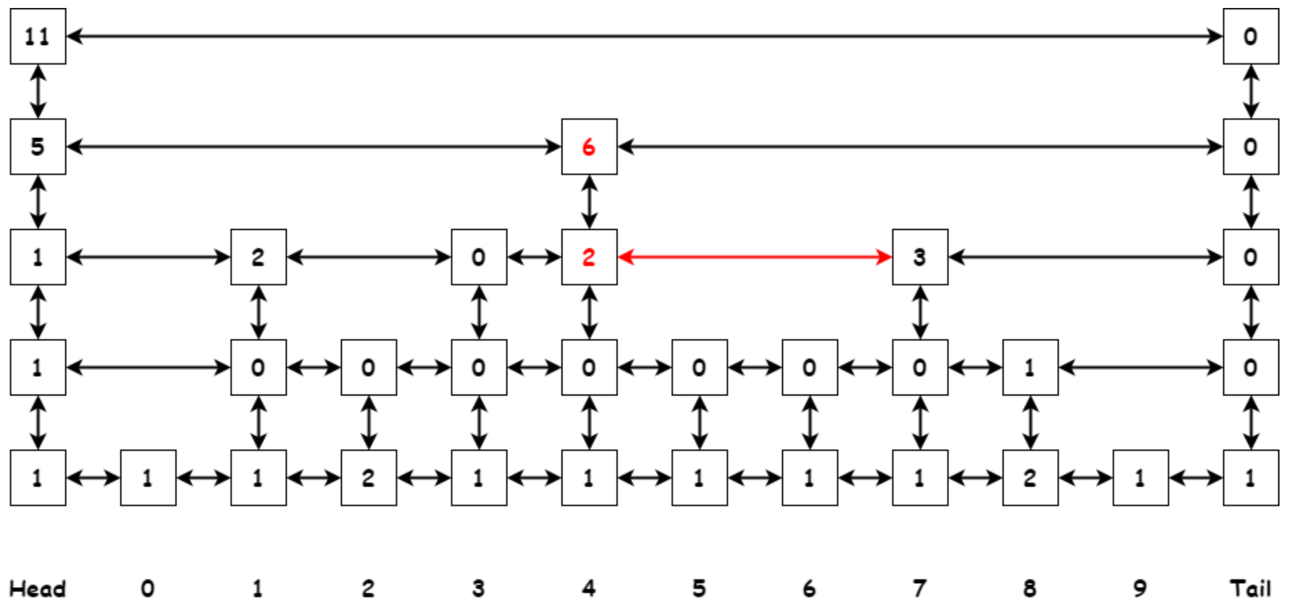


Рисунок 8 – Спуск вершины 6 и переход к 4

2) Идём от наибольшего общего предка к `head`, попутно обновляя $hits$ всех проходимых вершин, однако увеличиваем мы не на 1, как это происходит в `SplayList`, а на 2, так как были изменены уже 2 вершины.

Так как в этом алгоритме происходит обновление не на 1, а на 2, теряется инвариант, что после каждого прохода не остаётся вершин, которые удовлетворяют условию подъёма. Для решения этой проблемы увеличим счётчик каждой вершины до 2, обратившись к ней как в `SplayList`.

Лемма 6. После операции `send` нет вершин, удовлетворяющих условию подъёма.

Доказательство. Доказательство построено на основе индукции по количеству операций `send`

1) *База индукции*. Согласно Лемме 1 [18] в изначальной сети S нет вершин, удовлетворяющих условию подъёма.
2) *Индукционный переход*. Предположим, что после определённого количества операций нет вершин, которые удовлетворяют условию подъёма. Так как $\forall v \in S \; v.selfHits \geqslant 2$, то для набора нод $u_1, u_2,...,u_k$ находящихся на одном уровне h $\Sigma_{j=i,1\leqslant i<k}^{k} hits(u_j,h) \geqslant \Sigma_{j=i+1}^{k} hits(u_j,h) + 2$. После операции `send` для всех u, $hits(u,h)$ изменяется не более чем на 2, причём на для вершин из набора $u_1, u_2, ..., u_k$ только одна из них может увеличить $hits$ на 2 (та в чьём поддереве находились s_i и f_i) или только 2 вершины, что увеличили свои $hits$ на 1. В любом случае, только u_1

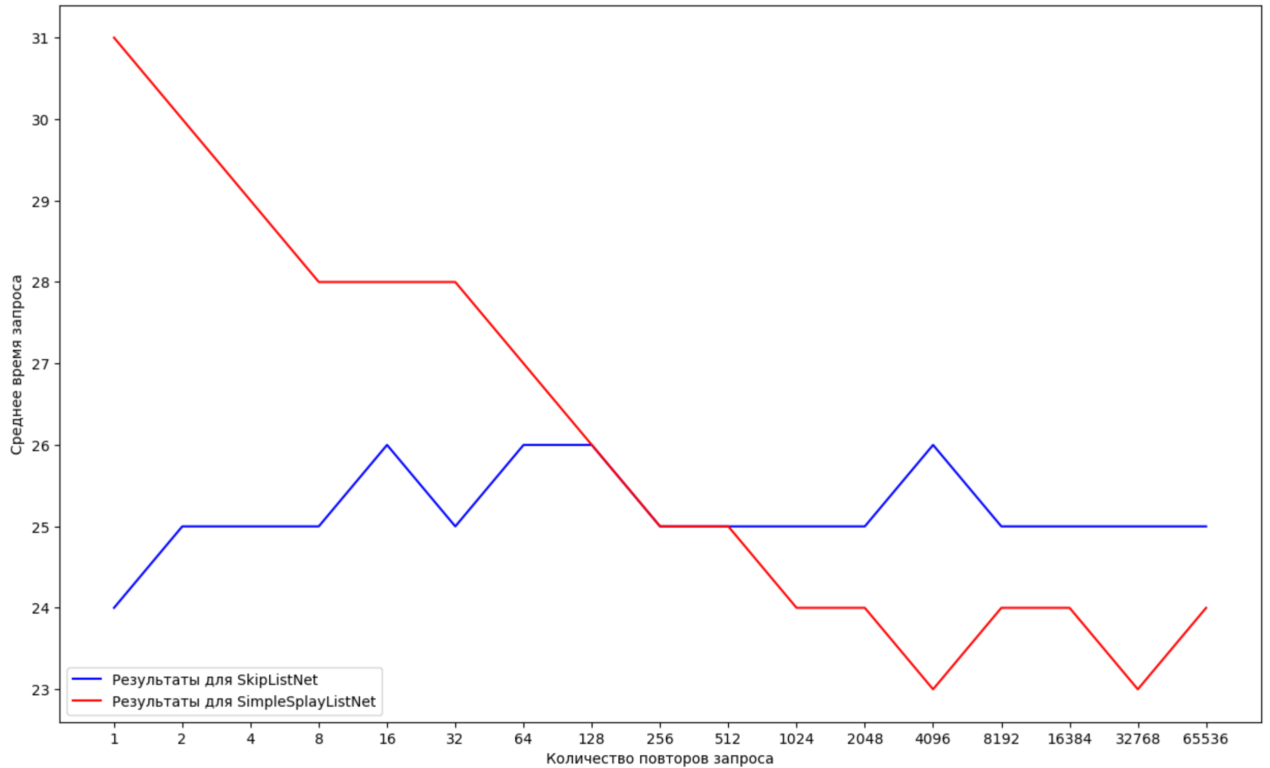


Рисунок 9 – Сравнение SimpleSplayListNet с SkipListNet

может удовлетворять условию подъёма, а так как она будет посещена на обратном пути при походе к `head` или к наибольшему общему предку, то она будет поднята на новый уровень.

Теорема 7. Если во время $send(s_i, f_i)$ было сделано d спусков, то общий пройденный путь составляет не более $2 \cdot d + 4 \cdot y_1 + 8 \cdot y_2$, где $y_1 = 3 + \log \frac{m}{s_i.selfHits}$, $y_2 = 3 + \log \frac{m}{f_i.selfHits}$.

Доказательство. Количество посещённых уровней при проходе от s_i до наибольшего общего предка не меньше, чем до `head`, что согласно лемме 2 в [18] $\leqslant 3 + \log \frac{m}{s_i.selfHits}$. Аналогично, от наибольшего общего предка до f_i (нужно также помнить, что от f_i мы уже идём до `head`). Значит, суммарно, мы посетим не более чем $y_1 + 2 \cdot y_2$ уровней, при этом из леммы 3 [18] на каждом уровне мы посещаем не более чем 4 вершины, неудовлетворяющих условию спуска $\Rightarrow$ мы посещаем не более чем $4 \cdot y_1 + 8 \cdot y_2$ вершин неудовлетворяющих условию спуска.

При проходе от наибольшего общего предка, до f_i мы не изменяем вершины, которые удовлетворяют условию спуску, поэтому они будут посещены два раза (обозначим их количество за d_1). При этом, это единственный раз, когда мы можем посетить вершину, удовлетворяющую условию спуска дважды.

Резюмируя всё выше сказанное, максимальный путь может составлять $4 \cdot y_1 + 8 \cdot y_2 + 2 \cdot d_1 + (d - d_1) \leqslant 4 \cdot y_1 + 8 \cdot y_2 + 2 \cdot d$

Замечание 3. Исходя из доказанного в теореме 7, можно сказать, что операция `send` выполняется не дольше, чем $2 \cdot d + 8 \cdot (y_1 + y_2)$, как следствие можно аналогично теореме 6 из [18] доказать, что амортизированное время равно $\mathcal{O}(\log \frac{m}{s_i.selfHits} + \log \frac{m}{f_i.selfHits})$

Рассмотрим следующий пример.

1) Предположим, что у нас есть структура `TreeSplayListNet` S (Рисунок 10). Давайте пошлём запрос из вершины 5 в вершину 8.
2) Заметим, что вершина 4 является наибольшим общим предком вершин 5 и 8 $\Rightarrow$ мы должны добраться от $s_i = 5$ до 4, обновляя все посещённые по пути вершины на 1 (Рисунок 11).
3) Когда мы поняли, что дошли до наибольшего общего предка ($4.next[4.topLevel] = tail$), мы идём до f_i, чтобы доставить запрос (Рисунок 12).
4) Дальнейшее обновление до наибольшего общего предка схоже с тем, как оно делается в `SimpleSplayListNet` (Рисунки 13, 14, 15). Идём увеличивая hits на 1 и обновляя вершины, если они удовлетворяют условию обновления.
5) После достижения 4 нужно обновить и всю остальную систему. Так как у нас были обновлены 2 вершины, то $hits$ всех последующих вершин должны быть увеличены на 2 (как например $hits(head, 5)$ с 20 увеличено до 22) (Рисунок 16).

Как можно увидеть из рисунка 17, `TreeSplayListNet` показывает себя гораздо лучше при частом повторении запроса, а как следствие при большом трафике. Однако он гораздо хуже чем даже `SimpleSplayListNet` на малом повторении запроса, что достаточно странно, учитывая, что выглядит он оптимальнее. Вероятно это происходит из-за того, что $selfHits$ каждой вершины уже 2, а значит подняться им на следующий уровень становится труднее. Попробуем отказаться от инварианта и оставим изначальный $selfHits$ равным 1.

Как можно видеть из графиков (Рисунок 18), приведённых ниже ускоряет время работы, когда запрос посылается редко, однако при увеличении повторов алгоритм становится идентичен по скорости обыкновенному

Рисунок 10 – Изначально данный TreeSplayListNet

Рисунок 11 – Поход от 5 до наибольшего общего предка 5 и 8

`TreeSplayListNet`. Так как наш алгоритм нужен для сетей с высоким трафиком, то смысла жертвовать инвариантом нет.

