САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Решения задач связности и двусвязности на динамически меняющихся графах

Дипломная работа студента 545 группы
Копелиовича Сергея Владимировича

Научный руководитель	. . . . . /подпись/	старший преподаватель Лопатин А.С.
Рецензент	. . . . . /подпись/	доцент кафедры КТ НИУ ИТМО Станкевич А.С.
“Допустить к защите” заведующий кафедрой,	. . . . . /подпись/	д.ф.-м.н., проф. Терехов А.Н.

Санкт-Петербург
2012

SAINT PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

Offline solution of connectivity and 2-edge-connectivity problems for fully dynamic graphs

Sergey Kopeliovich

Master's thesis

Supervisor	. . . . .	Senior lecturer A.S. Lopatin
Reviewer	. . . . .	Associated Professor of NRU ITMO A.S. Stankevich
"Approved by" Head of Department,	. . . . .	Professor A. N. Terekhov

Saint Petersburg
2012

Содержание

1 Краткий обзор

Для задачи dynamic connectivity problem существует простое Offline решение за время $O(k \log k)$, где k — общее количество запросов. Это решение в 1992-м году предложил David Eppstein, подробное описание можно найти в статье [3].

Для задачи dynamic 2-edge-connectivity problem лучшее Online решение предложил в 2000-м году Mikkel Thorup [4]. Запрос изменения графа обрабатывается за время $O(\log^3 n \log \log n)$. Offline решений, превосходящих данное по асимптотике, не существует.

Данная работа описывает простое Offline решение задачи dynamic 2-edge-connectivity problem, работающее за время $O(k \log k)$, где k — общее количество запросов.

Кроме нового алгоритма за $O(k \log k)$ для dynamic 2-edge-connectivity problem в данной работе описаны различные подходы к построению Offline решений dynamic graph problems. Подходы проиллюстрированы на примере dynamic connectivity problem.

2 Введение

2.1 Постановка задачи

В этой работе речь пойдет о задачах поиска компонент связности и компонент реберной двусвязности в динамически меняющихся неориентированных графах. Графом G называется $\langle V, E \rangle$, где V — множество вершин графа, $E \subset V \times V$ — множество ребер графа. $|V|$ обозначим за n, $|E|$ обозначим за m.

Запросы изменения графа:

1. Добавить ребро
2. Удалить ребро

Такие запросы я, следуя терминологии, введенной в англоязычной литературе, буду называть *update-запросами*.

Запросы к текущему состоянию структуры данных:

1. Для двух вершин определить, лежат ли они в одной компоненте связности (и, соответственно, реберной двусвязности).
2. Для графа сказать, сколько в нем компонент связности (реберной двусвязности).

Такие запросы я буду называть *get-запросами*.

2.2 Обозначения

Общее количество запросов (и update-запросов, и get-запросов) я буду обозначать символом k.

Для простоты оценки времени сложности описанных ниже алгоритмов, предположим, что граф изначально пуст. Если мы хотим, чтобы в нем изначально уже было m ребер, можно добавить их, сделав дополнительные update-запросы добавления.

В некоторых алгоритмах для анализа сложности мной используется обратная функция Аккермана. Причина ее появления в моей работе — применение СНМ (Системы Непересекающихся Множеств). Как известно из[1], амортизационное время на один запрос у этой структуры данных равно $\alpha(n, m)$. В этой работе для краткости и красоты формул я буду использовать обозначение α.

В работе я пользуюсь терминами offline задача, online задача. При разработке структур данных, отвечающих на пользовательские запросы возможна одна из двух ситуаций:

1. Все запросы известны заранее. В таком случае это offline задача.
2. Перед ответом на следующий запрос, нужно обязательно ответить на предыдущий. В таком случае это online задача.

2.3 Англоязычные термины

На протяжении всей работы я буду пользоваться названиями задач, заимствованными из англоязычной литературы.
Задачи, заключающиеся в поддержке

1. компонент связности
2. компонент реберной двусвязности
3. минимального по весу остовного дерева

на динамически меняющихся графах, в англоязычной литературе называются, соответственно

1. fully dynamic connectivity
2. fully dynamic 2-edge-connectivity
3. fully dynamic minimum spanning tree

Термин «минимальное по весу остовное дерево» я для краткости буду обозначать MST (сокращение от minimum spanning tree).

2.4 Кратко о последующих главах

Работа состоит из нескольких глав. Разбиение на главы сделано таким образом, что читать главы следует последовательно.

В главе [3] подробно описана история исследования задачи. В главе [4] описаны вспомогательные структуры данных, используемые затем в решениях задач fully dynamic connectivity и 2-edge-connectivity.
В главе [5] дано четкое описание формата входных данных алгоритмов, которые будут решать задачи fully dynamic connectivity и 2-edge-connectivity.
Далее в главе [6] подробно описаны известные мне offline решения fully dynamic connectivity problem. В конце этой главы [6.5] дано описание нового offline алгоритма.
Наконец, в главе [7] представлена самая интересная часть работы — новое offline решение fully dynamic 2-edge-connectivity problem. В той же главе, в части [7.1], описано оптимальное решение задачи в случае когда ребра только добавляются в граф.

