Санкт-Петербургский национальный исследовательский университет
информационных технологий, механики и оптики

Кафедра компьютерных технологий

Минаев Б. Ю.

Реализация динамических Rake-Compress деревьев в случае отсутствия ограничения на степени вершин

Бакалаврская работа

Научный руководитель: Буздалов М. В.

Санкт-Петербург
2015

Содержание

Введение

Динамические деревья находят множество применений в информатике. В частности, они используются в алгоритмах поиска максимального потока, а также для решения задачи динамической связности ациклических графов. Например, они позволяют уменьшить время работы алгоритма проталкивания предпотока с $O(EV^2)$ до $O(EV \log(V^2/E))$ [ссылочка на статью].

Зачастую, термин "динамические деревья" ассоциируется только с link-cut деревьями, которые были предложены Слетером и Тарьяном [еще одна ссылка]. Однако, существуют и другие динамические деревья. Например, Top-деревья [ссылочка] и Rake-Compress деревья [еще одна]. В данной работе будут рассмотрены именно последние. Они обладают существенным преимуществом перед другими динамическими деревьями — их построение можно довольно легко сделать параллельным.

На сегодняшний день все больше и больше компьютеров имеют внутри себя несколько процессоров, способных работать параллельно. Поэтому актуальной задачей является разработка алгоритмов, способных работать на нескольких процессорах одновременно. Rake-Compress деревья потенциально являются именно такой структурой данных.

Реализация Rake-Compress деревьев, которая была предложена Умутом Акаром в оригинальной статье обладала несколькими недостатками. Во-первых, рассматривались только деревья, у которых степень каждой вершины ограничена некоторой константой. Во-вторых, статья носит скорее теоретический, чем практический характер. В ней уделяется внимание только асмптотическим оценкам. На практике же необходимо иметь структуры данных эффективные не только с точки зрения асимптотики,

но и с точки зрения констант, которые в них скрыты.

В данной работе показано как избавиться от ограничения на степени вершин, а также как эффективно реализовывать Rake-Compress деревья.

Глава 1. Постановка задачи

1.1. Необходимые элементы теории графов

Для начала введем несколько определений из теории графов, которые понадобятся в дальнейшем.

Графом называется упорядоченная пара (V, E), где V — множество вершин, а E — множество ребер.

Различают **ориентированные** и **неориентированные** графы. В первом случае каждое ребро это упорядоченная пара вершин, во втором — неупорядоченная.

Вершина v является **смежной** с u, если в графе существует ребро (u, v).

Говорят, что из u **достижима** v, если существует список вершин такой, что:

- первая вершина в списке — u
- последняя вершина в списке — v
- соседние в списке вершины являются смежными

Граф называется **связным**, если для любой пары вершин (u, v) u достижима из v.

Деревом называется связный неориентированный граф с n вершинами и $n - 1$ ребром.

Степенью вершины называется количество вершин, которые смежны с данной. В случае ориентированных графов различают **входящую** и **исходящую** степень.

Корневым деревом называется граф с выделенной вершиной — корнем. Исходящая степень корня равна нулю, а исходящая степень лю-

бой другой вершины — единице. Корень должен быть достижим из всех вершин.

Поддеревом вершины v называется множество всех вершин, из которых достижима v.

Листом называется вершина, входящая степень которой равна нулю.

Лесом называется множество деревьев, которые не пересекаются по вершинам.

В данной работе большинство внимания будет уделено эффективному хранению леса, каждое дерево которого является корневым. Однако, в главе [???] будет рассмотрен случай неориентированных деревьев.

1.2. Динамические деревья

Задача, которая решается с помощью динамических деревьев, формулируется следующим образом. Необходимо поддерживать лес деревьев и выполнять на нем следующие операции:

- Добавить ребро (u, v). Вершина u должна быть корнем некоторого дерева.
- Удалить ребро (u, v). Ребро (u, v) должно присутствовать в графе.
- Некоторый вопрос относительно структуры дерева.

