МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»
Кафедра математической кибернетики и компьютерных наук

РЕАЛИЗАЦИЯ ПОЛНОЙ ПЕРСИСТЕНТНОСТИ НА ОСНОВЕ COMPRESSED SUPER-NODE TREE

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 451 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Шнирельмана Алексея Павловича

Научный руководитель
зав. каф., к. ф.-м. н., доцент _____ С. В. Миронов

Заведующий кафедрой
к. ф.-м. н., доцент _____ С. В. Миронов

Саратов 2025

ВВЕДЕНИЕ

Актуальность темы. Персистентные структуры данных — это структуры данных, которые при изменениях сохраняют доступ ко всем своим предыдущим версиям, в отличие от обычных, эфемерных структур данных, любые изменения в которых приводят к потере предыдущей версии. Персистентные структуры данных находят применение в текстовых редакторах, системах контроля версий, базах данных, в некоторых параллельных алгоритмах, а также в функциональном программировании.

Помимо этого, персистентные структуры нередко используются в других алгоритмах, которые на первый взгляд не имеют к персистентности никакого отношения. В некоторых работах персистентные структуры данных используются для планирования движения, а также некоторых геометрических задачах, как например при локализации точки на плоскости.

Однако, многие способы реализации персистентности сложно применимы в общем случае и требуют значительного изменения внутреннего устройства структуры данных, как например в методе копирования пути. Этого недостатка лишены способы реализации персистентности, основанные на кэшировании отдельных версий эфемерной структуры данных, как это происходит, например, в методе полного копирования. Такие методы как правило не требуют доступа к внутреннему устройству исходной структуры данных, но являются гораздо менее эффективными.

Цель бакалаврской работы — разработка более эффективного метода реализации персистентных структур данных на основе кэширования версий. В связи с поставленной целью в данной работе решаются следующие **задачи**:

- исследование существующих подходов к реализации персистентных структур данных;
- разработка нового метода преобразования структуры данных в персистентную на основе кэширования версий;
- сравнение полученного алгоритма с существующими подходами.

Структура и объём работы. Бакалаврская работа состоит из введения, 4 разделов, заключения, списка использованных источников и 3 приложений. Общий объем работы — 57 страниц, из них 43 страниц — основное содержание, включая 6 рисунков и 1 таблицы, список использованных источников информации — 20 наименований.

1 КРАТКОЕ СОДЕРЖАНИЕ РАБОТЫ

1.1 Персистентность

В начале данного раздела описываются уровни персистентности:

— частичная;
— полная;
— конфлюэнтная;
— функциональная.

В данной работе рассматривались только частичная и полная персистентность.

Затем описываются классические подходы реализации персистентных структур данных.

1. *Полное копирование*: при каждом запросе изменения создается новая копия структуры данных.
2. *Метод хранения только начальной версии*: хранится только корневая версия структуры данных, а для ответа на get-запрос эта сохраненная версия приводится в искомое состояние последовательным применением $update$-запросов, затем происходит непосредственно ответ на запрос, после чего необходимо произвести откат в исходное состояние.
3. *Метод толстых узлов*: для каждого узла структуры данных информация о версиях хранится отдельно. Подходит только для реализации частичной персистентности.
4. *Метод копирования пути*: если структура данных представляет собой набор узлов со ссылками на другие, которые образуют ациклический граф, то можно при изменении одного узла создавать новую версию этого узла, а также всех узлов, из которых он доступен.

1.2 Дерево версий

В данном разделе описывается дерево версий персистентной структуры данных и подход к реализации персистентности на основе кэширования отдельных версий.

Сначала предлагается в качестве кэшируемых версий выбирать несколько случайных, а для ответа на get-запрос выбирать из них ближайшую к версии из запроса. Поиск такой версии не является тривиальным поэтому далее эта задача рассматривается более подробно.