3 Существующие решения

Прежде всего стоит отметить, что описание существующих решений уже хорошо изложено в [5] и в [4].

Теорема 3.1 : *Если задача fully dynamic minimum spanning tree имеет решение, обрабатывающее k запросов за время $f(k)$, то задача fully dynamic connectivity, имеет решение, обрабатывающее k запросов за время $O(f(k))$*
Доказательство: подробно о сведении можно прочитать в [2] (стр. 21-22).

Далее я приведу достижения по решению задачи fully dynamic connectivity problem и fully dynamic 2-edge-connectivity problem в хронологическом порядке. Почти все оценки времени работы, указанные ниже, амортизационные.

Задача fully dynamic connectivity

Читая нижеприведенный список, важно помнить теорему [3.1].

1985 – Fredrickson изобрел новую структуру данных, topology trees. Эта структура данных позволяет решать задачу fully dynamic minimum spanning tree за время $O(\sqrt{m})$ на один update-запрос и $O(\log n)$ на один get-запрос [2].

1992 – Eppstein et al. улучшили время на update-запрос до $O(\sqrt{n})$. Один get-запрос обрабатывается в этом алгоритме за время $O(1)$. Для этого они использовали sparsification technique [6].

1992 – Eppstein предложил Offline алгоритм для решения fully dynamic minimum spanning tree, обрабатывающий k update-запросов за время $O(k \log k)$ [3].

1995 – Monika Henzinger и Valerie King предложили первый полилогарифмический алгоритм для решения fully dynamic minimum spanning tree. Алгоритм был рандомизированным и update-запросы обрабатывал за $O(\log^3 n)$, а get-запросы за время $O(\log n / \log \log n)$ [13].

1996 – Monika Henzinger и Mikkel Thorup улучшили рандомизированную идею 1995-го года до $O(\log^2 n)$ на запрос.

1997 – Monika Henzinger и Valerie King предложили детерминированный алгоритм обрабатывающий update-запрос за время $O(n^{\frac{1}{3}} \log n)$ и get-запрос за время $O(1)$ [11]

1998 – Mikkel Thorup, Jacom Holm, Kristian Lichtenberg предложили детерминированный алгоритм с оценкой $O(\log^2 n)$ на update-запрос, $O(\log n / \log \log n)$ на get-запрос и используемой памятью $O(m+n \log n)$[5].

2000 – Mikkel Thorup предложил детерминированный алгоритм с оценкой $O(\log n \log \log^3 n)$ на update-запрос, $O(\log n / \log \log \log n)$ на get-запрос и используемой памятью $O(m)$ [4]. В той же работе Mikkel Thorup улучшил используюмую память алгоритма 1998-го года до $O(m)$.

2004 – Mihai Patrascu и Eric Demaine показали доказали оценку для Online алгоритмов: если update-запрос обрабатывается за время $O(\log n \cdot x)$ (где $x > 1$), то get-запрос обрабатывается за время $\omega(\log n / \log x)$ [7].

Итог можно подвести следующей таблицей:

Год	Время обработки k запросов	Комментарий
1992	$O(k \log k)$	Offline решение через сведение к MST
2000	$O(k \log k \log \log^3 k)$	Лучшее online решение
2012	$O(k \log k)$	Offline решение, не использующее MST

В последней строке таблицы приведено достижение, описанное в данной работе. Оно не улучшает асимптотику, но проще чем идея предложенная Дэвидом Эппштейном.

Задача fully dynamic 2-edge-connectivity

1991 – Fredrickson обобщил свою идею для 1-connectivity до 2-edge-

connectivity. Update-запрос обрабатывался за время $O(\sqrt{m})$, get-запрос за время $O(\log n)$ [2].

1993 – Eppstein, применив sparsification technique, получил время $O(\sqrt{n})$ на один update-запрос [6].

1997 – Monika Henzinger и Valerie King предложили рандомизированный алгоритм, обрабатывающий update-запросы за время $O(log^5 n)$ [12].

1998 – Mikkel Thorup, Jacom Holm, Kristian Lichtenberg предложили детерминированный алгоритм, обрабатывающий update-запросы за время $O(log^4 n)$ [5].

2000 – Mikkel Thorup предложил детерминированный алгоритм с оценкой $O(\log^3 n \log\log n)$ на update-запрос, $O(\log n)$ на get-запрос и используемой памятью $O(m)$ [4].

Итог можно подвести следующей таблицей:

Год	Время обработки k запросов	Комментарий
1997	$O(k \log^5 k)$	Первое полилогарифмическое online решение
2000	$O(k \log^3 k \log\log k)$	Лучшее online решение
2012	$O(k \log k)$	Offline решение

В последней строке таблицы приведено достижение, описанное в данной работе.