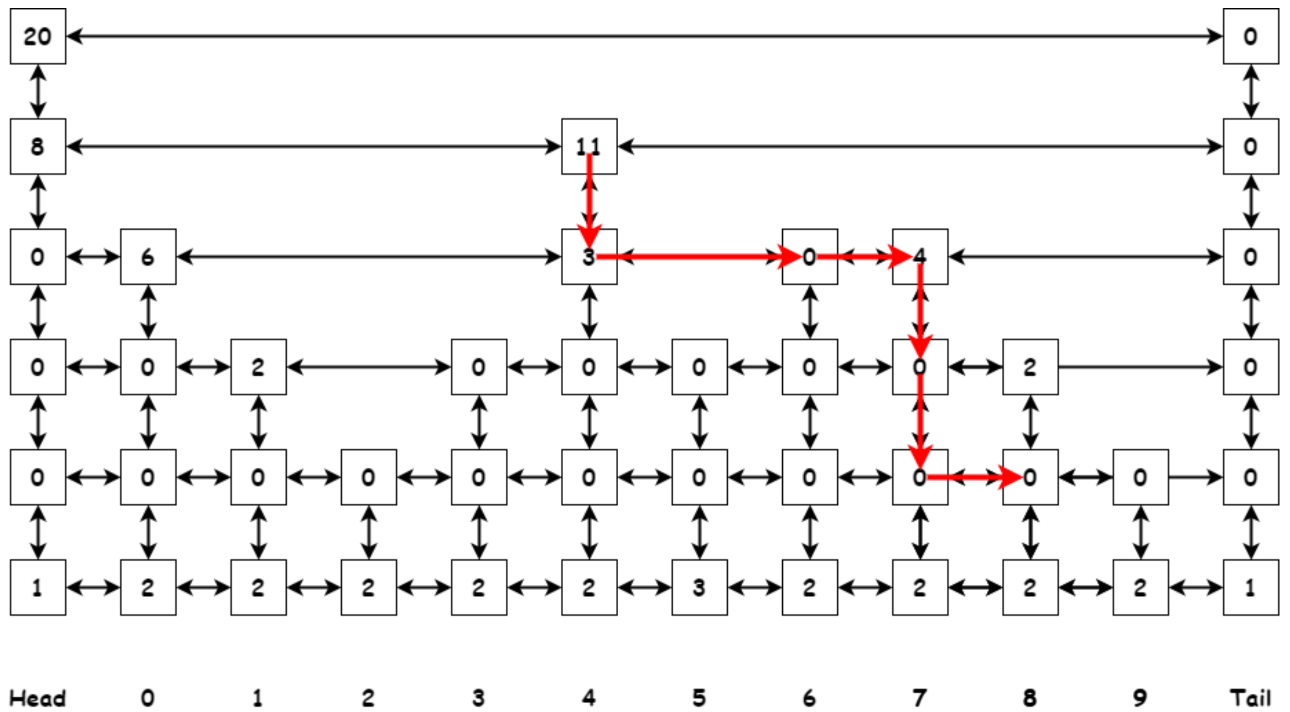


Рисунок 12 – Поиск 8

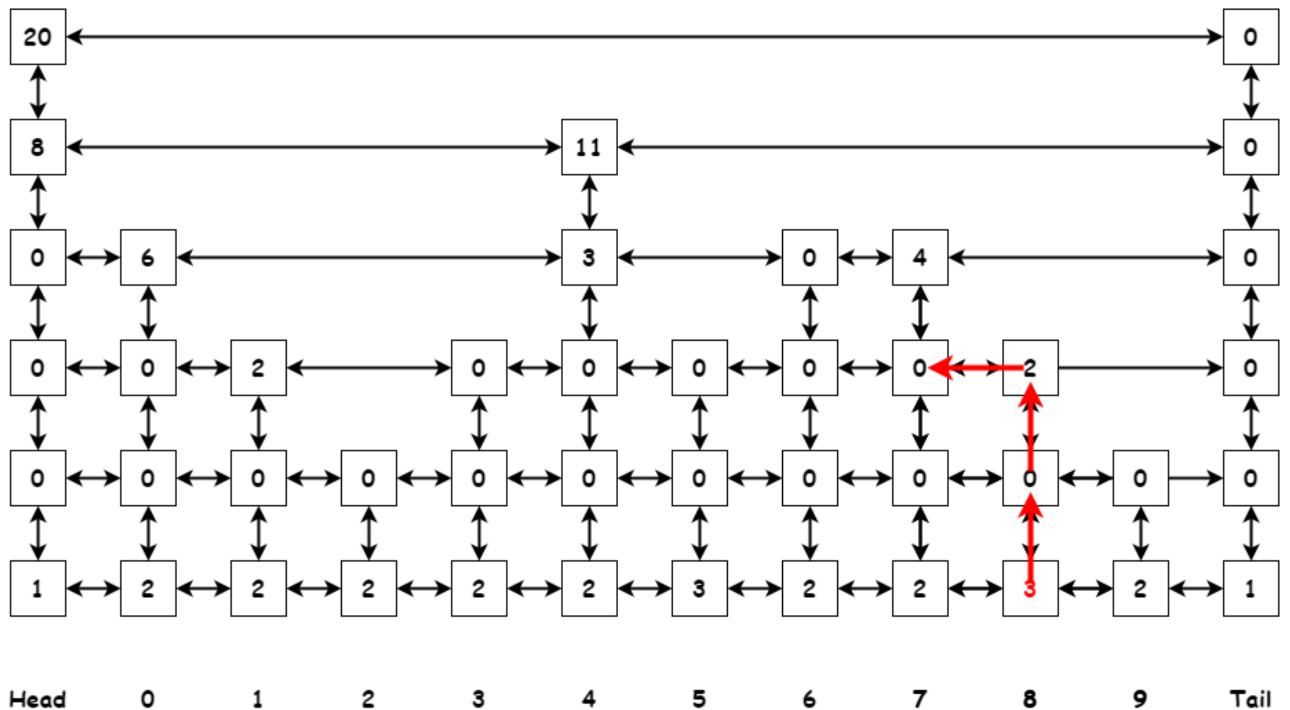


Рисунок 13 – От 8 к 7 в TreeSplayListNet

2.3. Выводы к главе 2

В данной главе были разработаны и реализованы структуры на основе `SplayList`, а именно:

1) `SimpleSplayListNet` - простейшая реализация адаптивной сети, идущая при каждом запросе в `head`. Сильно проигрывает статическим

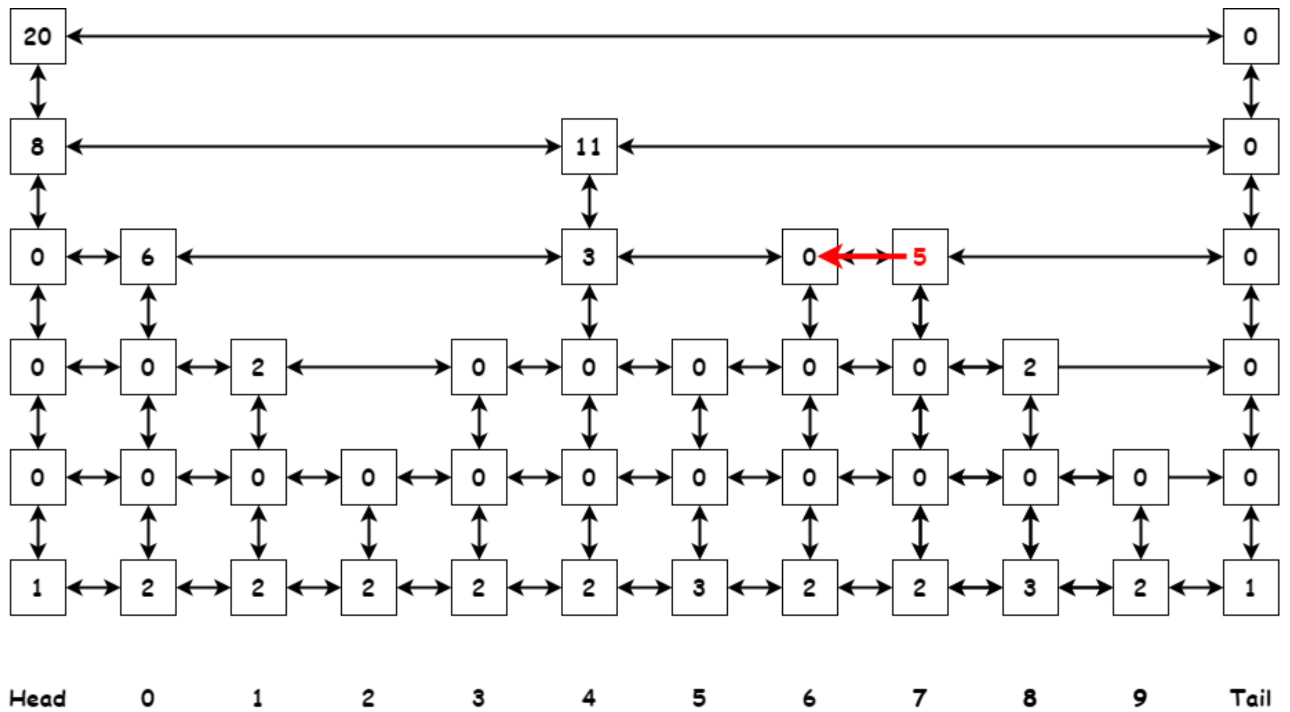


Рисунок 14 – От 7 к 6 в TreeSplayListNet

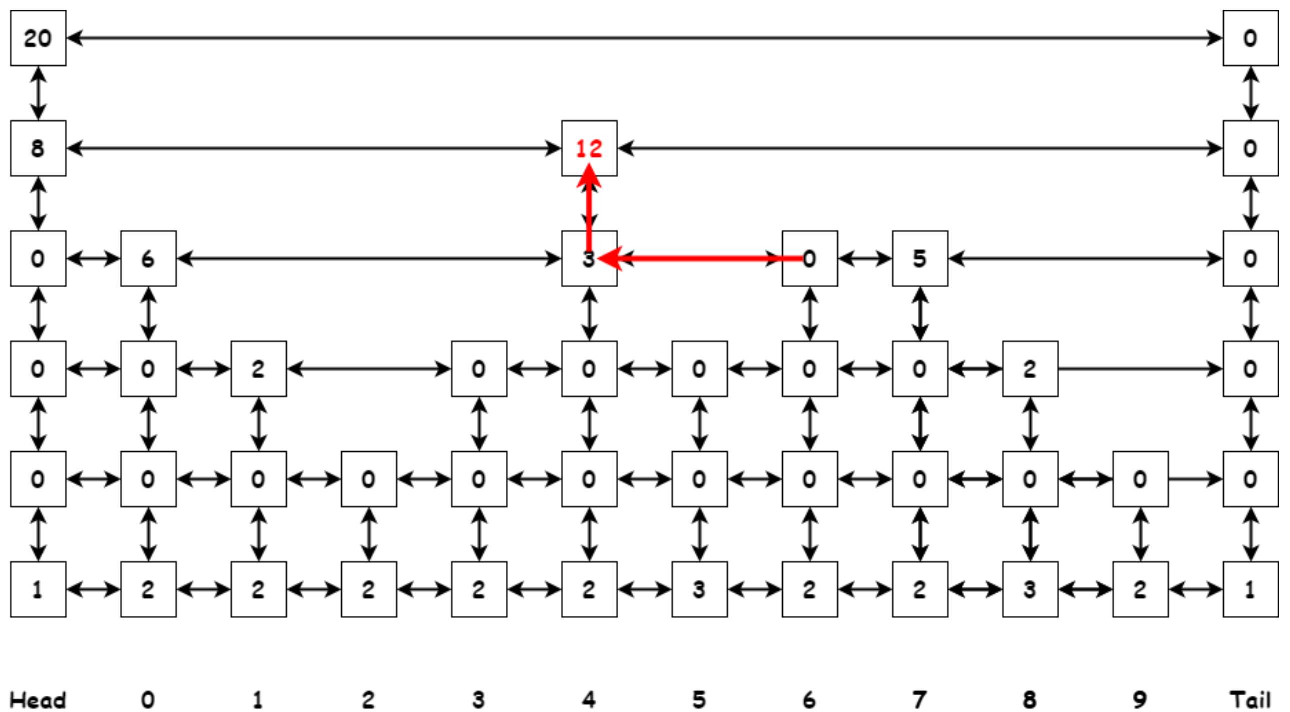


Рисунок 15 – От 6 до наибольшего общего предка

структурам на редко повторяющихся запросах, но дающая выигрыш до 8% на часто повторяемых запросах.