Примером последней операции может быть запрос "связны ли вершины u и v?", "сколько ребер на кратчайшем пути из u в v?" или "какова сумма номеров вершин, которые находятся в поддереве вершины u?". Можно легко реализовать структуру данных, которая будет выполнять данные операции за время $O(n)$, где n — количество вершин в графе. Динамические деревья нужны для того, чтобы обрабатывать запросы более эффективно. В частности, все предложенные операции возможно выполнять за время $O(\log n)$.

1.3. Актуальность

Динамические деревья находят свое применение во многих областях информатики. Например, их можно использовать в алгоритмах нахождения максимального потока. Так использование динамических деревьев в алгоритме проталкивания предпотока уменьшает время его работы с $O(V^2E)$ до $O(VE\log(V^2/E))$. А время работы алгоритма Динница уменьшается с $O(V^2E)$ до $O(VE\log V)$. К сожалению, на практике их использование часто оказывается неэффективным из-за большой константы, которая скрыта в асимптотике работы. Поэтому актуальной является задача уменьшения константы в алгоритмах динамических деревьев.

Также динамические деревья используются решения для задачи о динамическом определении связности ацикличных графов. Более подробно об этом можно почитать в [???].

Кроме этого динамические деревья используются для поддержания индексов в базах данных [???].

Также динамические деревья используются для решения следующей задачи. Пусть есть алгоритм, на вход которого подаются данные, которые он обрабатывает и выдает результат. После этого некоторые входные данные изменяют и смотрят, что поменялось в работе алгоритма. В [???] показано, как с помощью динамических деревьев можно эффективно пересчитывать результат выполнения алгоритма. Примером такой задачи может служить вычисление некоторого арифметического выражения, для которого строится дерево разбора. Необходимо уметь эффективно пересчитывать значение выражения при изменении его структуры или чисел, которые в него входят.

1.4. Существующие динамические деревья

Самыми известными динамическими деревьями являются link-cut деревья, которые были предложены Слетером и Тарьяном в 1982 году. В них каждое дерево представляется как набор путей, которые не пересекаются по вершинам. Каждый такой путь в свою очередь хранится в сбалансированном дереве поиска. Если в качестве такого дерева использовать Splay-дерево, то амортизировано каждая операция с link-cut-деревом будет выполнена за $O(\log n)$. Такие деревья позволяют пересчитывать некоторые функции на путях, но не позволяют считать функции на поддереве.

Другим примером динамических деревьев является Top-деревья. Основная их идея заключается в построении сбалансированного двоичного дерева, в котором каждому листу соответствует ребро исходного дерева, а внутренней вершине — подмножество вершин исходного дерева. С помощью этого дерева можно находить наибольшее ребро на пути между вершинами за время $O(\log n)$, а также наибольшее ребро в поддереве вершины за такое же время. Также с помощью них можно пересчитывать диаметр и центр дерева, а также некоторую другую информацию о структуре дерева.

Еще один тип динамических деревьев — Rake-Compress деревья. Их отличительной особенностью является возможность их параллельного построения. Однако, реализация, которая была описана в оригинальной статье обладала существенными недостатками. Например, их можно было строить только на деревьях, у которых степень каждой вершины ограничена константой. А также константа, которая скрыта в асимптотике времени работы и используемой памяти, сильно мешает их использованию на практике.

Довольно подробно динамические деревья сравниваются в статье Тарьяна [???].

1.5. Постановка задачи

Rake-Compress деревья являются довольно перспективной структурой данных с точки зрения возможности их распараллеливания. Однако, на текущий момент они находят мало применений в реальной жизни из-за сильно большой константы во времени работы, а также используемой памяти. Из-за невозможности их использования на произвольных деревьях, на практике их место занимают другие динамические деревья. Целью данной работы стало избавление Rake-Compress деревьев от перечисленных недостатков.

Глава 2. Rake-Compress деревья

2.1. Идея

Входными данными для алгоритма Rake-Compress является лес корневых деревьев. К нему поочередно применяются операции Rake и Compress до тех пор, пока существует хотя бы одна живая вершина. После каждой операции лес сохраняется в специальном виде, который в дальнейшем дает возможность отвечать на запросы о структуре леса. Рассмотрим более подробно операции Rake и Compress. Во время операции Rake все листья дерева сжимаются к своим родителям.