Так же, стоит заметить, что я не обнаружил достижений по улучшению асимптотики худшего случая времени работы, появившихся после 2001-го года. Например, я просмотрел все публикации по ключевому слову *Dynamic* с конференций *ACM-SIAM Symposium on Discrete Algorithms* за 2001-2012 года, а также публикации авторов David Eppstein, Mikai Patrascu, Mikkel Thorup, Valerie King, Monika Henzinger, Giuseppe Italiano, Eric Demaine, т.е. всех тех, чьему авторству принадлежат существующие ныне оптимальные алгоритмы и оценки из области Dynamic Connectivity. В дополнении с только что приведенной таблицей этот факт представляется мне весьма удивительным.

4 Вспомогательные структуры данных

4.1 Система Непересекающихся Множеств

Для некоторых решений fully dynamic connectivity problem нам понадобится структура данных *Система Непересекающихся Множеств*

(или, более кратко, СНМ). Подробнее о ней и об оценках времени работы можно прочесть в книге Кормена [1].
Структура оперирует с множеством V. Начинает она с того, что каждый элемент V лежит в отдельном множестве.

```
1. Init()
2.     for v ∈ V
3.         p[v] = v, size[v] = 1
```

`size[v]` — размер множества, `p[v]` — ссылка вершины на следующую. Таким образом, каждое множество представлено в виде дерева. Корнем дерева является вершина, ссылающаяся на себя саму (`p[v] = v`).
Операция *Get* возвращает представителя множества. Для всех вершин одного множества результат будет одним и тем же. Для двух вершин разных множеств результат будет разным. С помощью процедуры *Get* можно определять, лежат ли две вершины в одном множестве.
Представителем дерева у нас будет его корень. Собственно *Get* поднимается от любой вершины v до корня и возвращает его.
Здесь также используется эвристика сжатия путей.

```
1. Get(v)
2.     return v == p[v] ? v : (p[v] = Get(p[v]))
```

Операция *Join* объединяет два множества в одно. Множество, в котором лежит вершина a, и множество, в котором лежит вершина b.
Для этого *Join* ссылку одного из двух корней направляет на второй.
Здесь используется эвристика «выгодно меньшее множество подвешивать к бо́льшему»

```
1. Join(a, b)
2.     a = Get(a)
3.     b = Get(b)
4.     if a = b then
5.         return // вершины лежат в одном множестве
6.     if size[a] < size[b] then
7.         swap(a, b)
8.     // теперь a является корнем бо́льшего множества.
9.     p[b] = a
10.    size[a] += size[b]
```

И *Join*, и *Get* работают за амортизированное время $O(\alpha)$. Доказательство можно прочесть в первоисточнике — работе Роберта Тарьяна 1975 года [9] или в Кормене [1].

4.2 Структуры данных с откатами назад

Следующий инструмент, который нам потребуется, — структуры данных с возможностью отката к предыдущим версиям.

На самом деле, любую структуру данных можно научить откатывать состояние к предыдущим версиям. Для этого достаточно хранить стек пар ⟨адрес, значение⟩ — адреса изменяемых ячеек памяти и старые значения этих ячеек. Теперь каждую операцию присваивания нужно обернуть в специальную функцию, которая перед изменением памяти кладет нужную информацию в стек.

Откатиться к предыдущей версии = снять со стека несколько последних операций и выполнить их в обратном порядке.

Запомнить версию (чтобы в будущем можно было к ней откатываться) = запомнить указатель на вершину стека.

Откат — операция необратимая, если у нас были версии $V_1, V_2, \ldots, V_9$ и мы откатимся к версии V_2, все версии кроме V_1 и V_2 будут навсегда забыты.

Теорема 4.1 : *Пусть есть две версии — A и B. B была порождена из A. Время, требуемое на откат от B к A, асимптотически не превышает времени, требуемого на внесение изменений при порождении B из A*
Доказательство: Пусть во время внесения изменений, у нас есть только одна операция для изменения памяти — `set`, меняющая значение ровно одного байта. Время требуемое на откат = O(количества вызовов процедуры `set`) = O(времени, потраченного на внесение изменений).
Следствие: В оценках асимптотики времени работы алгоритмов временем работы откатов к предыдущим версиям можно пренебречь.

5 Формат входных данных

В этой главе мы сформулируем формат входных данных и сделаем первый шаг в решении произвольной задачи из области dynamic graphs — преобразуем формат входных данных к удобному для работы виду.

Итак, входные данные. Поступают update-запросы двух видов — `"ADD a b"`, `"DEL a b"`. Гарантируется, что к моменту, когда поступает запрос `"DEL a b"`, ребро в графе присутствует. Нужно после каждого update-запроса знать текущее количество компонент связности (или, соответственно, реберной двусвязности) и уметь отвечать на get-запрос

`"GET a b"` — лежат ли вершины a и b в одной компоненте связности (или, соответственно, реберной двусвязности).