2) `TreeSplayListNet` - версия адаптивной сети, которая идёт до наибольшего общего предка. Из-за сохранения инварианта `TreeSplayListNet` хуже на 27%, чем статические сети на ред-

Рисунок 16 – От 4 до head

Рисунок 17 – Сравнение TreeSplayListNet с остальными алгоритмами

ких запросах, однако даёт выигрыш до 23% на частых запросах. `TreeSplayListNet` не поддерживающий инвариант на редких запросах по скорости почти не отличается от статических сетей (разница

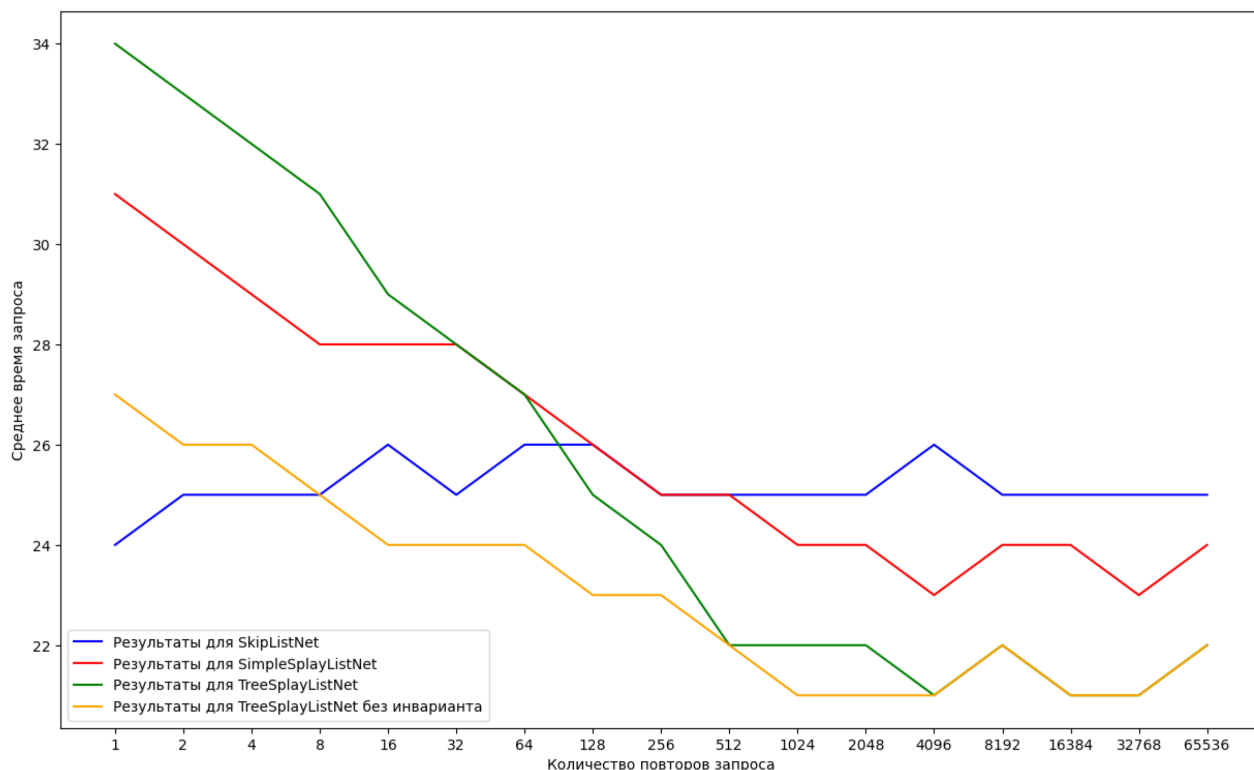


Рисунок 18 – Сравнение TreeSplayListNet без инварианта с остальными алгоритмами

всего 12%), но при частом повторении запроса работает также как и обычный `TreeSplayListNet`.

3) Также было доказано, что аммортизированное время работы алгоритмов $\mathcal{O}(\log \frac{m}{s_i.selfHits} + \log \frac{m}{f_i.selfHits})$, что показывает, что скорость алгоритма достаточно быстра и при частом обращении к вершине скорость алгоритма будет только расти.

ГЛАВА 3. СТРУКТУРЫ, НЕ ИДУЩИЕ НАЗАД ПРИ ПОИСКЕ

Основной проблемой всех алгоритмов на основе `SplayList` является то, что они все идут к `head` для обновления счётчиков, а как следствие, проходится дополнительный путь и создаётся дополнительная нагрузка на вершину, что может стать узким местом в нашей структуре. Попробуем увеличить число связей вершины, но ускорить время работы.

3.1. LeftRightSplayNet

Предположим, что у нас есть список вершин $[0, 1, 2,...,n]$. Разобъём список на две половины и построим на обеих `SplayList`.

Первый `SplayList` будет состоять из вершин $[0, 1, ..., \frac{n}{2}]$, причём порядок на нём будет от большего к меньшему, то есть $i.next[0] = i - 1$. Начало первого листа назовём `left middle`.

Второй будет состоять из вершин $[\frac{n}{2} + 1,...,n]$, порядок на нём будет идти от меньшего к большему. Начало второго листа назовём `right middle`. Листы будут соединены `left middle` к `right middle`.

Также будем хранить для `LeftRightSplayNet` значение $middle$ — значение, по которому мы будем понимать в какой половине лежит вершина (если $value \leqslant middle$, то значение в левой половине, если $value > middle$, то в правой). Очевидно, что запрос можно будет послать $\Leftrightarrow$ $s_i \leqslant middle$ и $f_i > middle$.

Операция $send(4, 10)$ в структуре на 12 вершин будет выглядеть следующим образом:

1) В изначальной структуре (Рисунок 19) увеличиваем число запросов M на 1 и $MaxLevel$, если $\log M \geqslant MaxLevel$.
2) Идём из s_i до `left middle` попутно обновляя все значения. Пример на рисунке 20.
3) Переходим из левой половины в правую в поисках f_i. При этом мы можем обновлять все значения не идя к голове, также как это сделано в `Forward Rebalancing` в [18]. Пример на рисунке 21.

3.2. ParentChildNet

Однако, пересылки из одной половины нам не достаточно, так как большое количество запросов останутся просто без внимания. Построим `LeftRightSplayNet` на обеих половинах, потом на их половинах, и так

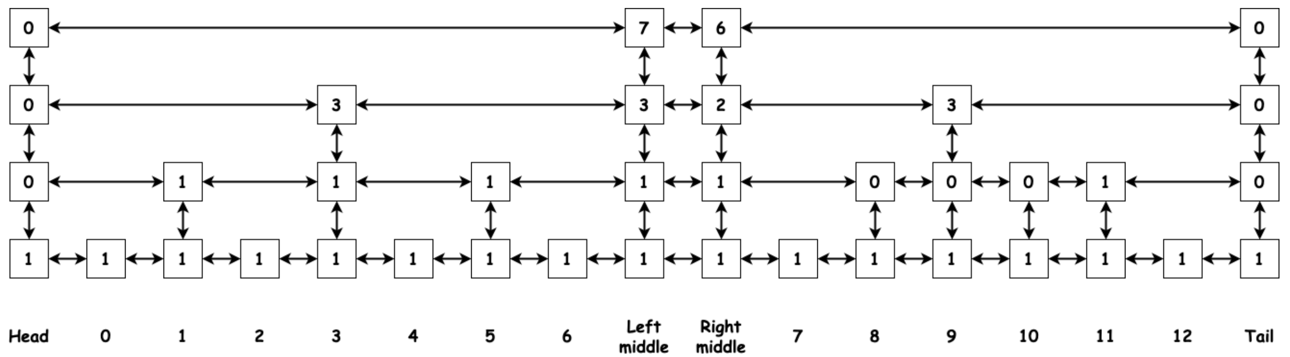


Рисунок 19 – Изначально имеющийся LeftRightSplayNet

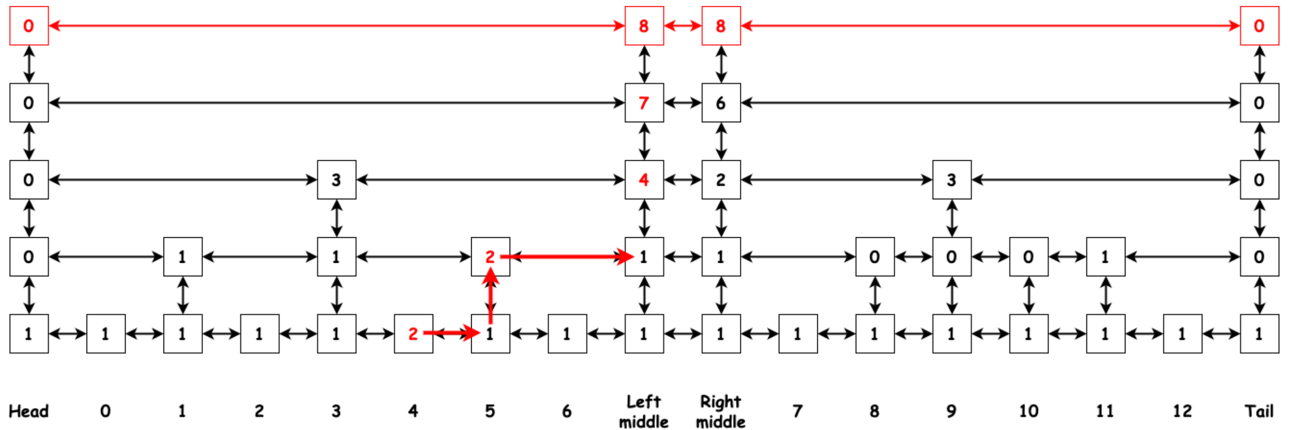


Рисунок 20 – Действия в левой половине

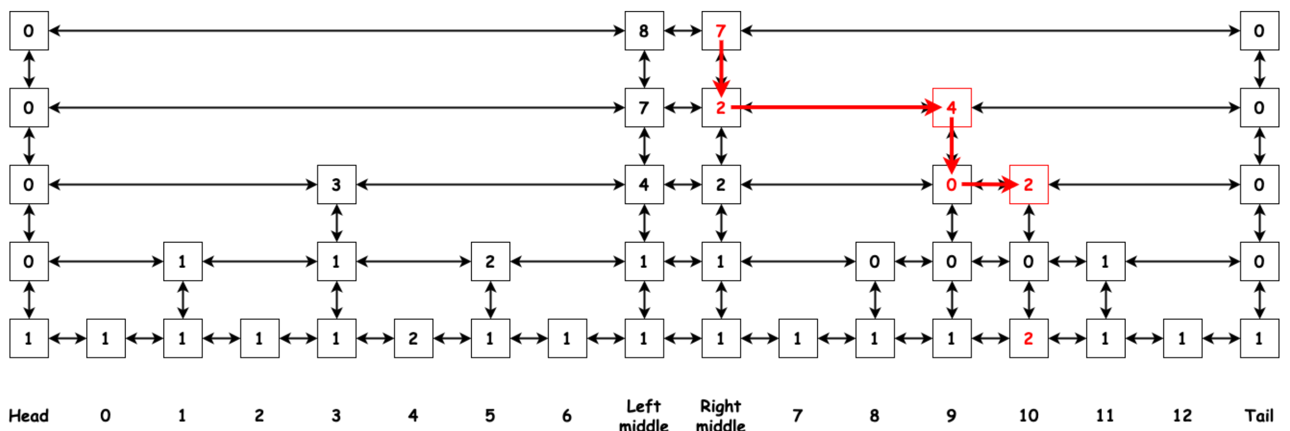


Рисунок 21 – Действия в правой половине

далее рекурсивно. Когда интервал станет достаточно малым, воспользуемся структурами, которые уже были описаны выше, как например `SkipListNet` или `TreeSplayListNet`. Рисунок 22.

