Персистентность

Фёдор Мамаев[1]

[1]Современные структуры данных, МКН СПбГУ, весна 2025

Статьи: (Driscoll и др., 1989) (ссылка), (Sarnak & Tarjan, 1986) (ссылка)

1. Постановка задачи

- Структуры данных, с которыми мы почти всегда имеем дело, являются *эфемерными*, то есть после изменения их состояния старое состояние разрушается. Но бывает так, что нам приходится смотреть состояния структуры данных, которые были когда-то раньше. Структуры данных, поддерживающие такую возможность, называются *персистентными* (от англ. persistent). Различают *частичную персистентность* (partial persistency), когда мы можем читать любые состояния структуры данных, но менять только последовательные, и *полную персистентность* (full persistency), когда мы можем вносить изменения в прошлые версии структуры данных, создавая дерево версий (нечто вроде «альтернативной истории»).

2. Наивные подходы

Если у нас есть эфемерная структура данных, то очевидный способ превратить её в персистентную — просто копировать всю структуру при каждом изменении и хранить в памяти все версии. Понятно, что это очень неэффективно.

Также можно хранить в памяти последнюю версию и последовательность операций, а при необходимости прочитать прошлую версию восстанавливать её с нуля, выполняя все операции заново. Этот способ лучше, но всё равно неэффективный.

Можно зафиксировать число k и хранить последовательность операций и каждую k-ю версию, а прошлые версии восстанавливать с последнего «чекпоинта». К сожалению, какое k мы ни возьмём, у нас либо время на чтение прошлой версии будет $\mathcal{O}(\sqrt{m})$, либо занимаемая память будет $\mathcal{O}(n\sqrt{m})$. Можно лучше.

3. Частичная персистентность

3.1. Предположения о структуре данных

Все подходы выше предполагали структуру данных неким «чёрным ящиком». Мы сделаем о ней несколько предположений. Пусть структура состоит из вершин. У каждой вершины есть поля двух типов: **информационные** (в которых хранятся данные простых типов) и **указатели** (на другие вершины структуры). Набор полей фиксирован для конкретной структуры. Кроме того, есть один или несколько **указателей доступа**, которые указывают на какие-то вершины структуры.

Пусть есть 2 типа операций со структурой: **операции чтения**, состоящие из **шагов чтения**, и **операции записи**, состоящие из шагов чтения и **шагов записи**. Один шаг чтения заключается в том, что мы переходим в некоторую вершину структуры, читаем её поля и переходим в следующую вершину по одному из указателей в вершинах, в которых

мы уже побывали. Шаг записи заключается в том, что мы меняем одно или несколько полей этой вершины. Все операции начинаются с вершин по указателям доступа.

3.2. Метод толстой вершины

Пусть у нас есть эфемерная структура данных с предположениями выше. Построим на её основе частично персистентную структуру, которая будет устроена так же, но в каждой вершине вместо каждого поля мы будем хранить список пар [значение, номер версии], добавляя в него новое значение каждый раз, когда в эфемерной структуре значение поля меняется. Чтобы выполнить операцию чтения над версией с номером i эфемерной структуры данных, нужно на шаге чтения искать в списке значение с наибольшим номером версии, не превосходящим i. На такой поиск тратится $\mathcal{O}(\log m)$ времени. Если считать, что каждая операция записи меняет $\mathcal{O}(1)$ полей, то занимаемая память, очевидно, $\mathcal{O}(n + m)$.

3.3. Метод копирования вершин

Идея: пусть у нас одна персистентная вершина может содержать только фиксированное число версий значений её полей, но будет несколько версий одной вершины. В нашей реализации одна персистентная вершина будет содержать только одну версию всех информационных полей, но несколько версий указателей (реализация легко обобщается и на использование нескольких версий информационных полей). Пусть одна персистентная вершина $\bar{x}$ содержит $d + p + e + 1$ полей, где d – число её полей-указателей, p – максимальное число предков вершины в эфемерной структуре (считаем константой) – мы будем хранить указатели на предков вершины, e – число дополнительных мест для указателей (константа, выберем позже), и 1 поле – указатель на следующую версию вершины $\bar{x}$. Также вершина $\bar{x}$ содержит номер своей версии i и номер версии для каждого дополнительного указателя ($\geqslant i$; считается, что значения в основных полях-указателях и информационных полях относятся к версии i). Операции чтения выполняются точно так же, как и в методе толстой вершины, но поиск нужного значения, очевидно, занимает константное время.