Пусть общее количество запросов (и get, и update) равно k. Запросы пронумеруем числами от 0 до $k-1$ ($q_0, q_1, \ldots, q_{k-1}$). Будем говорить, что у нас есть $k+1$ момент времени (в i-й момент времени уже выполнены все update-запросы с номерами от 0 до $i-1$). Будем также говорить, что i-й зарос случился в момент времени i.

Мы решаем задачу Offline, т.е. нам заранее доступны все запросы. Поэтому можно все update-запросы модифицировать следующим образом: ребро $\langle a, b\rangle$ в момент времени L добавилось, в момент времени R удалилось. L — номер запроса `"ADD a b"`. R — номер запроса `"DEL a b"`. Если какое-то ребро в конце не удалилось, скажем, что оно удаляется в момент времени $k+1$.

Теперь у нас вместо пары запросов `ADD`, `DEL` есть один запрос $\langle a, b, L, R\rangle$ — добавить ребро $\langle a, b\rangle$ так, чтобы оно присутствовало во все моменты от L+1 до R включительно. Данное преобразование update-запросов, если мы использовали сортировку подсчетом, заняло у нас $O(k+n)$ времени. Новые update-запросы будем также называть ребрами-запросами.

В итоге у нас есть запросы типа get (q.type = `get`), и запросы-ребра (q.type = `update`). У update-запросов есть свойства q.a, q.b — номера вершин, которые соединяет ребро и q.L, q.R — моменты времени, когда ребро добавлялось и удалялось, соответственно.
Входные данные — массив запросов q.

6 Связность

В этой части работы я полностью разберу множество offline решений dynamic connectivity problem. Все подходы, описанные ниже, так или, иначе, можно обобщить до решения dynamic 2-edge-connectivity problem.

6.1 Только добавление ребер: Online решение за $O(\alpha)$ на запрос

Если ребра только добавляются в граф, решением является в чистом виде CHM. Секрет остальных offline решений заключается в том, что добавление ребра — дешевая операция, а удаление можно не делать, если удаляемое ребро в граф не добавлять.

6.2 Offline решение за $O(k)$ на запрос

Для каждого момента времени возьмем все те ребра, которые в соответствующий момент времени добавлены и выделим компоненты связности за $O(k)$. Каждая компонента связности покрашена в свой цвет, что позволяет теперь за $O(1)$ отвечать на get-запросы.

6.3 Offline решение за $O(\sqrt{k}\alpha)$

Разобьем моменты времени на группы (отрезки) по $\sqrt{k}$ моментов в каждой группе. Научимся обрабатывать за $O(k)$ все get-запросы в любой из групп. Пусть наша группа — это отрезок времени $[L..R]$. Давайте возьмем все ребра-запросы, которые во все моменты времени от L до R присутствуют в графе. Т.е. $q_i : (q_i.L < L)$ `and` $(q_i.R \geqslant R)$. Добавим эти ребра в граф. Добавление ребра обрабатывается системой непересекающихся множеств. Чем отличаются моменты времени $L..R$? Тем, что некоторые ребра-запросы присутствуют в графе не во все моменты времени от L до R. Это такое q_i, что $(q_i.L < R)$ `and` $(q_i.R \geqslant L)$.

Чтобы получить теперь структуру данных, умеющую отвечать на get-запросы для момента времени $i \in [L..R]$, нужно сделать не более $\sqrt{k}$ добавлений ребер. После этого все сделанные изменения следует отменить (откатить, см. [4.2]).

Чтобы $\sqrt{k}$ добавлений ребер работали за $O(\sqrt{k}\cdot\alpha)$ нужно предварительно сделать сжатие всех путей за время $O(k)$.

Алгоритм А:

1. tn := k + 1
2. M := $[\sqrt{k}]$
3. **for** (i = 0; i · M < tn; i++) {
4. L = i · M
5. R = min(i · M + M - 1, tn - 1)
6. // обрабатываем отрезок моментов времени [L..R]
7. множество ребер-запросов A = { $q_i : q_i.L < L$ `and` $q_i.R \geqslant R$ }
8. множество ребер-запросов B = { $q_i : q_i.L < R$ `and` $q_i.R \geqslant L$ }
9. // оба множества мы нашли за время $O(k)$, |B| = $O(\sqrt{k})$
10. инициализация CHM
11. **for** $e \in A$: Join(e.a, e.b) — добавляем ребра
12. **for** $e \in A$: Get(e.a), Get(e.b) — сжатие путей
13. текущее состояние CHM: base
14. **for** $t \in [L..R]$ {

```
15.             for e ∈ B
16.                 if e.L < t and e.R ⩾ t
17.                     Join(e.a, e.b)
18.             // новые ребра мы добавили за время O(√k)
19.             сейчас можно ответить на все get-запросы
20.             откатываем состояние СНМ к моменту base
21.         }
22. }
```