Саму `routing` структуру легче будет представлять как дерево, состоящее из `SearchSplayNode`, где каждая вершина либо лист, либо имеет двоих детей. Каждая вершина хранит в себе сеть, которая может исполнять запросы. Как показано на рисунке 23.

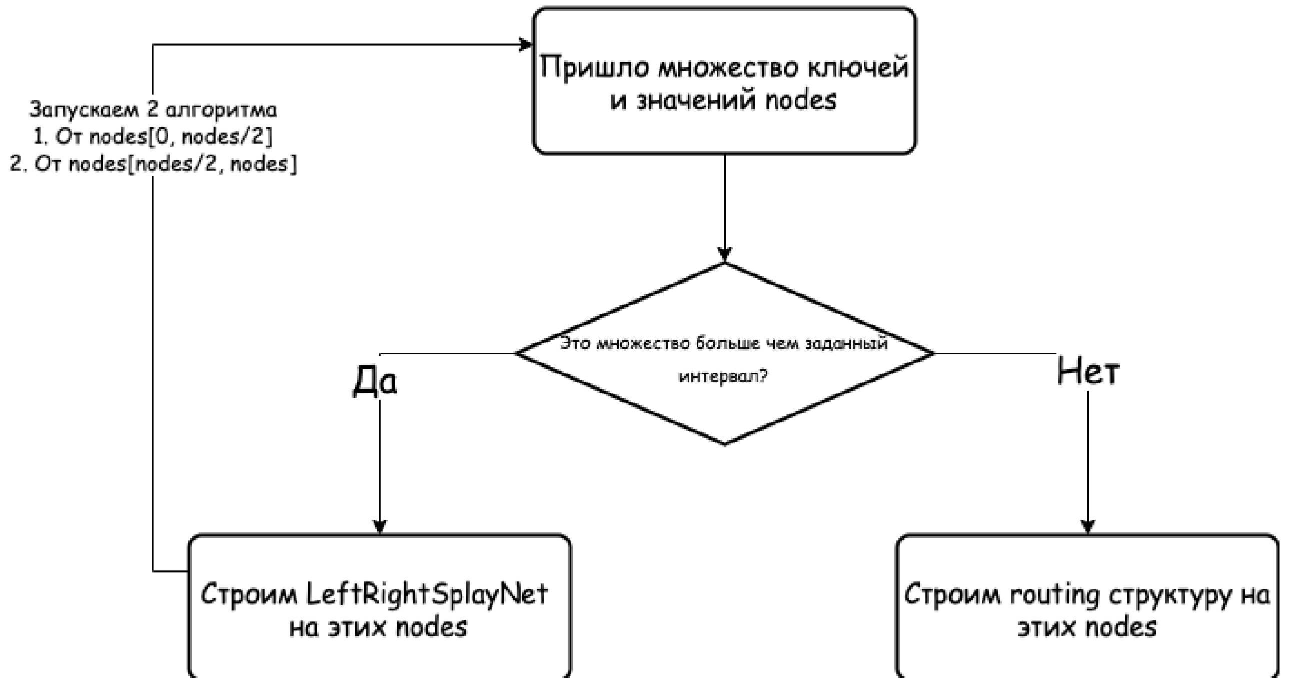


Рисунок 22 – Создание ParentChildNet

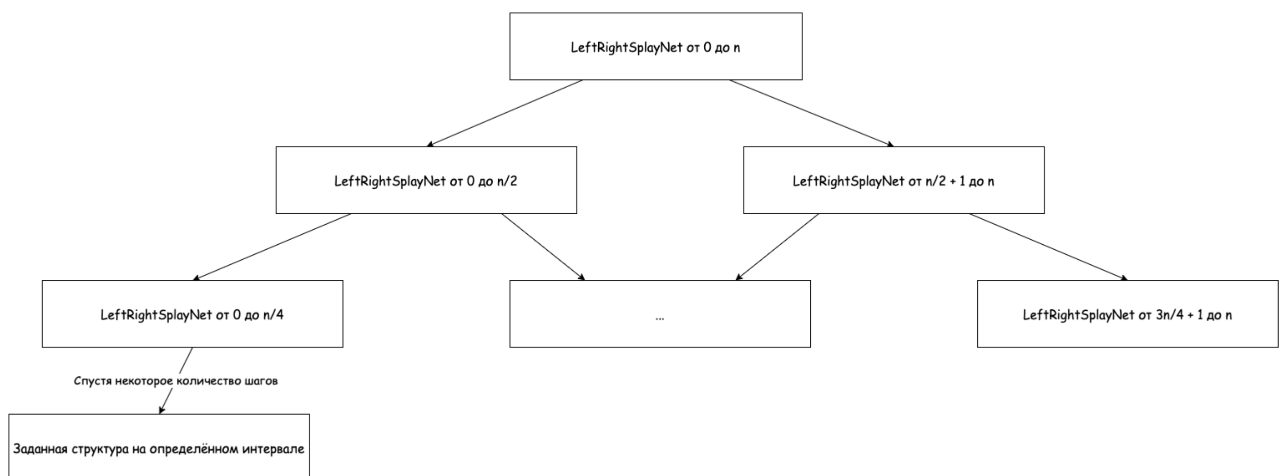


Рисунок 23 – Структура ParentChildNet

Сама SearchSplayNode имеет следующие поля:

1) middle – значение, по которому мы определяем можно ли в данной вершине сделать этот запрос или нужно идти в детей. Листья хранят null в этом поле.
2) net — LeftRightSplayNet в которой исполняется запрос. Листья хранят null в этом поле.
3) leaf — сеть, которая находится в листах. Не листья хранят null в это поле.
4) left — левый ребёнок вершины, null в листьях.
5) right — правый ребёнок вершины, null в листьях.

Код SearchSplayNode представлен в листинге 9

Листинг 9 – Код SearchSplayNode

```
class SearchSplayNode<K : Comparable<K>, V, N : Node<K, V>>(
    val middle: K?,
    val net: LeftRightSplayNet<K, V>?,
    val leaf: Net<K, V, N>?,
    val left: SearchSplayNode<K, V, N>?,
    val right: SearchSplayNode<K, V, N>?)
```

Однако, так как, что в родительской вершине, что в её детях находятся одни и те же сервера, то структура ParentChildSplayNode вершины, используемой в LeftRightSplayNet выглядит как:

1) key — ключ, по которому мы находим сервер;
2) value — значение, хранящееся на сервере;
3) link — ссылка на значение в нижестоящих вершинах;
4) selfHits — то же самое, что и в SplayNode
5) next — то же самое, что и в SplayNode
6) previous — то же самое, что и в SplayNode
7) hits — то же самое, что и в SplayNode
8) topLevel — то же самое, что и в SplayNode

Пример кода в листинге 10:

Листинг 10 – Код ParentChildSplayNode

```
class ParentChildSplayNode<K : Comparable<K>, V>(
    val key: K?,
    var value: V?,
```

```
var next: MutableList<N> = ArrayList(),
var previous: MutableList<SplayNode<K, V>> =
  ArrayList(),
var topLevel: Int = 0,
var selfHits: Int = 0,
var hits: MutableList<Int> = ArrayList(),
var link: Node<K, V>) : Node<K, V>()
```

Итоговый алгоритм посылки запроса выглядит как поиск первой сети, которая может исполнить запрос. Подробнее на рисунке 24 и в листинге 11.

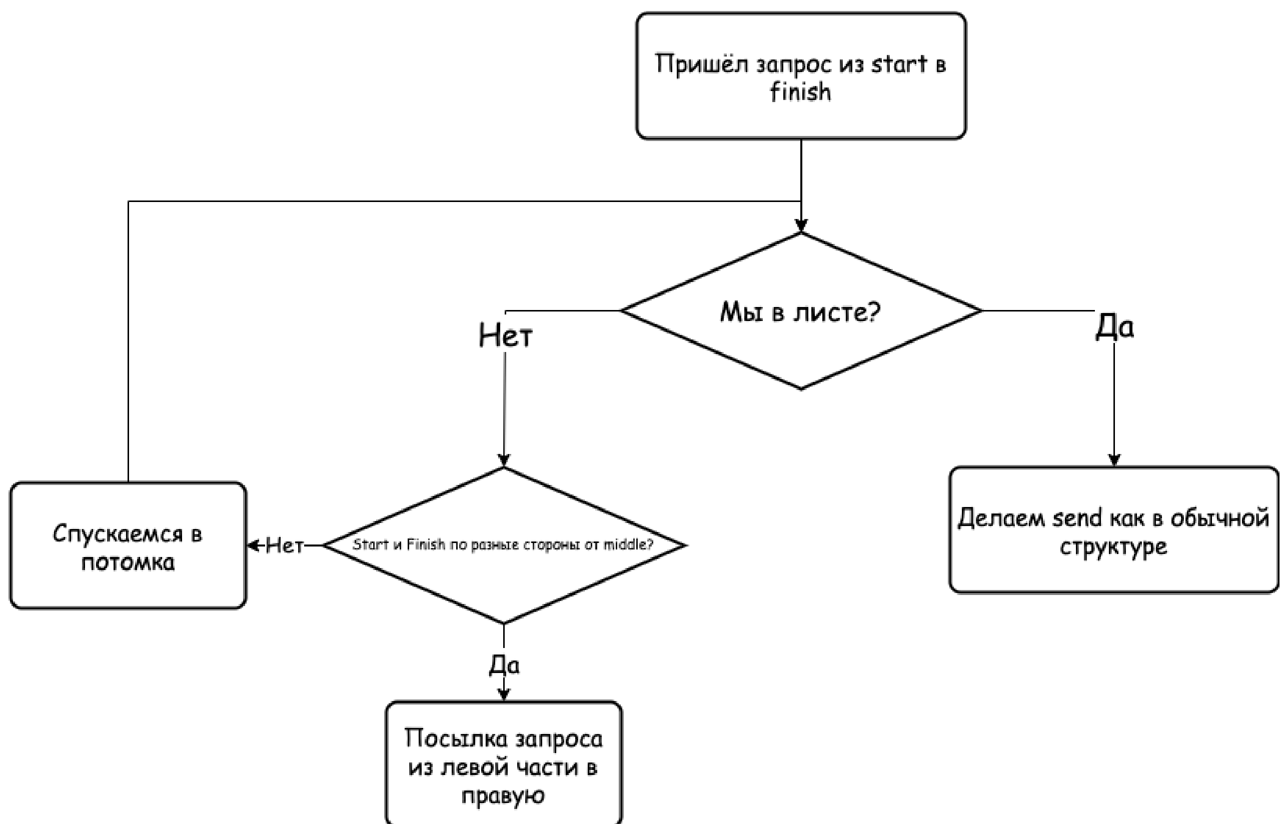


Рисунок 24 – Запрос в ParentChildNet

Листинг 11 – Send в ParentChildNet

```
fun send(start: ParentChildSplayNode<K, V>, finish: K) {
    var current = root
    var begin = start
    while (!current.isLeaf() && !(begin.key < current.middle && finish >=
        current.middle)) {
        current = if (current.middle <= begin.key) {
            current.right
        } else {
            current.left
```

```
        }
        begin = begin.link
    }
    current.send(begin, finish)
}
```

Как показывает рисунок 25, `ParentChildNet` даёт достаточно большой выигрыш вне зависимости от того какая структура используется в её листьях.