Во время операции записи мы храним множество скопированных вершин S. Пусть последняя версия структуры $i - 1$. Каждый раз, когда шаг записи меняет информационное поле в эфемерной вершине x, в персистентной структуре мы делаем копию последней версии вершины $\bar{x}$, назначая ей версию i (если такой копии ещё нет) и копируя туда значения всех полей (обновляя обратные указатели, если это нужно). После этого мы добавляем $\bar{x}$ в S. Если шаг записи меняет поле-указатель, то в персистентной вершине мы добавляем новое значение в дополнительные места для указателей (которых e), а если там места не осталось, то копируем вершину, как описано выше. После операции нужно выполнить постпроцессинг множества S: для каждой вершины $\bar{y}$ из S нужно поправить указатели в её предках, чтобы они указывали на последнюю версию, т.е. на копию. В каждый из предков мы добавляем новое значение указателя на $\bar{y}$ версии i, причём если места не осталось, то предка придётся скопировать и добавить в S.

3.4. Анализ метода копирования вершин

Выберем константу e так, чтобы $e \geqslant p$. Определим **потенциал** персистентной структуры данных как $\frac{e}{e-p+1}$, умноженное на число **живых вершин** в ней, т. е. вершин, у которых нет копий более поздней версии, минус $\frac{1}{e-p+1}$, умноженное на число неиспользуемых дополнительных указателей в них. Очевидно, что потенциал пустой структуры равен нулю, и что потенциал всегда больше нуля. Определим **амортизированную стоимость по памяти** операции записи как число вершин, создаваемых ей, плюс изменение потенциала

за эту операцию. Очевидно, что эта величина – верхняя оценка для числа вершин, созданных операцией.

Пусть у нас есть операция записи, делающая u шагов. Пусть $t \leqslant u$ – число вершин в S до постпроцессинга и k – число скопированных при постпроцессинге вершин. Когда вершина копируется при постпроцессинге, она перестаёт быть живой и создаётся новая вершина с потенциалом 0, т. о. потенциал уменьшается на $\frac{e}{e-p+1}$. Когда вершина удаляется из S, мы добавляем указатели на её копию. Всего мы можем добавить не более $p(t+k)$ таких указателей, но на самом деле не более $p(t+k)-k$, так как эти k указателей лежат в основных полях вершин-копий, добавленных в S при постпроцессинге. Итого, АСП постпроцессинга составляет не более $k + \frac{p(t+k)-k}{e-p+1} - \frac{ke}{e-p+1} = \ldots = \frac{pt}{e-p+1} = \mathcal{O}(t) = \mathcal{O}(u)$, значит, и АСП всей операции $\mathcal{O}(u)$, или амортизированное $\mathcal{O}(1)$ на шаг записи. Аналогичным рассуждением можно доказать и амортизированную оценку $\mathcal{O}(1)$ времени на шаг записи.

4. Полная персистентность

4.1. Метод толстой вершины

Первая проблема, с которой мы сталкиваемся при реализации полной персистентности, – версии теперь образуют собой не линейную последовательность, а дерево, где каждая версия может быть ответвлением любой из предыдущих. Удобно хранить версии в виде списка таким образом, чтобы для любой версии всё версии из её поддерева стояли сразу после неё в списке. Статья (Dietz & Sleator, 1987) описывает алгоритм, позволяющий вести такой список и выполнять все операции с ним за $\mathcal{O}(1)$.

В остальном метод толстой вершины не отличается от описанного ранее. Один момент: когда мы вставляем значение некоторого поля версии i в толстую вершину, если в этой толстой вершине оно попадает между значениями версий $j < i$ и $k > i+1$, мы должны добавить в вершину также значение версии i_+ (следующей за i в списке версий), такое же, как и значение версии j, чтобы на интервале версий от i_+ до k это значение осталось прежним.

Оценка времени прежняя: $\mathcal{O}(\log m)$ на шаг чтения/записи.

4.2. Метод разделения вершин

Метод разделения вершин основан на методе копирования вершин. Идея такая: при создании копий мы будем оставлять свободное место в обеих вершинах для будущих обновлений.

Для простоты давайте добавим обратные указатели явно в эфемерную структуру, получив **аугментированную эфемерную структуру**. Пусть k – число прямых и обратных указателей в вершине этой структуры.

Пусть в персистентной вершине будет $k + 2e + 1$ указателей: k – значения полей-указателей той же версии, что и вершина; $2e$, где $e \geqslant k$ – константа, – дополнительные значения для указателей; и 1 – указатель на следующую копию вершины.

Пусть **правильный интервал** вершины $\bar{x}$ – интервал версий от i (версии вершины) до версии следующей копии вершины $\bar{x}$ (не включительно) или до последней версии, если такой копии не существует. Пусть **правильный интервал** указателя – интервал версий от версии указателя до версии следующего значения того же указателя не включительно.

Назовём указатель версии i **правильным**, если он указывает на вершину, правильный интервал которой содержит i; назовём указатель на вершину $\bar{x}$ **пересекающимся**, если его правильный интервал не содержится полностью внутри правильного интервала $\bar{x}$. Нужно поддерживать инвариант: между операциями записи все указатели правильные и не пересекающиеся.