6.4 Offline решение за $O(\log^2 k / \log\log k)$

Обобщим идею, описаную выше, до дерева отрезков (segment tree) [10] на моментах времени. Чтобы обработать все моменты времени $[L..R]$, можно сперва обработать моменты времени $[L..M]$, а затем моменты времени $[M{+}1..R]$, где $M = [\frac{L+R}{2}]$. Решением задачи будет рекурсивная процедура, которая получает некоторый отрезок времени $[L..R]$ и массив запросов. На каждом шаге эта процедура добавляет в граф все ребра-запросы, которые существуют во все моменты времени от L до R включительно. Далее процедура делит отрезок времени пополам, и от каждой половины вызывается рекурсивно, передавая внутрь, то множество запросов-ребер, которые в какой-то момент еще предстоит добавить в граф.

Алгоритм B:

```
1.  go( queries, L, R ) {
2.      if L = R
3.          for q ∈ queries
4.              if (q.type = get)
5.                  q.answer := (Get(q.a) == Get(q.b))
6.          return
7.      текущее состояние СНМ: base
8.      queries_2 := ∅
9.      for q ∈ queries
10.         if (q.type = update) and (q.L < L) and (q.R ⩾ R)
11.             Join(q.a, q.b)
12.         else if (q.type = get) or (q.L < R) and (q.R ⩾ L)
13.             queries_2.add(q)
14.     M := [(L+R)/2]
15.     go(queries_2, L, M)
16.     go(queries_2, M+1, R)
17.     откатываем состояние СНМ к моменту base
```

```
18. }
19. main() {
20.         инициализация CHM
21.         go(all_queries, 0, k)
22. }
```

Оценим время работы алгоритма B.

Теорема 6.1 : $|queries_2| \leqslant R - L + 1$
Доказательство: если ребро-запрос попало в $queries_2$, значит или $q.L \in [L..R]$, или $q.R \in [L..R]$. А поскольку все координаты L и R попарно различны (за исключением фиктивного R, появляющегося, если ребро так и не будет удалено), то размер множества $queries_2$ не больше длины отрезка $[L..R]$.
Следствие : $|queries| \leqslant R - L + 1$

Теорема 6.2 : *Суммарная длина массивов queries по всем состояниям рекурсии равна* $O(k \log k)$
Доказательство: глубина рекурсии $\log k$. На каждом уровне рекурсии каждый момент времени встречается не более чем в одной ветви рекурсии, т.е. на одном уровне рекурсии суммарная длины отрезков времени $= O(k)$. Следовательно суммарная длина отрезков времени по всем состояниям $= O(k) \cdot \log k = O(k \log k)$. А, значит, по теореме [6.1], суммарная длина массивов *queries* тоже не более $O(k \log k)$.

Теорема 6.3 : *Каждый get-запрос обработается ровно один раз, в соответствующем листе дерева рекурсии.*

Теорема 6.4 : *Алгоритм B работает за время* $O(k \log k \cdot tJoin + k \cdot tGet))$. *Где* $tJoin$ *— время работы операции Join в CHM, а* $tGet$ *— время работы операции Get в CHM.*
Доказательство: Запрос Get к CHM по теореме [6.3] вызывается не более k раз. Запрос Join к CHM вызывается не более одного раза для каждого элемента массива *queries* значит, по теореме [6.2], запрос Join к CHM вызывается $O(k \log k)$ раз.

Осталось оценить время работы CHM. Реализация CHM, описанная ранее в главе [4.1] устроена так, что Join вызывает Get. Поэтому $tGet = O(tJoin)$, и по сути нас интересует время работы только одного запроса Join. Один запрос к CHM работает, казалось бы, за амортизационное время $O(\alpha)$, но в нашем случае эвристика сжатия путей в худшем случае не работает, т.к., сделав сжатие путей в некоторой версии CHM, сжатие путей во всех родительских версиях мы не делаем. Поэтому, когда мы сделаем откат к родительской версии, эффект сжатия путей пропадет.

На самом деле нас интересует не амортизационное время работы, а время на один запрос в худшем случае. Наша СНМ дает оценку в худшем случае на один запрос $\theta(\log k)$. Лучшая же с точки зрения «время работы одного запроса в худшем случае» реализация СНМ дает время $\theta(\log k / \log\log k)$ на один запрос [8]. Получается, что суммарное время выполнения всех $O(k \log k)$ операций с СНМ равно $O(k \log^2 k / \log\log k)$

6.5 Offline решение за $O(\log k)$

Теорема 6.5 : *Сжатие компонент связности*
Если в графе $G = \langle V, E\rangle$ всегда присутствует набор ребер A (нет update-запросов, удаляющих ребра из A), то можно построить новый граф G_2, вершинами которого будут компоненты связности графа $\langle V, A\rangle$, а множество ребер $= \{\langle color[a], color[b]\rangle : \langle a, b\rangle \in E \setminus A\}$. Здесь $color[a]$ — номер компоненты связности, в которой лежит вершина a. При переходе к новому графу запросы (и update-запросы, и get-запросы) следует поменять следующим образом: $a \to color[a]$ для всех вершин a, участвующих в запросе. **Утверждение теоремы:** результаты новых запросов типа `get` на графе G_2 будут совпадать с результатами старых запросов на графе G.