Сначала рассматривается более простая версия этой задачи: поиск ближайшей вершины в дереве из набора выделенных с запросами выделения и снятия выделения с вершин. Эта задача широко известна и может быть решена с помощью центроидной декомпозиции. В данной работе было необходимо также обрабатывать запросы добавления листа в дерево. Для этого будем поддерживать множество центроидных декомпозиций, у каждой из которых кроме основной корень присоединяется к вершине другой центроидной декомпозиции. Таким образом, мы получаем дерево центроидных декомпозиций. При добавлении листа в исходное дерево мы создаем новую центроидную декомпозицию, состоящую только из этого листа, и подсоединяем ее к центроидной декомпозиции, в которой находится родитель добавленного листа в исходном дереве. Затем мы идем вверх по дереву центроидных декомпозиций и перестраиваем все текущее поддерево в одну центроидную декомпозицию, если его размер стал равен удвоенному размеру центроидной декомпозиции в корне этого поддерева. В работе доказываются следующие свойства полученной структуры данных:

— размер родителя в дереве центроидных декомпозиций больше размера сына;
— высота H дерева центроидных декомпозиций, построенного для дерева из N элементов, не превосходит $1 + \log_2 N$;
— время построения дерева центроидных декомпозиций, начиная с одной вершины запросами добавления листа в исходное дерево до размера N, равно $O(N \log^2 N)$.

Затем было предложено для случая частичной персистентности для кэширования выбирать вершины, которые соответствуют началам блоков корневой декомпозиции. После чего предлагается переделать это решение на случай полной персистентности с помощью обобщения корневой декомпозиции на случай дерева — Compressed super-node tree. В данной работе эта структура данных в несколько измененном виде используется для совершенно других целей в сравнении с оригинальной статьей: для выбора версий для кэширования. В работе отмечается также, что compressed super-node tree можно поддерживать при запросах добавления листа в исходное дерево. Для этого его нужно перестраивать заново через H запросов. При этом вершины, которые уже принадлежали Compressed super-node tree будут принадлежать Compressed super-node tree и после перестройки, таким образом сохраненные копии будут оставаться неизмен-

ными. Нам же нужно будет лишь добавлять новые версии. В этом разделе также приводятся детали реализации полной персистентности на основе Compressed super-node tree.

1.3 Использование персистентных структур данных в offline- и online-алгоритмах

В некоторых задачах на обработку запросов решение может быть значительно упрощено, если отвечать на запросы можно на все сразу после ввода всех запросов, а не на каждый запрос в отдельности после его ввода. В некоторых случаях возможно преобразовать такие решения для online-версий задач с использованием персистентных структур данных. В данном разделе сначала приводятся примеры таких задач.

Сначала рассматривается задача о поиске количества различных элементов на отрезке массива. В offline-решении можно отвечать на запросы в произвольном порядке. Давайте отсортируем запросы по левой границе. Оценим вклад, который дает каждое число массива в ответ: скажем, что если число встречается в полуинтервале запроса в первый раз оно дает 1, в противном случае 0. Составим на основе этого массив из нулей и единиц. Если значение встречается в первый раз в полуинтервале $[l, r_1)$ на позиции i, то и в полуинтервале $[l, r_2)$ это значение в первый раз может встретиться только на позиции i (оно может и не встретиться вовсе, если $i \geq r_2$). На основе этого составляется массив из нулей и единиц. Теперь для данного l ответы на запросы с левой границей l равны просто сумме в этом массиве нулей и единиц на полуинтервале $[l, r)$.

Рассмотрим, как изменится этот массив при переходе от l к $l+1$. Понятно, что изменения могут коснуться только позиций, на которых находится то же значение, что и на позиции l. Пусть nxt_l — позиция ближайшего справа от l элемента, равного a_l. Тогда изменение массива из нулей и единиц будет заключаться только в том, что в позиции nxt_l значение должно стать единицей (так как до этого оно было равно нулю, то это можно рассматривать как прибавление в элементе).

Таким образом, для поддержки описанного массива из нулей и единиц и обработки запросов к нему нам нужно такая структура данных, которая позволит осуществлять прибавление в элементе и поиск суммы на отрезке. Нам подойдет, например, дерево Фенвика или дерево отрезков. Такое решение ис-

пользует $O(N + Q)$ памяти (слагаемое Q появилось из-за того, что нам надо сохранить все запросы, прежде чем на них отвечать) и $O((N + Q) \log N)$ времени.