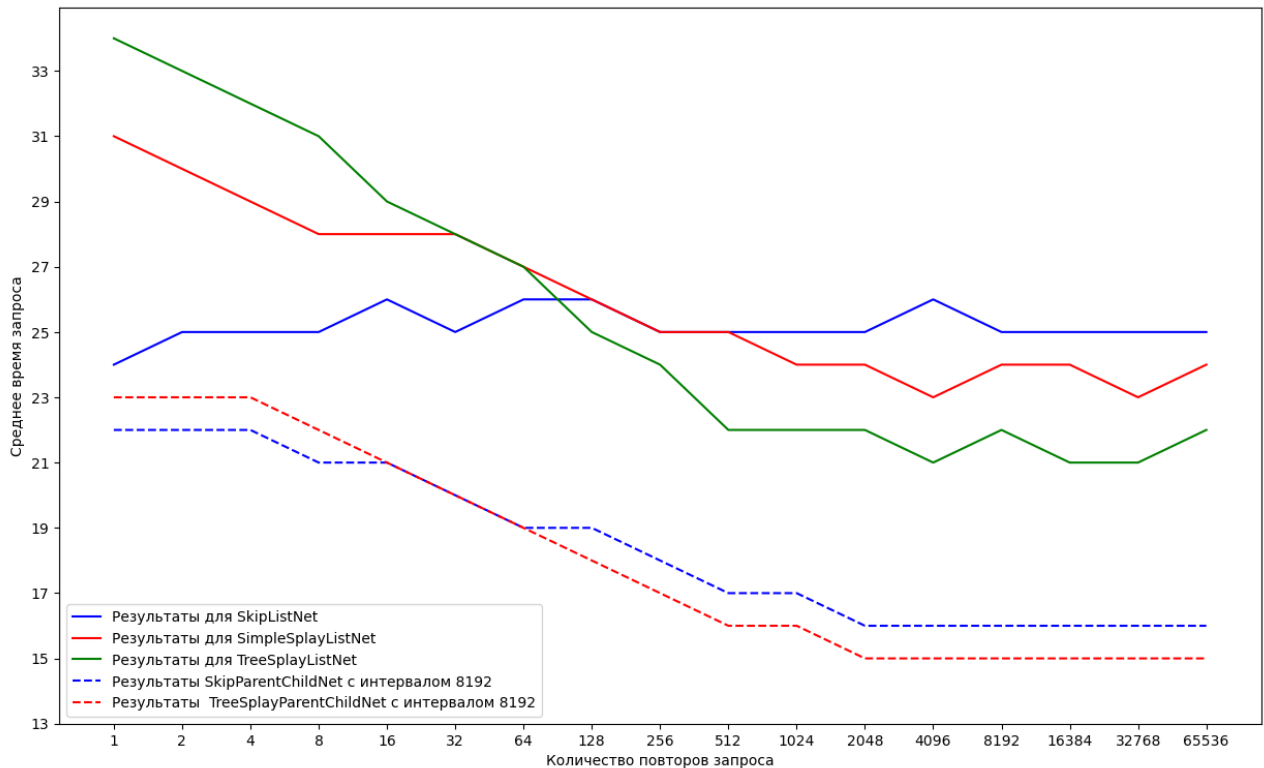


Рисунок 25 – Сравнение ParentChildNet

Не менее важным вопросом является каким определять порог, после которого мы прекращаем строить `LeftRightSplayNet` и создаём лист. Делать много соединений может оказаться дорого, а также привести к нежелательной нагрузке, поэтому их число хотелось бы уменьшить. С другой стороны, если сделать порог слишком большим, мы можем потерять в производительности, чего бы нам делать также не хотелось. Как мы знаем из [18], только добавление новой вершины в `ParentChildNet` со `LeftRightSplayNet` длины l может добавить серверу до $\log l$ соединений. Попробуем же разобраться как длина порога влияет на скорость работы алгорима.

Как можно видеть из графиков 26 27 и таблиц 1 2 для `ParentChildNet`, увеличение порога хоть и влечёт к увеличению вре-

мени работы, но зависимость явно не серьёзная; так скорость работы при пороге 512 и 16 почти одинаково, хотя при интервале 16 мы можем получить до 255 дополнительных соединений на сервер.

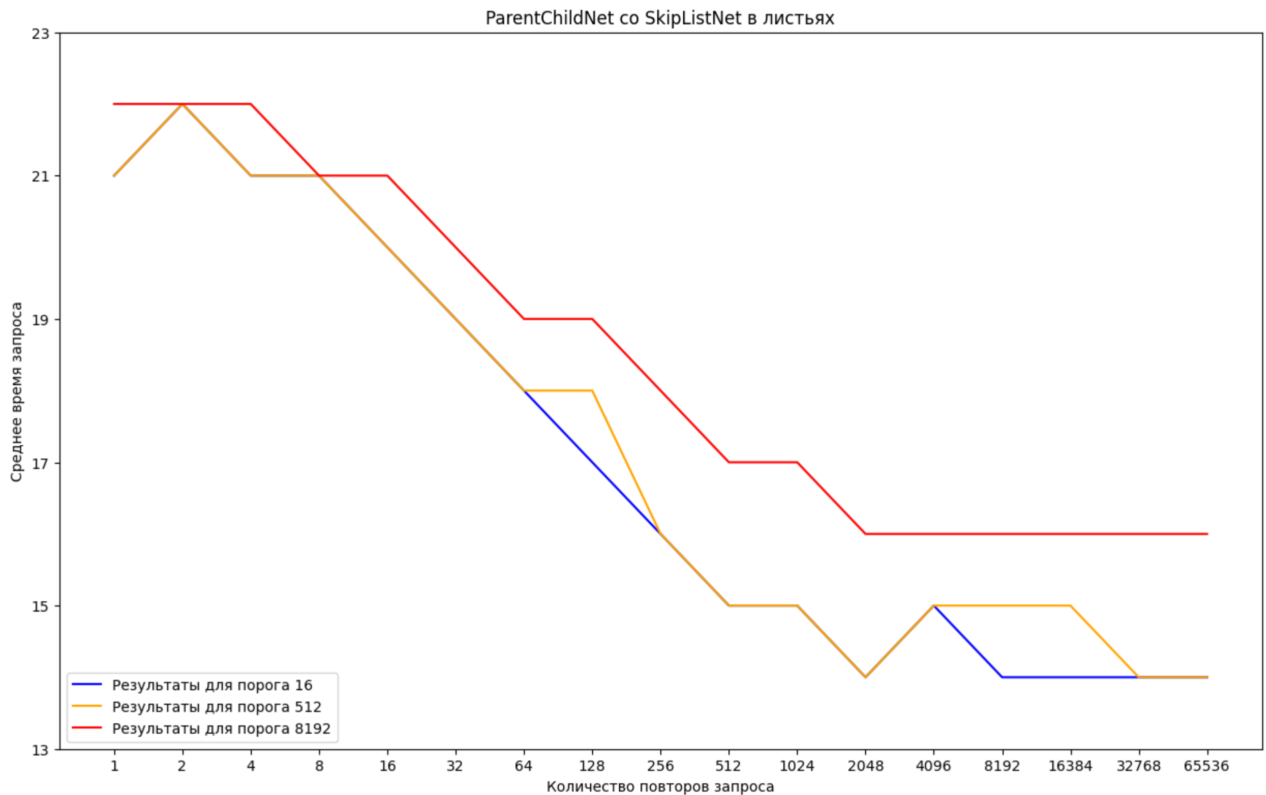


Рисунок 26 – Сравнение ParentChildNet со SkipListNet в листьях

Таблица 1 – Таблица числав шагов `ParentChildNet` с `SkipListNet` в листьях. В верхней строке указан log от числа шагов. В первом столбце log от порога.

–	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
4	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
5	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
6	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
7	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
8	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
9	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*18*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
10	*21*	*22*	*22*	*21*	*20*	*19*	*18*	*18*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
11	*22*	*22*	*22*	*21*	*20*	*19*	*19*	*18*	*16*	*16*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
12	*22*	*22*	*22*	*21*	*20*	*19*	*19*	*18*	*17*	*16*	*15*	*15*	*15*	*15*	*15*	*14*	*15*
13	*23*	*23*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*16*	*15*	*15*	*15*	*15*	*15*	*15*

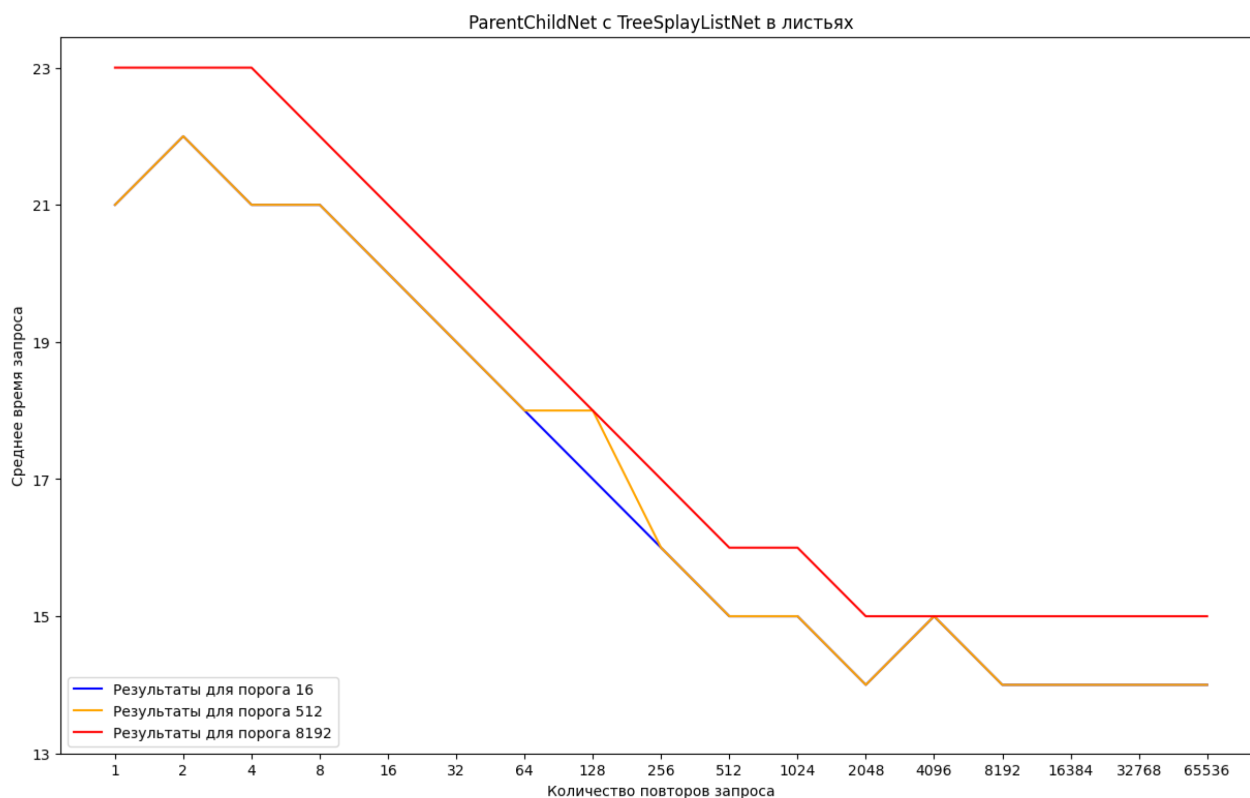


Рисунок 27 – Сравнение ParentChildNet с TreeSplayListNet в листьях

Таблица 2 – Таблица числа шагов `ParentChildNet` с `TreeSplayListNet` в листьях. В верхней строке указан log от числа шагов. В первом столбце log от порога.

–	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
4	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
5	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
6	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
7	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
8	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
9	*21*	*22*	*21*	*21*	*20*	*19*	*18*	*18*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
10	*21*	*22*	*22*	*21*	*20*	*19*	*18*	*18*	*16*	*15*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
11	*22*	*22*	*22*	*21*	*20*	*19*	*19*	*18*	*16*	*16*	*15*	*14*	*15*	*14*	*14*	*14*	*14*
12	*22*	*22*	*22*	*21*	*20*	*19*	*19*	*18*	*17*	*16*	*15*	*15*	*15*	*15*	*15*	*14*	*15*
13	*23*	*23*	*23*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*16*	*15*	*15*	*15*	*15*	*15*	*15*

3.3. Выводы к главе 3

Поняв, что в структурах, представленных в главе 2 все алгоритмы идут в `head`, чтобы синхронизировать обновления и нагружая тем самым изначальную вершину, было принято решение о разработке структур, которые не используют какую-то одну вершину во всех запросах.