Доказательство:
1. Добавим в граф G_2 петли, тогда любому пути $p_1, p_2, \ldots, p_k$ в графе G соответствует путь $color[p_1], color[p_2], \ldots, color[p_k]$ в графе G_2. Значит, если a и b связны в G, то $color[a]$ и $color[b]$ связны в G_2.
2. Пусть в графе G нет пути из a в b, значит, после добавления новых ребер не появилось пути из компоненты связности a в компоненту связности b. Поэтому в графе G_2 вершины $color[a]$ и $color[b]$ не связны.

Теорема 6.6 : *Редукция графа*
Если сейчас G пуст (не содержит ребер), а запросы затрагивают только множество вершин B, то можно все запросы рассматривать, как запросы к меньшему графу $G_2 = \langle B, \varnothing\rangle$. **Утверждение теоремы:** Количество вершин в графе $G_2 = O(k)$, где k — общее количество запросов. Результаты get-запросов над G и G_2 одинаковы.

Доказательство:
1. $|B| \leqslant 2k$ (т.к. один запрос влияет на две вершины, значит, у нас не более $2k$ различных вершин).
2. Ни одно ребро никогда не будет смежно $V \setminus B$. Поэтому результат get-запроса на $\langle V, E\rangle$ равен результату get-запроса на $\langle B, E\rangle$ для любого множества ребер E.

Решением задачи будет рекурсивная процедура, которая на каждом шаге добавляет все ребра, которые точно нужно добавить, делит отрезок времени пополам, сжимает компоненты связности в вершины (т.е. создает новый граф), из нового графа выкидываются вершины, не используемые в ребрах-запросах. Эту рекурсивную процедуру изначально следует вызвать от множества всех запросов и отрезка $[0..k]$.

Алгоритм C:

```
1.  go( G, queries, L, R ) {
2.      // G = ⟨V, ∅⟩, нам важно только множество вершин V
3.      if L = R
4.          Если queries содержит get-запрос, обработать
5.          return
6.      A := ∅.
7.      queries_2 := ∅
8.      for q ∈ queries
9.          if (q.type = update) and (q.L < L) and (q.R ⩾ R)
10.             A.add(q)
11.         else if (q.type = get) or ((q.L < R) and (q.R ⩾ L))
12.             queries_2.add(q)
13.     сожмем компоненты графа ⟨V, A⟩, получим новый граф G_2
14.     G_3 = G_2 - вершины, не участвующие в queries_2
15.     M := [(L+R)/2]
16.     go(G_3, queries_2, L, M)
17.     go(G_3, queries_2, M+1, R)
18. }
19. main() {
20.     инициализация СНМ
21.     go(all_queries, 0, k)
22. }
```

Оцени время работы алгоритма C. Пусть длина отрезка $[L..R] = K$.

Теорема 6.7 : *Алгоритм C работает за время* $O(K \log K)$.

Доказательство: запишем реккурентное соотношение на время работы рекурсивной функции. $T(K) = O(|queries|) + T(K/2) + T(K/2)$ Как мы уже знаем из теоремы 6.1, $|queries| = O(K)$. Решив реккурентность, получаем время работы $O(K \log K)$.

7 Реберная двусвязность

7.1 Только добавление ребер: Offline решение за $O(\alpha)$ на запрос

Научимся сперва online обрабатывать запрос добавления ребра в граф. Будем поддерживать лес компонент реберной двусвязности. Каждую компоненту связности (дерево) будем хранить как подвешенное ориентированное дерево. Т.е. у каждой вершины `v` есть отец `parent[v]`, если вершина `v` — корень дерева, то пусть `parent[v]` = -1.

Обработка запроса состоит из двух случаев.

1. Если концы ребра принадлежат одной компоненте связности, то новое ребро образует ровно один цикл. При этом все вершины нового цикла лежат теперь в одной компоненте реберной двусвязности. Этот цикл можно сжать в новую вершину за время $O(Len \cdot \alpha)$, где Len — длина цикла. При сжатии цикла используется СНМ.

2. Если концы ребра принадлежат разным компонентам связности, то новых циклов не образуется, и нужно одну компоненту связности (дерево) подвесить к другой (к другому дереву). Чтобы это можно было просто сделать, достаточно чтобы один из двух концов ребра был корнем дерева. Обозначим это условие `[*]`. Действительно, если у нас есть ребро $\langle a, b \rangle$, и его конец a — корень своего дерева, то одной операцией `parent[b] := a`, мы соединим деревья в новое большое дерево.