В offline-решении для каждого l неявно строилась своя версия массива из нулей и единиц. Давайте на этапе предпосчета построим каким-либо образом все эти версии. Теперь мы можем отвечать на запросы по мере их поступления.

Рассмотрим подробнее, как именно построить эти версии. По сути, у нас есть некоторая структура данных и к ней применяют запросы изменения(в нашем случае это прибавление в элементе). Нам нужно сохранить некоторые версии этой структуры данных. Понятно, что необходимо реализовать частичную персистентность. В данном случае подойдет персистентное дерево отрезков. Такое решение использует $O(N \log N)$ памяти и $O((N + Q) \log N)$ времени.

Далее описывается, как, используя описанный метод реализации персистентности, добавить поддержку запросов изменения. Будем поддерживать тот же массив из нулей и единиц, но будем использовать вместо дерева отрезков простейшую корневую декомпозицию: разделим массив из нулей и единиц на блоки примерно равного размера и для каждого блока будем хранить сумму элементов в этом блоке. При изменении элемента в массиве из нулей и единиц меняется только одна такая сумма, а при изменении элемента исходного массива меняются $O(1)$ элементов массива из нулей и единиц для фиксированного l. Таким образом при фиксированном l запрос изменения можно обработать за $O(\log N)$(логарифм появился из-за необходимости пересчитывать описанный выше массив nxt, что может быть тривиально реализовано с помощью двоичных деревьев поиска, но важно понимать, что массив nxt один и тот же для разных l, поэтому при обработке изменений для K значений l будет потрачено только $O(\log N + K)$ времени). При изменении l на единицу также меняется $O(1)$ элементов массива из нулей и единиц, но в этом случае массив nxt не меняется, поэтому такое изменение будет обработано за $O(1)$. Ответ на запрос представляет собой поиск суммы на отрезке, что может быть реализовано с помощью описанной корневой декомпозиции за $O(\sqrt{N})$.

Теперь подобно Online-решению сделаем эту структуру данных персистентной. Для этого воспользуемся другой корневой декомпозицией: разделим все значения l от 0 до $N - 1$ на блоки и в каждом блоке построим описанную структуру данных, состоящую из массива из нулей и единиц и корневой

декомпозиции на сумму. Теперь описанные задача без изменений решается за $O(N \log N + Q_g\sqrt{N})$ времени и $O(N\sqrt{N})$ памяти. Однако затраты памяти можно уменьшить до $O(N)$, если хранить только сумму в блоках, но не сами элементы массивов из нулей и единиц, так как эти элементы могут быть легко получены из массива nxt.

Чтобы обрабатывать $update$-запросы, нужно производить описанные выше действия над каждой сохраненной версией корневой декомпозиции. Тогда изменение элемента массива потребует $O(\sqrt{N})$ времени (так как версий $O(\sqrt{N})$ и на каждую версию нужно потратить $O(1)$ времени). Таким образом, итоговое время работы всего решения будет $O(N \log N + Q\sqrt{N})$.

В данном разделе приводятся и другие примеры подобных задач.

1.4 Вычислительные эксперименты

В последнем разделе приводятся несколько вычислительных экспериментов для сравнения описанных вариантов реализации персистентности на основе кэшируемых версий.

Тестировались эти реализации на примере персистентного массива. Формально, пусть дан числовой массив a из N элементов и Q запросов к нему. Запросы бывают двух типов:

— $\mathrm{update\ v\ i\ x}$ — создать новую версию массива на основе версии v, сделав присвоение $a_i \leftarrow x$.

— $\mathrm{get\ v\ i}$ — вывести значение i-го элемента в версии v массива a.

В этой задаче запросы в неперсистентном случае обрабатываются за константное время: $T_g(N) = O(1), T_u(N) = O(1)$.