Операция записи выполняется так же, как и в методе копирования вершины, но: если она меняет информационное поле или если она меняет поле-указатель и для нового значения нет места, то копирование вершины происходит так. Пусть $\bar{x}_-$ – вершина, которую нужно скопировать, i_- – её версия; $\bar{x}_>$ – её следующая копия, её версия $i_>$; пусть i – версия нашего нового значения, а i_+ – следующая за ней версия в списке версий. Мы создаём две новые вершины: $\bar{x}$ версии i и $\bar{x}_+$ версии i_+. Указатель на следующую копию из $\bar{x}_-$ указывает на $\bar{x}$; из $\bar{x}$ – на $\bar{x}_+$; из $\bar{x}_+$ – на $\bar{x}_>$. После этого все значения из дополнительных полей-указателей с версиями i_+ и более в вершине $\bar{x}_-$ мы перемещаем в $\bar{x}_+$. В $\bar{x}$ мы записываем новое значение поля. Если $i_+ = i_>$, то вершин будет меньше ($\bar{x}_+$ будет совпадать с $\bar{x}_>$). Во время операции мы также меняем все неправильные указатели, делая их правильными, переназначая их на нужную копию. Также мы строим множество S из всех вершин, в которых есть пересекающиеся указатели.

После этого мы выполняем постпроцессинг. Он состоит их трёх шагов: 1) берём вершину $\bar{x}$ из S. Строим список L всех указателей из $\bar{x}$. Причём если указатель пересекающийся, то добавляем в список новый указатель на нужную копию версии i. Теперь в списке L все указатели непересекающиеся. 2) Если все указатели из L поместятся в вершине $\bar{x}$, помещаем их туда и пропускаем шаг 3; если нет, делим список на группы по e указателей, после этого для каждой группы (кроме первой) создаём новую вершину. В итоге в каждой вершине останется e свободных мест. 3) делаем все указатели правильными. После постпроцессинга все указатели будут правильными и непересекающимися.

4.3. Анализ метода разделения вершин

Определим потенциал персистентной структуры как $\frac{e}{e-k+1}$ на число вершин минус $\frac{1}{e-k+1}$, умноженное на число свободных мест для дополнительных указателей, считая (внимание!) не более e свободных мест в каждой вершине. Таким образом, созданные на шаге 2 постпроцессинга вершины имеют потенциал 0.

Пусть у нас есть операция записи из u шагов записи. Сама операция создаёт не более $2u$ новых вершин. Пусть l – число вершин, созданных на шаге 2 постпроцессинга. При создании новой вершины мы добавляем не более k новых указателей. Амортизированная стоимость операции по памяти, таким образом, не более $2u + l + \frac{(2u+l)k}{e-k+1} - \frac{(e+1)l}{e-k+1} = \ldots = 2u + \frac{2uk}{e-k+1} = \mathcal{O}(u)$.

5. Пример

В качестве примера применения только что описанных нами способов построения персистентной структуры данных разберём следующую задачу из статьи (Sarnak & Tarjan, 1986). Пусть у нас есть N точек на плоскости. Точки соединены непересекающимися отрезками или лучами, которые делят плоскость на части, имеющую форму многоугольников, возможно, бесконечных. Требуется в режиме онлайн для произвольной точки на плоскости определить многоугольник, в который она входит. Согласно формуле Эйлера, всего будет $\mathcal{O}(n)$ отрезков и многоугольников.

Мы решим эту задачу следующим образом. Создадим персистентное двоичное дерево поиска и будем идти сканлайном по всем точкам (для простоты будем считать, что никакие две точки не лежат на одной вертикальной прямой). В дереве поиска будут храниться отрезки, которые в данный момент пересекает прямая сканлайна, причём отрезок, находящийся левее какого-то другого отрезка (по направлению слева направо), будет лежать левее его в двоичном дереве. В каждой точке сходятся сколько-то отрезков и сколько-то выходят из неё. Причём все эти отрезки будут лежать в дереве последовательно. Соответственно, мы должны будем удалить и добавить эти отрезки в дерево соответственно, создавая новую версию. Это будет предподсчёт. Теперь про запросы.

Когда мы получаем точку, мы заглядываем в персистентное дерево, каким оно было, когда прямая сканлайна была на x-координате этой точки, и ищем, между какими отрезками эта точка лежала. Так мы можем определить многоугольник, в котором точка лежит. Время решения составляет $\mathcal{O}(n \log n)$ на предподсчёт и $\mathcal{O}(\log n)$ на запрос. $\mathcal{O}(n)$ памяти.

References

Dietz, P., & Sleator, D. D. (1987). Two algorithms for maintaining order in a list. *Proceedings, 19th Annual ACM Symp. on Theory of Computing, 1987, pp. 365-372.*

Driscoll, J. R., Sarnak, N., Sleator, D. D., & Tarjan, R. E. (1989). Making Data Structures Persistent. *Journal of Computer and System Sciences*, *38*(1).

Sarnak, N., & Tarjan, R. E. (1986). Planar Point Location Using Persistent Search Trees. *Communications of the ACM*, *29*(7).