Во время операции Compress выбирается некоторое множество не смежных друг с другом вершин. Причем каждая такая вершина должна иметь ровно одного сына и при этом не быть корнем дерева. Чтобы выбрать такое множество применяется следующий метод. Для каждой вершины с помощью генератора псевдослучайных чисел выбирается случайный бит. Вершина добавляется в множество, если у нее ровно один ребенок, она не является корнем и биты, которые были сгенерированы для нее, ребенка и родителя равны 0, 1 и 1 соответственно. Несложно заметить, что в выбранном таким способом множестве не будет смежных вершин.

Рассмотрим, что происходит с деревом, когда к нему приминяют операции Rake и Compress. Разобьем все вершины дерева на три группы: степени ноль, один и больше одного. Обозначим их количество за T_0, T_1 и T_2 соответственно.

Лемма 2.1. $T_2 <= T_0 - 1$

Доказательство. *Докажем по индукции по высоте дерева. Для дерева из одной вершины утверждение верно. Пусть утверждение доказано для деревьев высоты меньше h. Докажем для дерева высоты ровно h.*

Рассмотрим степень корня. Если корень имеет ровно одного ребенка, то переходим к случаю дерева высоты $h-1$. Если у дерева несколько поддеревьев, то имеем

$$T_2 = 1 + \sum_{u \in children(root)} T_2(u) \leq 1 + \sum_{u \in children(root)} (T_0(u) - 1)$$
$$\leq -1 + \sum_{u \in children(root)} T_0(u) \leq -1 + T_0 = T_0 - 1.$$

Заметим, что для леса деревьев лемма также справедлива.

Лемма 2.2. *После применения операций Rake и Compress к лесу, количество вершин в нем в среднем уменьшится в константное число раз.*

Доказательство. *В среднем будет удалено* $deleted = T_0 + \frac{T_1}{8}$ *вершин. Из леммы 2. 1. получаем*

$deleted \geq \frac{1}{2}(T0 + T2) + \frac{1}{8}T1 \geq \frac{1}{8}(T0 + T1 + T2)$

Теорема 2.3. *Математическое ожидание количества операций Rake и Compress, которые будут выполнены до полного сжатия дерева, равно* $O(\log n)$*, где* n *— общее количество вершин.*

Доказательство. *Из леммы 2. 2. известно, что после каждой итерации применения операций Rake и Compress число вершин в среднем уменьшается в константное число раз. Значит, количество итераций в среднем ограничено* $O(\log n)$*.*

Из леммы 2. 2. также следует, что суммарное число живых вершин по всем итерациям в среднем равно $O(n)$. Это утверждение будет полезно для оценки количества используемой памяти.

2.2. Хранение Rake-Compress деревьев

Для того, чтобы отвечать на запросы относительно структуры леса необходимо сохранить информацию о том, как происходил процесс сжатия леса. Для этого предлагается сделать следующее. Будем хранить таблицу. В каждой ее строке будет храниться состояние леса после применения опе-

рации Rake или Compress. Количество строк в такой таблице будет в среднем $O(\log n)$. Каждый столбец таблицы будет соответствовать некоторой вершине. А каждая клетка будет хранить состояние некоторой вершины после какой-то итерации. Состоянием вершины будем считать ее родителя, а также множество детей. Если же вершина уже была сжата к этому моменту, то пометим это специальным образом.

Заметим, что суммарное количество памяти, которое необходимо для хранение такой таблицы равно $O(n \log n)$. Если же в каждой строке хранить только существующие вершины с помощью ассоциативного массива, то количество используемой памяти уменьшится до $O(n)$. Однако, на практике использование ассоциативного массива оказывается нецелесообразным из-за большой скрытой константы.