Были разработаны `LeftRightSplayNet` - подструктура умеющая пересылать запросы из одной своей половины в другую, и `ParentChildNet` - дерево в вершинах которого находятся `LeftRightSplayNet`, а в листьях структуры, описанные в главе 2. Такой подход помогает перераспределять нагрузку, потому что в зависимости от s_i и f_i основная нагрузка идёт на разные вершины.

Что же касается скорости, то структура получает выигрыш по скорости до 62.5% при `SimpleSplayListNet` в листьях и до 73% при `TreeSplayListNet` в листьях по сравнию со статическими структурами. Также было проведенно сравнение скорости в зависимости от порога при котором создаётся лист. Как показывают графики зависимость между порогом и скоростью не линейная, так скорость при пороге в 16 и в 512 почти не различается для обоих вариаций. Порог в 8192 проигрывае на 14% и 7% при `SkipListNet` и `TreeSplayListNet` в листьях соответственно.

Как следствие структура показывает более чем удовлетворительные результаты, причём рекомендуется выбирать порог перехода в лист равным 512, как тот, что не проигрывает по скорости, но требует гораздо меньше соединений от вершины.

ГЛАВА 4. ВЕРОЯТНОСТНЫЕ СТРУКТУРЫ

При работе с адаптивной структурой может оказаться так, что частое обновление структуры является препятствием к высокой производительности. Мы можем иметь два часто общающихся региона и как следствие мы бы хотели, чтобы вершины, соединяющие их находились как можно выше, однако случайные запросы могут всё испортить и добавить дополнителные препятствия на пути или еще хуже — опустить использующиеся вершины.

4.1. ProbabilitySimpleSplayListNet

Первым мы переведём `SimpleSplayListNet` на вероятностные рельсы. Фактически сам алгоритм будет таким же как в оригинальной структуре. Основным отличием, будет лишь то, что запрос $send(s_i, f_i)$ будет выполняться как в оригинале, лишь с вероятностью p, в остальных случаях мы будем идти в `head` из s_i и из `head` в f_i, но не обновляя счётчики.

Как показывает график 28 такой подход более чем оправдан, мы получаем выигрыш уже при вероятности обновления $\frac{1}{2}$. Наиболее вероятной причиной такого результата является, что мы не проводим обновления по достижении f_i, а как следствие мы уже экономим четверть времени от запроса. Полный отчёт приведён в рисунке 28 и в таблице 3.

Таблица 3 – Таблица числа шагов `ProbabilitySimpleSplayListNet`. В верхней строке указан log от числа шагов. Вероятность обновления

–	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
$\frac{1}{2}$	*25*	*25*	*25*	*24*	*23*	*22*	*22*	*22*	*21*	*20*	*19*	*19*	*19*	*20*	*19*	*19*	*20*
$\frac{1}{3}$	*23*	*23*	*23*	*23*	*22*	*21*	*21*	*21*	*20*	*19*	*19*	*18*	*18*	*18*	*17*	*18*	*17*
$\frac{1}{5}$	*22*	*21*	*21*	*21*	*21*	*20*	*19*	*19*	*19*	*19*	*17*	*17*	*16*	*16*	*17*	*16*	*16*
$\frac{1}{10}$	*20*	*20*	*20*	*20*	*20*	*20*	*19*	*18*	*18*	*18*	*17*	*17*	*16*	*16*	*16*	*15*	*16*
$\frac{1}{20}$	*20*	*20*	*20*	*20*	*19*	*20*	*19*	*18*	*18*	*18*	*18*	*17*	*16*	*16*	*16*	*15*	*15*
$\frac{1}{50}$	*20*	*19*	*19*	*19*	*19*	*19*	*19*	*19*	*18*	*18*	*17*	*18*	*17*	*16*	*16*	*15*	*14*
$\frac{1}{100}$	*19*	*19*	*19*	*19*	*19*	*19*	*19*	*19*	*19*	*18*	*18*	*17*	*17*	*17*	*16*	*15*	*15*

4.2. ProbabilityTreeSplayListNet

Попробуем такой же трюк с `TreeSplayListNet`, однако будем идти уже до наибольшего предка, а от него до f_i. Как показывают результаты на рисунке 30 и таблице 3 `ProbabilityTreeSplayListNet` также как и `TreeSplayListNet` проигрывает аналогичным `SimpleSplayListNet`

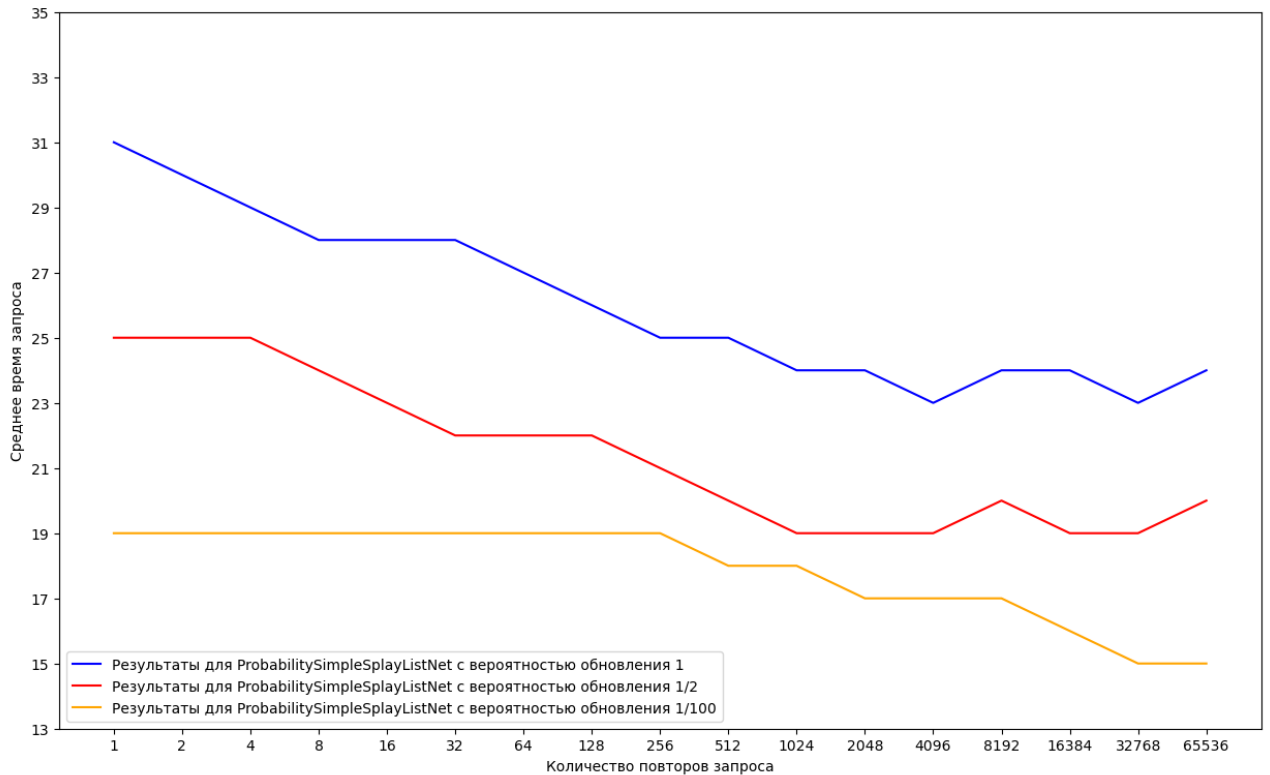


Рисунок 28 – Сравнение ProbabilitySimpleSplayListNet с SimpleSplayListNet

на малом повторе запросов, но получает выигрыш при частом повторении запроса.

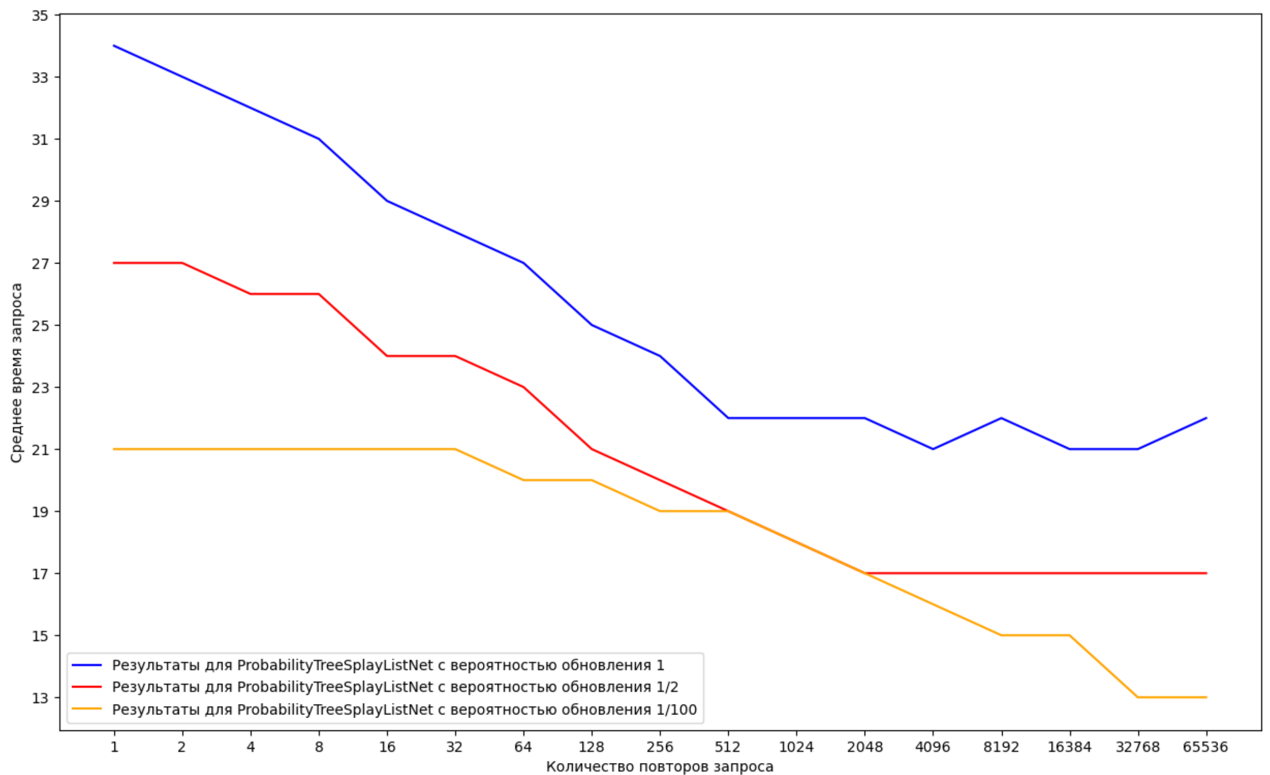


Рисунок 29 – Сравнение ProbabilityTreeSplayListNet с TreeSplayListNet

Таблица 4 – Таблица числа шагов `ProbabilityTreeSplayListNet`. В верхней строке указан log от числа шагов. Вероятность обновления

–	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
$\frac{1}{2}$	*27*	*27*	*26*	*26*	*24*	*24*	*23*	*21*	*20*	*19*	*18*	*17*	*17*	*17*	*17*	*17*	*17*
$\frac{1}{3}$	*26*	*25*	*24*	*24*	*23*	*22*	*21*	*20*	*19*	*17*	*17*	*16*	*16*	*16*	*16*	*15*	*15*
$\frac{1}{5}$	*24*	*24*	*23*	*22*	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*15*	*14*	*14*	*14*
$\frac{1}{10}$	*23*	*22*	*22*	*21*	*21*	*20*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*14*	*14*	*13*	*13*
$\frac{1}{20}$	*22*	*22*	*22*	*21*	*21*	*20*	*20*	*19*	*18*	*17*	*16*	*15*	*15*	*14*	*14*	*13*	*13*
$\frac{1}{50}$	*22*	*22*	*22*	*21*	*21*	*20*	*20*	*19*	*19*	*18*	*18*	*16*	*16*	*15*	*14*	*13*	*13*
$\frac{1}{100}$	*21*	*21*	*21*	*21*	*21*	*21*	*20*	*20*	*19*	*19*	*18*	*17*	*16*	*15*	*15*	*13*	*13*

4.3. ProbabilityFrontTreeSplayListNet

Если мы знаем, что нам не нужно проводить обновление, то вместо того, чтобы идти к наибольшему общему предку, можно пойти вперёд. Алгоритм `send` в `ProbabilityFrontTreeSplayListNet` выглядит следующим образом:

1) Если выпала вероятность p, то делается `send` также как и в `TreeSplayListNet`. Назовём это `send` через предка.
2) Если выпала вероятность $1 - p$, то делается `send` также как и в `SkipListNet`. Назовём это `send` напрямую. Причём, пусть h' – максимальный уровень на который мы поднимаемся.