Чтобы гарантировать условие `[*]`, заранее сделаем offline проход по всем ребрам, выполним все операции добавления в нужном порядке. В процессе добавления, некоторые ребра будут образовывать цикл. В этом случае мы считаем ребро не интересным и ничего не делаем. А некоторые ребра, будут соединять разные компоненты связности, такие ребра мы считаем интересными. Все интересные ребра образуют остовный лес F. Подвесим все деревья леса F за произвольные вершины. Утверждается, что теперь у нас выполнено условие `[*]`. Поскольку добавляемое ребро является ребром F, мы делаем отцом нижней вершины верхнюю вершину. После всех операций добавления мы получим ровно лес F.

7.2 Offline решение за $O(\log k)$

В этой части работы, для удобства рассуждения, все ребра имеют длину. Также все мосты имеют длину, поэтому запрос «сколько в графе мостов» переформулируется на запрос «суммарная длина всех мостов».

Соответственно у каждого запроса q помимо полей $q.a$, $q.b$, $q.L$, $q.R$ появилось поле $q.len$ — длина соответствующего ребра. Исходно все ребра имеют длину 1.

Сформулируем теорему аналогичную теореме [6.5].

Теорема 7.1 : *Сжатие компонент реберной двусвязности связности*
Если в графе $G = \langle V, E\rangle$ всегда присутствует набор ребер A (нет update-запросов, удаляющих ребра из A), то можно построить новый граф G_2, вершинами которого будут компоненты реберной двусвязности графа $\langle V, A\rangle$, а множество ребер $= \{\langle color[a], color[b]\rangle : \langle a, b\rangle \in E\}$. Здесь $color[a]$ — номер компоненты реберной двусвязности, в которой лежит вершина a. При переходе к новому графу запросы (и update-запросы, и get-запросы) следует поменять следующим образом: $a \to color[a]$ для всех вершин a, участвующих в запросе. **Утверждение теоремы:** результаты новых запросов типа `get` на графе G_2 будут совпадать с результатами старых запросов на графе G.

Доказательство:
Для доказательства я буду использовать следующий факт из теории графов, который оставлю без доказательства. За доказательством можно обратиться к [14].

Утверждение : *(Вершины a и b лежат в одной компоненте реберной двусвязности) $\Leftrightarrow$ (существуют два простых, не пересекающихся по ребрам, пути между a и b)*
Вернемся к доказательству теоремы.
1. Пусть в графе G есть два простых реберно не пересекающихся пути между a и b. Применим ко всем вершинам пути преобразование $a \to color[a]$ и уберем возможно появившиеся петли и циклы. Мы получили простые реберно не пересекающиеся пути в G_2.
2. Пусть в графе G_2 есть два простых реберно не пересекающихся пути между $color[a]$ и $color[b]$. Дополним их до путей в графе G.

Теперь сформулируем теорему аналогичную теореме [6.6].

Теорема 7.2 : *Редукция графа*
Если сейчас граф G является лесом, все деревья этого леса подвешены, а запросы затрагивают только множество вершин B, то можно все запросы рассматривать, как запросы к меньшему графу $G_2 = \langle B \cap \{LCA(b_i, b_j)\} : b_i, b_j \in B, E_2\rangle$. Здесь E_2 — ребра задающие лес на вершинах графа G_2 и имеющие длину равную расстоянию между соответствующими вершинами в графе G. **Утверждение теоремы:** Количество вершин в графе $G_2 = O(k)$, где k — общее количество запросов. Результаты get-

запросов над G и G_2 одинаковы.

Доказательство:
1. $|B| \leqslant 5k$. Нужно сперва увидеть, что у нас есть $2k$ концов ребер-запросов, затем сказать, что это листья некоторого леса. Вершины вида $LCA(b_i, b_j)$ — это развилки в этом дереве и особые вершины, корни деревьев. есть в каждом дереве есть еще одна особая вершина — корень дерева. В дереве с $2k$ листьями может быть не более $4k$ развилок, в каждом дереве не более одного корня. Каждый из k запросов породил не более одного корня. Итого, суммарное количество вершин не превосходит $5k$.
2. Рассмотрим два случая.
2.1. Граф G больше не будет меняться. Тогда достаточно заметить, что расстояния между вершинами при перехода от G к G_2 не поменялись.
2.2. В граф G добавятся новые ребра. Тогда нужно заметить, что любой кратчайший путь, проходящий по ребрам (и старым, и только что добавленным) графа G пройдет по любому ребру графа G_2 от начала до конца.
Доказательство закончено.

Решением задачи, как и в случае односвязности, будет рекурсивная процедура, которая на каждом шаге добавляет все ребра, которые точно нужно добавить, делит отрезок времени пополам, сжимает компоненты реберной двусвязности в вершины (т.е. создает второй граф), из второго графа создает третий с помощью преобразования, описанного в теореме [7.2]. Эту рекурсивную процедуру изначально следует вызвать от множества всех запросов и отрезка $[0..k]$.