2.3. Возможность параллельного построения

Заметим, что и операцию Rake и операцию Compress можно выполнять параллельно для всех вершин. Единственная часть алгоритма, которую непонятно, как реализовать параллельно, является пересчет множества детей вершины при переходе к следующему слою. В оригинальной статье степень каждой вершины считалась константой и множества можно было хранить и пересчитывать любым способом. Если же предположить, что множество детей можно пересчитывать за $O(1)$, то Rake-Compress дерево можно построить за $O(\log n)$ при условии наличия $\Omega(n)$ процессоров.

2.4. Ответ на запросы про структуру леса

Рассмотрим, как с помощью таблицы, которая была описана ранее отвечать на запросы. Самым простым запросом является “определить корень дерева, в котором находится вершина v”. Чтобы ответить на этот запрос необходимо проследить за вершиной v в таблице. Будем переходить

по строчкам таблицы начиная с первой. Если в некоторой строке нет клетки, которая соответствует вершине v, то необходимо перейти к родителю вершины v на предыдущей строке. Если же у нее не было родителя, то значит вершина v и является корнем.

Чтобы проверить, находятся, ли две вершины в одном дереве, необходимо найти корни деревьев, в которых находятся обе вершин и сравнить их.

Для ответа на более сложные запросы, при сжатии вершины можно в соответствующую клетку таблицы записывать дополнительную информацию, а потом пользоваться ей. Заметим, что ответ на запрос происходит за время $O(\log n)$, так как на каждой строке таблицы тратится константное время.

2.5. Перестроение Rake-Compress дерева при изменении структуры леса

Динамические Rake-Compress деревья основываются на следующей теореме.

Теорема 2.4. *При добавлении или удалении одного ребра из леса, в таблице, которая соответствует лесу, поменяется в среднем $O(\log n)$ клеток. Считается, что клетка поменялась, если поменялся родитель или множество детей. Также, если клетка поменялось, то считается, что и все клетки, которые соответствуют этой вершине в более позднее время также поменялись.*

Доказательство. *Доказательство этой теоремы можно прочитать в оригинальной статье про Rake-Compress деревья. Хоть в самой статье авторы рассчитывают на то, что степень каждой вершины ограничена константой, в конкретно этом доказательстве они этим не пользуются.*

Таким образом, если научиться находить каждую из $O(\log n)$ изменившихся клеток за $O(1)$, то можно будет обрабатывать добавление и удаление ребер за $O(\log n)$ времени.

2.6. Определение изменившихся клеток

В первой строке таблицы изменились только клетки, которые соответствуют концам удаленного или добавленного ребра. Если на некотором слое известно множество изменившихся клеток, то в аналогичное множество на следующем слое могут войти только вершины смежные с данными. В случае наличия ограничения не степень вершин можно проверить все соседние вершины. В рассматриваемом случае необходимо действовать немного сложнее. Для этого будем честно определять, состояние каких вершин могло измениться. Необходимо рассмотреть моменты времени, когда вершина сжалась к своему родителю (как минимум в одной из версий). В любом случае, если вершина удаляется, то у нее не более двух соседей, которые и нужно пометить как изменившиеся.

Однако, после обнаружения, что клетка изменилась, необходимо также пересчитать ее состояние. В оригинальной статье говорится, что это можно сделать за константное время, так как количество детей ограничено. В нашем же случае нельзя пересчитывать состояние заново, поэтому необходимо узнать, как именно оно изменилось и поменять его за время пропорциональное количеству изменений. Однако, проблема состоит не только в сложности реализации, но и в том, что изменения необходимо внести и в клетки, которые соответствуют вершине в более поздний момент времени. Если хранить множества детей с помощью структуры данных, которая позволяет добавлять и удалять элементы за $O(1)$, то общее время обработки одного добавления или удаления ребра будет $O(\log^2 n)$.

Глава 3. Оптимизация Rake-Compress деревьев

3.1. Оптимизация памяти

В разделе [???] говорилось, что Rake-Compress дерево принято хранить в виде таблице из $O(\log n)$ строк и n столбцов. Суммарное количество непустых клеток $O(n)$, однако, их хранение с помощью ассоциативного массива, хоть и уменьшает асимптотику до $O(n)$, значительно увеличивает константу во времени работы. Поэтому на практике такой подход не используется.