Лемма 8. Во время `send` напрямую на каждом новом уровне посещается не более трёх вершин, неудовлетворяющих условию спуска.

Доказательство. Предположим что при `send` напрямую, на каком-то уровне l существуют хотя бы 5 вершин u_1, u_2, u_3, u_4, которые не удовлетворяют условию спуска. В таком случае $\forall\, i < 4\; hits(u_i, l) + hits(u_{i+1}, l) > \frac{m}{2^{MaxLevel-l}}$ $\Rightarrow \Sigma_{i=1}^{5} hits(u_i, l) = (hits(u_1, l) + hits(u_2, l)) + (hits(u_3, l) + hits(u_4, l)) > \frac{m}{2^{MaxLevel-l}} + \frac{m}{2^{MaxLevel-l}} > \frac{m}{2^{MaxLevel-l-1}} \Rightarrow u_1$ удовлетворяет условию подъёма, однако при `send` напрямую, мы не изменяли никаких значений, а значит u_1 стало удовлетворять условию подъёма, ещё во время прошлых `send` через предка, что противоречит лемме 6

Теорема 9. Предположим во время `send` напрямую было пройдено d объектов, удовлетворяющих условию спуска. Тогда количество шагов будет состовлять не более, чем $d + 3 \cdot y_1 + 3 \cdot y_2$, где $y_1 = \log \frac{m}{s_i.selfHits}$, а $y_2 = \log \frac{m}{f_i.selfHits}$.

Доказательство.

1) Так как во время `send` напрямую, мы посещаем каждую вершину лишь один раз, то и каждую вершину, что удовлетворяет спуску мы посетим лишь единожды.
2) Согласно лемме 2 [18] расстояние, которое мы пройдём от `head` находящейся на $MaxLevel$ до $s_i = \log \frac{m}{s_i.selfHits}$. Так как $h' \leqslant MaxLevel$, то и число уровней, которое мы пройдём от s_i до $h' \leqslant \log \frac{m}{s_i.selfHits}$. Аналогично для f_i.
3) Так как согласно лемме 8 на каждом новом уровне посещается не более чем 3 вершины не удовлетворяющих условию подъёма $\Rightarrow$ число пройденных вершин, не удовлетворяющих условию подъёма равно $3 \cdot y_1 + 3 \cdot y_2$.

Резюмируя всё выше сказанное, можно сказать, что общее число шагов не превышает $d + 3 \cdot y_1 + 3 \cdot y_2$.

Теорема 10. Пусть вероятность обновления в `ProbabilityFrontTreeSplayListNet` равна p. Тогда амортизированное время работы запроса $send(s_i, f_i)$ равно $\mathcal{O}(\frac{1}{p} \cdot (\log \frac{m}{s_i.selfHits} + \log \frac{m}{f_i.selfHits}))$

Доказательство. Основная проблема заключается в том, что при `send` напрямую те вершины, которые удовлетворяют условию спуска не будут спускаться. Попробуем оценить матожидание времени работы, а из него уже сделаем выводы.

1) Рассмотрим вершину x, которая была посещена во время операции $send$ и удовлетворяет условию спуска. Вероятность спуститься у такой вершины равна p, значит матожидание числа запросов, прежде чем, эта вершина спустится равно $\frac{1}{p}$. Значит, на каждые $\frac{1}{p}$ проходов по вершине, удовлетворяющей условию спуска, приходится один спуск.
2) Заметим, что как было сказано в [18] число подьёмов суммарно меньше, чем $\Sigma_{i=0}^{m} y_i$, а количество спусков не превышает количество подъёмов, так как для того, чтобы спуститься на 1 уровень, вершине нужно прежде на него подняться.

Рассмотрим общее число шагов, сделанных во время всех запросов. Предположим, что запросов через предка было сделано r, тогда матожидание запросов напрямую равно $\frac{1-p}{p} \cdot r$, таким образом общее число шагов $Steps$ равно $\Sigma_{i=0}^{r}(2 \cdot d_i + 4 \cdot y_{i1} + 8 \cdot y_{i2}) + \Sigma_{i=0}^{\frac{1-p}{p} \cdot r}(d_i + 3 \cdot y_{i1} + 3 \cdot y_{i2})$, как было выяснено выше $\Sigma_{i=0}^{\frac{1-p}{p} \cdot r} d_i = \frac{1}{p} \cdot \Sigma_{i=0}^{r} d_i$, где $\Sigma_{i=0}^{r} d_i$ - суммарное число спусков вершин.

Тогда получается, что $Steps \leqslant \Sigma_{i=0}^{r}(4 \cdot y_{i1} + 8 \cdot y_{i2}) + \Sigma_{i=0}^{\frac{1-p}{p} \cdot r}(3 \cdot y_{i1} + 3 \cdot y_{i2}) + \frac{1+p}{p} \cdot \Sigma_{i=0}^{r} d_i$ $\leqslant 8 \cdot \Sigma_{i=0}^{\frac{1}{p} \cdot r}(y_{i1} + y_{i2}) + 2 \cdot \frac{1}{p} \cdot \Sigma_{i=0}^{r}(y_{i1} + y_{i2}) = \mathcal{O}(\frac{1}{p} \cdot r \cdot (\log \frac{2 \cdot r}{s_i.selfHits} + \log \frac{2 \cdot r}{f_i.selfHits}))$ $\Rightarrow$ в худшем случае амортизированное время запроса равно $\mathcal{O}(\frac{1}{p} \cdot (\log \frac{m}{s_i.selfHits} + \log \frac{m}{f_i.selfHits}))$.

Как показывают замеры (Рисунок 30 и таблица 6) `ProbabilityFrontTreeSplayListNet` показывает себя даже лучше, чем обычные адаптивные алгоритмы, причём, чем меньше вероятность обновления, тем лучше отрабатывает сам алгоритм.

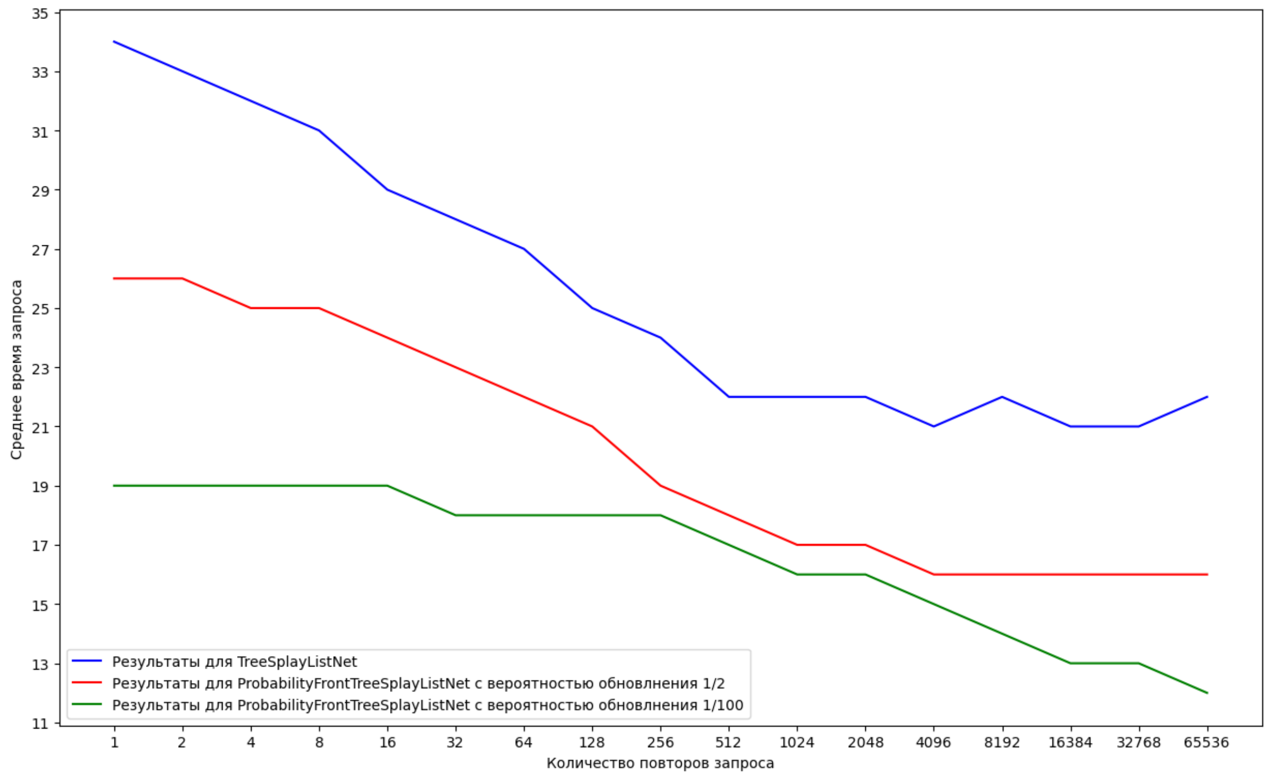


Рисунок 30 – Сравнение ProbabilityFrontTreeSplayListNet с TreeSplayListNet

Таблица 5 – Таблица числа шагов `ProbabilityFrontTreeSplayListNet`. В верхней строке указан log от числа шагов. Вероятность обновления

–	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
$\frac{1}{2}$	*26*	*26*	*25*	*25*	*24*	*23*	*22*	*21*	*19*	*18*	*17*	*17*	*16*	*16*	*16*	*16*	*16*
$\frac{1}{3}$	*24*	*24*	*23*	*23*	*22*	*21*	*21*	*19*	*18*	*17*	*16*	*15*	*15*	*15*	*15*	*14*	*14*
$\frac{1}{5}$	*22*	*22*	*21*	*21*	*20*	*19*	*19*	*18*	*17*	*16*	*15*	*14*	*14*	*13*	*14*	*13*	*13*
$\frac{1}{10}$	*20*	*20*	*20*	*20*	*19*	*19*	*18*	*18*	*17*	*15*	*15*	*14*	*13*	*13*	*13*	*12*	*13*
$\frac{1}{20}$	*20*	*20*	*19*	*19*	*19*	*18*	*18*	*18*	*17*	*16*	*15*	*14*	*14*	*13*	*13*	*12*	*12*
$\frac{1}{50}$	*19*	*19*	*19*	*19*	*19*	*18*	*18*	*18*	*17*	*17*	*16*	*15*	*14*	*14*	*13*	*12*	*12*
$\frac{1}{100}$	*19*	*19*	*19*	*19*	*19*	*18*	*18*	*18*	*18*	*17*	*16*	*16*	*15*	*14*	*13*	*13*	*12*

4.4. Выводы к главе 4

В данной главе были рассмотрены вероятностные версии алгоритмов, которые проводят обновления с вероятностью p такие как `ProbabilitySimpleSplayListNet` и `ProbabilityTreeSplayListNet`, а также их модификации, которые, если не проводится обновление посылают запрос напрямую - `ProbabilityFrontTreeSplayListNet`.

Как было доказано, амортизированная скорость операции $send$ таких алгоритмов равна $\mathcal{O}(\frac{1}{p} \cdot (\log \frac{m}{s_i.selfHits} + \log \frac{m}{f_i.selfHits}))$, однако как показывают графики такие алгоритмы только выигрывают в скорости, причём чем ниже вероятность обновления (в допустимых значениях, иначе получим `SkipListNet`), тем лучше работает структура.