Сравнивались следующие решения:

1. Полное копирование: храним каждую версию.
2. Корневая версия: храним только начальную версию массива и все запросы изменения. По ним можно легко получить любую другую версию.
3. Случайные версии(1): после каждого H-го запроса изменения выбираем случайную несохраненную версию и сохраняем ее. Для ответа на get-запрос используем ближайшую сохраненную версию и переносим эту версию в версию запроса.
4. Случайные версии(1): после каждого H-го запроса изменения выбираем случайную несохраненную версию и сохраняем ее. Для ответа на get-запрос используем ближайшую сохраненную версию, сохраненная версия

Решение	Время для update-запроса	Время для get-запроса	Память
Полное копирование	$O(N)$	$O(1)$	$O(QN)$
Корневая версия	$O(1)$	$O(Q)$	$O(N)$
Случайные версии(1)	$O(\frac{Q+N}{H})$ амортизировано	$\Omega(H)$ в худшем случае	$O(\frac{QN}{H})$
Случайные версии(2)	$O(\frac{Q+N}{H})$ амортизировано	$\Omega(H)$ в худшем случае	$O(\frac{QN}{H})$
Compressed Super-Node Tree	$O(\frac{Q+N}{H})$ амортизировано	$O(H)$	$O(\frac{QN}{H})$

Таблица 1 – Теоретические предсказания используемых времени и памяти для разных решений

при этом остается та же.

5. На основе Compressed Super-Node Tree: храним вершины, которые соответствуют вершинам Compressed Super-Node Tree, построенном на дереве версий с параметром H.

На основе предыдущих разделов построена таблица 1 оценки используемых времени и памяти этих решений.

Последние решения зависят от параметра H, который в каждом случае означает разные вещи, однако верно, что количество поддерживаемых версий равно $O(\frac{Q_u}{H})$). Поэтому для чистоты эксперимента будем считать, что H во всех этих случаях будет иметь одинаковое значение. Так как в последнем случае целесообразно выбрать $H = \sqrt{Q}$, то будем далее считать, что H всегда $\sqrt{Q}$.

Для экспериментов генерировались тесты с $N = Q$. Значения элементов в исходном массиве, в $update$-запросах, индексы во всех запросах а также версии в get-запросах генерировались равновероятным выбором из диапазона допустимых значений.

Отдельный интерес вызывал выбор версий в $update$-запросах, так как от этого зависит структура дерева версий. Далее рассматривается несколько случаев. Для каждого случая было сгенерировано по 10 тестов для Q от 10^3 до 10^4 с шагом 10^3 и от 10^4 до 10^5 с шагом 10^4 (итого, 200 тестов для каждого случая).

1.4.1 Случайный равновероятный выбор родительских версий

В этом случае версия, от которой будет образована новая, выбиралась равновероятно среди уже существующих. Полное копирование показало результат, значительно худший как по времени, так и по памяти, что в целом соответствует теоретическим предположениям. Ожидаемыми оказались также затраты по времени решений на основе случайного выбора кэшируемых версий и Compressed

Super Node Tree, а также затраты памяти решения со случайными версиями. Хороший результат решения с только корневой версией, а также маленькие затраты памяти в решении на основе CSNT можно объяснить тем, что при таком методе генерации дерева высота полученного дерева значительно меньше его размера.

1.4.2 Максимум из случайных версий

В следующем следующем случае для генерации родительской вершин случайно генерируются $t + 1$ вершин и из них выбирается максимум — этот максимум и будет родительской вершиной. При $t = 0$ этот способ вырождается в предыдущий. При такой генерации новые версии чаще будут образовываться на основе более поздних версий, что кажется довольно естественным в некоторых задачах. Высота такого дерева также будет больше, поэтому можно ожидать, что решения на основе корневой версии и CSNT потеряют свое преимущество. Эксперименты проводились при $t = 1000$. Решение с корневой версией действительно показало значительно худшие результаты по используемому времени. Решение на основе CSNT стало тратить чуть большее количество памяти, но все еще значительно меньше, чем решения на основе случайных версий, которые имеют схожее время работы.

1.4.3 Бамбук

Далее рассматривался крайний случай, в котором каждая следующая версия образуется на основе предыдущей, сводя задачу к частичной персистентности. Дерево версий в этом случае вырождается в бамбук. Высота такого дерева получается порядка Q. В результате получилось усиление тенденций, замеченных еще в предыдущем случае: решение с только корневой версией потратило значительно больше времени, чем решения со случайными версиями и на основе CSNT. Используемая решением с CSNT память максимально приблизилась к таковой у решений на основе случайных версий. Незначительное преимущество предположительно обеспечивается дополнительными издержками решений со случайными версиями на поддержку дерева центроидных декомпозиций.