Предлагаемая оптимизация заключается в том, чтобы хранить таблицу по столбцам, а не по строкам. Каждый столбец соответствует некоторой вершине. Заметим, что каждая вершина жива только первые несколько применений операций Rake и Compress. Если же в какой-то момент вершина была сжата к своему родителю, то далее она не станет живой.

Также следует отметить, что в случае изменения структуры леса, который хранится в дереве, необходимо пересчитывать не более $O(\log n)$ столбцов, каждый из которых занимает $O(\log n)$ памяти. Здесь предполагается, что множества детей занимают $O(1)$ памяти, что в случае реализации, предложенной в оригинальной статье, неправда. Хранение таблицы в таком виде позволяет процессору сохранить нужные столбцы в кеше.

Более подробно сравнение произвольности двух версий изложено в главе [???].

3.2. Хранение множества детей

Рассмотрим более подробно, какие операции должна поддерживать структура данных, которая хранит в каждой клетке текущее множество детей вершины. Необходимо за $O(1)$ уметь добавлять и удалять вершины в множество. Множество детей необходимо хранить для того, чтобы иметь возможность определять, можно ли применить операцию Compress к вершине. Для этого в этом множестве должен быть ровно один элемент и значение бита, который был сгенерирован для единственного элемента этого множества, должно быть равно единице. Значит, необходимо уметь определять, кто находится в множестве только в том случае, если в нем не более одного элемента.

Поэтому можно хранить всего две величины. А именно, количество элементов в множестве и сумму номеров вершин в множестве. Тогда для добавления вершины в множества необходимо увеличить счетчик количества вершина, а также добавить к сумме номер вершины. Для удаления вершины из множества необходимо соответственно уменьшить счетчик количества вершин, и вычесть номер вершины из суммы.

Данная оптимизация позволяет существенно уменьшить константу в асимптотике времени работы, а также используемой памяти. Также она позволяет избавиться от проблемы, о которой говорилось в предыдущем разделе. А именно, вместо того, чтобы изменить множество детей в $O(\log n)$ версиях вершины, можно запомнить "разницу" между множествами и передавать ее в следующий слой за $O(1)$. Это позволяет перестраивать Rake-Compress дерево за $O(\log n)$ времени вместо $O(\log^2 n)$.

3.3. Пересчет состояний вершин в таблице Rake-Compress дерева

Рассмотрим более подробно тонкости реализации Rake-Compress деревьев. В каждой клетке таблицы хранится состояние вершины в некоторый момент времени. В состояние вершин входит ее родитель, а также множество детей, которое хранится в виде суммы их номеров, а также их количества. Также в состоянии будем запоминать, как изменяется состояние при переходе к следующему слою. А именно, будем хранить, кто должен стать новым родителем, на сколько изменится количество детей, а также как изменится сумма их номеров. Все это необходимо для того, чтобы обрабатывать каждую изменившуюся клетку за $O(1)$.

Как только некоторая вершина помечается как изменившаяся, отменим ее действие на таблицу. А именно, найдем момент времени, когда вершина сжимается к родителю (если хранить таблицу по столбцам, это можно сделать за $O(1)$). Рассмотрим, какие вершины поменяются при сжатии данной. Это ее родитель (если он есть), а также сын (если он есть). Для каждой из этих вершин поменяем значения изменений, которые необходимо применить к состоянию. Также пометим, что эти вершины поменялись на этом слое. Для этого на каждом слое будем хранить список вершин, которые на нем поменялись. А перед тем как обрабатывать очередной слой будем добавлять в множество изменившихся вершин вершины из соответствующего списка.

При этом необходимо удостоверится, что вершина еще не находится в множестве. Это можно сделать простых проходом по всему множеству (так как математическое ожидание количества вершин в нем равно константе). Однако, есть и более эффективный с практической точки зрения метод. Будем считать, что к структуре данных поступают запросы о добавлении и удалении ребер. Тогда для каждой вершины будем запомнить

номер последнего запроса, в котором вершина была помечена изменившейся. В таком случае можно будет узнать, лежит ли вершина в множестве изменившихся за одно обращение к массиву.