Как можно видеть `ProbabilitySimpleSplayListNet` с $p = \frac{1}{100}$ даёт выигрыш до 73% чем её невероятностный аналог. `ProbabilityTreeSplayListNet` с $p = \frac{1}{100}$ даёт выигрыш в 69% над `TreeSplayListNet`, а `ProbabilityFrontTreeSplayListNet` до 83%.

Как следствие вероятностные аналоги хоть в худшем случае и проигрывают невероятностным, на практике получают гораздо больший выигрыш, причём уменьшение вероятности обновления уменьшает время работы (однако не стоит перебарщивать, иначе можно получить `SkipListNet`).

Стоит заметить, что единственный момент, когда мы как-то меняем структуру, является функция `update`. Также стоит заметить, что на каждой итерации цикла в `update` мы взаимодействуем только с четырьмя вершинами $current$, $previous$, $current.next[current.topLevel]$ и $previous.next[current.topLevel + 1]$ (и только в том случае, если $previous.topLevel > current.topLevel$ и только на 2-ух уровнях на $current.topLevel$ и $level = \min current.topLevel + 1, previous.topLevel$. Подробнее на рисунке 31

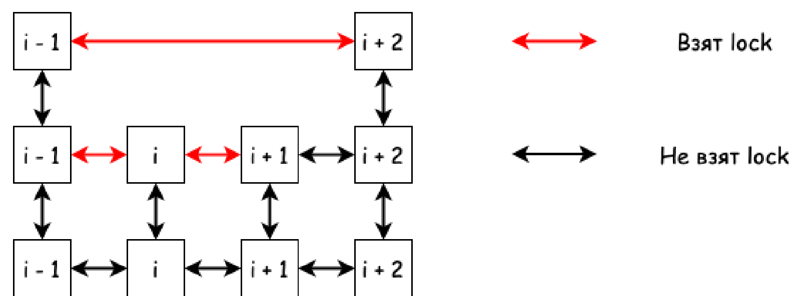


Рисунок 31 – Нужные нам lock

Из выше сказаного можно резюмировать, что для обновления достаточно на каждой итерации цикла брать `lock` на $i-1, i, i+1, i+2$, однако, это может оказаться достаточно дорого для нас. Второй вариант брать `lock` на нужные вершины на нужных уровнях. Введём для этого структуру `LockSplayNode`, содержащую, помимо основных полей:

1) `previousLocks` — массив локов на соединение с предыдущими вершинами на каждом уровне;
2) `nextLocks` — массив локов на соединение с следующими вершинами на каждом уровне.

1) $i-1.nextLocks[level]$ и $i+2.previousLocks[level]$. Так как в случае подъёма `current` нужно будет подержать место, чтобы не было конфликтов. Не нужно, если `previous` и `current` находятся на одном уровне.
2) $i.nextLocks[i.topLevel]$ и $i+1.previousLocks[i+1.topLevel]$. Так как во время спуска нужно будет удалить это ребро.
3) $i-1.nextLocks[i.topLevel]$ и $i.previousLocks[i.topLevel]$. Так как не можем допустить, чтобы какая-то вершина, поднялась и встала между $i-1$ и i.

Замечание 4. Когда мы будем выполнять `find`, то достаточно брать `lock` лишь на $nextLocks$ или $previousLocks$ (в зависимости от того в каку сторону идёт поиск).

Замечание 5. В `TreeSplayListNet` соединение между наибольшим общим предком и следующей за ней, лучше держать до тех пор пока алгоритм не дойдёт до него два раза (после f_i).

Замечание 6. Во избежание deadLock лучше брать `lock` именно в таком порядке.

5.1. Выводы к главе 5

В данной главе был рассмотрен способ перевести данные алгоритмы из последовательных в конкурентные. Как было выяснено, единственные момент, который может устроить конфлик - функция `update` (поэтому вероятностные алгоритмы выглядят наиболее выигрышными в конкурентном варианте).

Для того, чтобы `update` не вызывала конфликтов, стоит брать блокировки на 2 уровня: на $current.topLevel$ и $current.topLevel + 1$ (если $previous.topLevel > current.topLevel$), а после этого уже проводить все обновления.

ГЛАВА 6. ТЕСТИРОВАНИЕ НА РЕАЛЬНЫХ ДАННЫХ

Было решено провести тестирование на реальных данных, чтобы увидеть как быстро будут работать разработаные структуры. У нас есть три типа экспериментов: на равномерной рабочей нагрузке с 100 узлами, на синтетической рабочей нагрузке с 1023 узлами и разными значениями для параметра сложности во времени, то есть вероятности повторения последнего запроса [13] (0.25, 0.5, 0.75, 0.9), и на данных из трех реальных наборов данных: рабочая нагрузка высокопроизводительных вычислений `HPC` [2], рабочая нагрузка на `ProjectToR` [15] и синтетическая рабочая нагрузка `pFabric` [3].

Мы ограничиваем все наборы данных до 10^6 запросов для равномерной рабочей нагрузки с 100 узлами, HPC с 500 узлами, ProjectToR с 100 узлами и Facebook с 10000 узлами.

По причине того, что данных получается слишком много было решено ограничиться только результатами для `HPC`, `ProjectToR`, `pFabric`, а также равномерной и синтетической (с вероятностью повторения 0.5) рабочими нагрузками.

Алгоритмами в данной таблицы помечены:

ё

1) `Алгоритм 1` – `SkipListNet`
2) `Алгоритм 2` – `SimpleSplayListNet`
3) `Алгоритм 3` – `TreeSplayListNet`
4) `Алгоритм 4` – `SkipParentChildNet`
5) `Алгоритм 5` – `TreeSplayParentChildNet`
6) `Алгоритм 6` – `ProbabilitySimpleSplayListNet`
7) `Алгоритм 7` – `ProbabilityTreeSplayListNet`
8) `Алгоритм 8` – `ProbabilityFrontTreeSplayListNet`

Таблица 6 – Сравнение алгортмов на реальных данных.

–	**Facebook**	**HPC**	**ProjectToR**	**Равномерная**	**Синтетическая**
Алгоритм 1	*21*	*11*	*6*	*6*	*14*
Алгоритм 2	*18*	*19*	*5*	*13*	*19*
Алгоритм 3	*17*	*16*	*4*	*11*	*18*
Алгоритм 4	*15*	*12*	*4*	*9*	*14*
Алгоритм 5	*15*	*12*	*4*	*9*	*14*
Алгоритм 6	*13*	*10*	*3*	*7*	*12*
Алгоритм 7	*14*	*8*	*2*	*6*	*11*
Алгоритм 8	*12*	*7*	*2*	*5*	*10*

ГЛАВА 7. ИТОГИ

После реализаций всех алгоритмов, проведения доказательств сложности их работы и проверки замеров можно сделать следующие выводы.

1) Алгоритмы на основе `SplayList` действительно начинают работать быстрее по мере того как часто повторяется запрос, однако, каждый по своему.
2) `SimpleSplayList` работает гораздо хуже при редком повторении запроса и даёт лишь небольшой прирост на часто повторяющихся запросах, что в целом ожидаемо от простой адаптивной структуры.
3) `TreeSplayListNet` даёт больший прирост на часто повторяющихся запросах. Что же касается редко повторяющихся запросов, то здесь проигрыш ещё больший, чем у `SimpleSplayListNet`. Это обусловленно тем, что счётчики всех вершин уже увеличены для сохранения инварианта, поэтому редкие запросы не дают ожидаемого эффекта и не могут поднять вершину на нужную высоту. Если отказаться от инварианта, то скорость на редких запросах возрастёт и станет сравнимой с `SkipListNet`, однако при часто повторяемых запросах преимущество над обычным `TreeSplayListNet` нивилируется.
4) ParentChildNet даёт существенный прирост к скорости в независимости от того, какая структура используется в листьях. Чем меньше порог при котором образуется лист, тем быстрее работает структура, однако стоит выдерживать баланс, ведь чем меньше порог, тем больше соединений придётся выдерживать.
5) Амортизированное время запроса вероятностных структур составляет $\mathcal{O}(\frac{1}{p} \cdot (\log \frac{m}{s_i.selfHits} + \log \frac{m}{f_i.selfHits}))$, однако, как показывает практика, вероятностные алгоритмы работают лучше чем их невероятностные версии (а `ProbabilityFrontTreeSplayListNet` показыват результаты даже лучше, чем `ParentChildNet`), причём скорость работы тем больше, чем меньше вероятность обновления. Однако не стоит устремлять вероятность обновления к 0, ведь тогда мы получим `SkipListNet`.
6) Для реализации конкурентных версий алгоритма, нужно брать блокировки на не более чем четыра вершины за итерацию цикла и всего на двух уровнях: текущем и тот что выше. Когда мы не делаем обновления, до-

статочно брать блокировки только на ребро по которому мы перемещаемся.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Список с пропусками [Электронный ресурс]. — 2022. — URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_%D1%81_%D0%BF%D1%80%D0%BE%D0%BF%D1%83%D1%81%D0%BA%D0%B0%D0%BC%D0%B8.

2 *DOE U.* Characterization of the DOE Mini-apps. — 2016. — https://portal.nersc.gov/project/CAL/doe-miniapps.htm.

3 pfabric: Minimal near-optimal datacenter transport / M. Alizadeh [и др.] // ACM SIGCOMM Computer Communication Review. — 2013. — Т. 43, № 4. — С. 435–446.

4 Beyond fat-trees without antennae, mirrors, and disco-balls / S. Kassing [et al.] // SIGCOMM ’17: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. — 2017. — P. 281–294.

5 *C. Avin S. S.* Toward demand-aware networking: A theory for selfadjusting networks // ACM SIGCOMM Computer Communication Review. — 2018.

6 Distributed Self-Adjusting Tree Networks / B. S. Peres [et al.] // IEEE Transactions on Cloud Computing. — 2023. — P. 716–729.

7 Firefly: A reconfigurable wireless data center fabric using free-space optics / N. Hamedazimi [et al.] // ACM SIGCOMM Computer Communication Review. — 2014. — P. 319–330.

8 Helios: a hybrid electrical/optical switch architecture for modular data centers / N. Farrington [et al.] // ACM SIGCOMM Computer Communication Review. — 2011. — P. 339–350.

9 Inside the Social Network’s (Datacenter) Network / A. Roy [et al.] // SIGCOMM ’15: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication. — 2015. — P. 123–137.

10 *Kandula S.*, *Padhye J.*, *Bahl P.* Flyways to de-congest data center networks // ACM HotNets. — 2009.

11 *Noormohammadpour M.*, *Raghavendra C. S.* Datacenter Traffic Control: Understanding Techniques and Trade-offs // IEEE Communications Surveys and Tutorials. — 2017.

12 *O. Souza O. A. de*, *Goussevskaia O.*, *Schmid S.* CBNet: Demand-Aware Tree Topologies for Reconfigurable Datacenter Networks // Computer Networks: The International Journal of Computer and Telecommunications Networking. — 2022.

13 On the Complexity of Traffic Traces and Implications / C. Avin [et al.] // SIGMETRICS ’20: Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems. — 2020.

14 Proceedings of the 26th International Conference on Distributed Computing / Y. Afek [et al.] // Proceedings of the 26th International Conference on Distributed Computing. — 2012. — P. 1–15.

15 ProjecToR: Agile Reconfigurable Data Center Interconnect / M. Ghobadi [et al.] // SIGCOMM ’16: Proceedings of the 2016 ACM SIGCOMM Conference. — 2016. — P. 216–229.

16 Splaynet: Towards locally self-adjusting networks / S. Schmid [et al.] // IEEE/ACM Transactions on Networking (ToN). — 2016. — 24 (3). — P. 1421–1433.

17 The Nature of Datacenter Traffic: Measurements & Analysis / S. Kandula [et al.] // IMC ’09: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement. — 2009. — P. 202–208.

18 The Splay-List: A Distribution-Adaptive Concurrent Skip-List / V. Aksenov [et al.]. — 2020.