После этого последовала попытка создать максимально неблагоприятные условия: пусть дерево версий будет все еще представлять из себя бамбук, но теперь в качестве версий из get-запроса будем выбирать ту версию, которая максимально удалена от всех кэшированных версий. Для каждого решения вы-

бор конечно может отличаться, поэтому тесты генерировались независимо для каждого решения, отличаясь только в выборе версий в get-запросах и сохраняя все остальное.

Результаты оказались похожими на предыдущие. Серьезные изменения показало только 2-е решение, основанное на выборе случайных версий. Его время работы увеличилось более чем в 2 раза. Связано это, по-видимому, с тем, что при случайном выборе кэшируемых версий в бамбуке вероятно появление цепочки последовательных некэшированных версий, длина которой будет значительно превосходить идеальные H версий. В то время как в 1-м решении, основанном на выборе случайных версий, такие прогалы будут автоматически исчезать при движении кэшированных версий. Однако, такое поведение довольно предсказуемо и само по себе создает уязвимость. Поэтому последний тест составлялся специально против этого решения.

Будем генерировать запросы таким образом, чтобы все кэшированные версии находились максимально близко к корню бамбука. При $update$-запросе этот инвариант будет ломаться только, когда решение будет добавлять новую кэшированную версию. Однако мы можем это исправить за $O(\log V)$ get-запросов следующим образом: рассмотрим суффикс версий, среди которых нет кэшированных, сделаем запрос к версии, которая отстоит на единицу в сторону последней версии от середины этого суффикса. Для ответа на этот запрос будет гарантированно использована новая кэшированная версия. Теперь построенным по тому же принципу get-запросом приблизим новую кэшированную версию к корню. Легко видеть, что каждый раз расстояние между последней и предпоследней кэшированными версиями сокращается примерно в 2 раза и процесс остановится, когда между ними будет одна некэшированная версия. Таким образом, мы можем за лишние $O(\frac{Q}{H}\log Q)$ запросов гарантировать, что все кэшированные версии находятся на расстоянии $O(H)$ от корня бамбука. Потратим на построение дерева версий приммерно половину запросов, оставшиеся запросы будут get-запросами. Они будут идти парами: спросить про последнюю существующую версию, спросить про середину суффикса, в котором после построения бамбука не было кэшированных версий.

Из построения теста, на последние $O(\frac{Q}{2})$ запросов решение будет отвечать с использованием только одной кэшированной версии, а так как описанный суффикс без кэшированных версий при $H = O(\sqrt{Q})$ будет иметь размер $O(Q -$

$H) = O(Q)$, то каждый запрос из второй половины будет обрабатываться за $O(Q)$ времени, что приводит к квадратичной асимптотике $O(Q^2)$ по времени на весь тест. Стоит отметить, что так как речь идет только о половине запросов и последняя кэшированная версия будет каждый раз проходить только половину описанного суффикса, то константный фактор такого решения будет довольно маленьким.

Как и ожидалось 1-е решение, основанное на случайном выборе кэшированных версий, показало значительно худшие результаты, которые все еще несколько лучше, чем у решения, основанном на кэшировании только корневой версий, из-за константного фактора.

Стоит также отметить, что описанный способ генерации тестов не зависит от конкретного способа выбора случайных версий.

ЗАКЛЮЧЕНИЕ

В данной работе рассмотрены основные методы реализации персистентных структур данных, а также разработаны новые методы реализации частичной и полной персистентности на основе кэшируемых версий, которые позволяют преобразовывать практически любую структуру данных в персистентную без учета их внутреннего строения.

При сравнении этих методов, а также подобных им общеизвестных методов, на случайных деревьях версий лучше всего себя показал метод, заключающийся в хранении только начальной версии структуры данных, однако он оказался не таким хорошим в крайних случаях. В то время как, метод основанный на Compressed Super-Node Tree показал гораздо более стабильные результаты.

Также были рассмотрено применение персистентных структур данных для перехода от offline-алгоритмов обработки запросов к online-алгоритмам и был предложен способ перехода от таких online-алгоритмов к алгоритмам, поддерживающим запросы изменения, с помощью реализации персистентности на основе кэшируемых версий.