Кроме удаления эффекта от изменившихся вершин также необходимо и добавить правильный эффект. Для этого будем для каждой из изменившихся вершин определять, как ее состояние меняется при переходе к следующему слою. Если вершина сжимается к ее родителю, то пометим родителя и ребенка (если он есть) и поменяем изменение, которое хранится в соответствующих клетках. А для пересчета состояния клеток воспользуемся значениями изменений, которые сохранены в клетках.

Более подробно с деталями реализации можно познакомится на [github].

3.4. Случай неориентированного дерева

Ранее был рассмотрен только случай корневых деревьев, однако Rake-Compress деревья можно строить и в неориентированном случае. Операция Rake по-прежнему удаляет все лисья дерева. А операция Compress применяется для вершин, к которых ровно две смежные вершины. Предложенная оптимизация памяти также применима в данном случае. А чтобы хранить множество вершин необходимо приложить чуть больше усилий.

Отличие неориентированного случае заключается в том, что необходимо узнавать номера смежных вершин в случае, когда их становится не больше двух. А, значит, хранить только сумму их номеров недостаточно. Поэтому предлагается кроме этого хранить сумму квадратов их номеров. Тогда, чтобы узнать, какие вершины принадлежат множеству необходимо будет решить квадратное уравнение.

3.5. Пересчет аддитивных функций

Кроме запросов о структуре леса, Rake-Compress деревья можно использовать для подсчета значений некоторых функций. Например, каждой вершине можно сопоставить некоторое значение и узнавать, чему равна сумму значений всех вершин, которые находятся в поддереве.

Для этого в клетках таблицы Rake-Compress дерева необходимо хранить не только состояние вершины, но и значение функции. Если функция является аддитивной, то ее пересчет аналогичен пересчету множества детей вершины. Так, если некоторая вершина сжимается к родителю, то в соответствующей родителю клетке необходимо обновить значение функции. При добавлении и удалении ребер необходимо в изменившихся клетках пересчитывать значение функции. Аддитивность функции позволяет нам отменять сжатие вершины к своему родителю за $O(1)$. Поэтому общее время работы не изменится и будет составлять $O(\log n)$ на одно обновление.

Заметим также, что как и в случае с множеством детей, в каждой клетке необходимо хранить, как изменяется функция при переходе к следующему слою.

3.6. Пересчет ассоциативных функций

Если же необходимо считать значение функции, которая не является аддитивной, то можно воспользоваться следующим методом. В каждой клетке можно хранить все изменения функции, которые произошли из-за сжатия детей в виде сбалансированного двоичного дерева. Тогда при изменении множества детей, необходимо также соответствующим образом изменить дерево. Это приведет к тому, что в худшем случае пересчет таблицы Rake-Compress дерева при добавлении или удаления одного ребра будет работать за $\theta(\log^2 n)$.

Глава 4. Сравнение

4.1. Надо дописать прогу, тогда тут что-то появится

Заключение

В данной работе была детально изучена структура данных Rake-Compress дерево. Она позволяет хранить информацию о лесе корневых деревьев, а также эффективно обрабатывать запросы его изменения (добавление и удаление ребер).

Реализация Rake-Compress деревьев, которая была описана в исходной статье, была существенно модифицирована. В частности, было показано как сократить расход памяти с $O(n \log n)$ до $O(n)$ без потери скорости работы.

Описанная в статье структура данных работала только в том случае, если степень каждой вершины ограничена некоторой константой. Было показано как избавиться от этого ограничение не ухудшив время работы.

Также было показано, как применить предложенные оптимизации в случае неориентированных деревьев.

Кроме того, был предложен способ пересчета функций на лесе деревьев. Для аддитивных функций пересчет осуществляется за время $O(\log n)$, а для ассоциативных за $O(\log^2 n)$.

Структура данных Rake-Compress дерево с предложенными оптимизациями была реализована на языке программирования Java. Было проведено сравнение скорости работы и потребляемой памяти разных версий Rake-Compress деревьев.