Алгоритм D:

```
1.  go( G, queries, L, R ) {
2.      // G = ⟨V, ∅⟩, нам важно только множество вершин V
3.      if L = R
4.          Если queries содержит get-запрос, обработать
5.          return
6.      A := ∅.
7.      queries_2 := ∅
8.      for q ∈ queries
9.          if (q.type = update) and (q.L < L) and (q.R ⩾ R)
10.             A.add(q)
11.         else if (q.type = get) or ((q.L < R) and (q.R ⩾ L))
12.             queries_2.add(q)
```

13. сожмем компоненты реберной двусвязности графа $\langle V, A\rangle$, получим новый граф G_2
14. G_3 = `Reduction`(G_2)
15. M := $[\frac{L+R}{2}]$
16. **go**(G_3, $queries_2$, L, M)
17. **go**(G_3, $queries_2$, M+1, R)
18. }
19. **main**() {
20. инициализация CHM
21. **go**($all\_queries$, 0, k)
22. }

Оцени время работы алгоритма D.

Теорема 7.3 : *Алгоритм D работает за время* $O(k \log k)$.
Доказательство: для быстрого доказательства заметим, что алгоритм D отличается от алгоритма C только тем, что мы по-другому получаем граф G_3. Тем не менее, по-прежнему $|queries| = O(R-L+1)$, а $|G_3.V| = O(K)$. Т.е. доказать нужно только то, что граф G_3 мы построили из графа G за линейное время.
1. G_2 построен за линейное время, т.к. компоненты реберной двусвязности можно найти за линейное время [14], [1].
2. Преобразование `Reduction` также можно реализовать за линейное время, используя алгоритм Фараха-Колтона-Бендера подсчета LCA за $O(1)$. Упомянутый алгоритм описан здесь [15].

Теорема доказана.

8 Заключение

Данная работа содержит описание и обоснование корректности двух новых алгоритмов. Это простой алгоритм для решения задачи fully dynamic connectivity в режиме offline за время $O(k \log k)$ и аналогичный алгоритм для решения задачи fully dynamic 2-edge-connectivity в режиме offline за время $O(k \log k)$.
Второй по асимптотике времени работы превосходит все предыдущие достижения в данной области. Первый удобен и прост в реализации по сравнению с аналогичным по времени работы достижением Дэвида Эппштейна [3].

Список литературы

[1] Кормен, Лейзерсон, Ривест, Штейн. Алгоритмы. Построение и анализ. Издание 2-е, М.: Вильямс, 2007.

[2] Frederickson, Greg N., Data Structures for On-Line Updating of Minimum Spanning Trees, with Applications. CSTech-Report.No 83-449. May 1, 1984. Paper 368.

[3] Eppstein, D., Offline Algorithms for Dynamic Minimum Spanning Tree Problems. Tech-Report.No 92-04. January 10, 1992. Dept. of Information and Computer Science,Univ. of California, Irvine, CA 92717

[4] M. Thorup, Near-optimal fully-dynamic graph connectivity, in Proc. 32nd ACM Symposium on Theory of Computing (STOC), 2000, pp. 343-350.

[5] Holm, J., Lichtenberg, K., Thorup, M., Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity, Journal of the ACM, Vol. 48, No. 4, July 2001, pp. 723–760.

[6] Eppstein, D., Galily, Z., Italiano, G.F., Improved Sparsification, Tech. Report 93-20, April 15, 1993, Dept. of Information and Computer Science Univ. of California, Irvine, CA 92717

[7] Patrascu, M., Demaine E., Lower Bounds for Dynamic Connectivity, STOC'04, June 13–15, 2004, Chicago, Illinois, USA

[8] Alstrup, S., Ben-Amram, A.M., Rauhe, T., Worst-case and Amortised Optimality in Union-Find, 1999

[9] Tarjan, R.E., Efficiency of a good but not linear set union algorithm, Journal of the ACM, 22:215 225, 1975

[10] Mark de Berg, Mark van Kreveld. Computational Geometry Algorithms and Applications. Springer - Verlag Berlin Heidelberg New York, 1998

[11] Henzinger, M., King, V., Maintaining Minimum Spanning Fotests In Dynamic Graphs,2001 Society for Industrial and Applied Mathematics, SIAM J. COMPUT. Vol. 31, No. 2, pp. 364–374

[12] Henzinger, M., King, V., Fully Dynamic 2-Edge Connectivity Algorithm in Polylogarithmic Time per Operation, SRC Technical Note, 1997 - 004, June 27, 1997

[13] Henzinger, M., King, V., Randomized dynamic graph algorithm with polylogarithmic time per operation, 1995

[14] Асанов М.О., Баранский В.А., Расин В.В., Дискретная Математика: Графы, Матроиды, Алгоритмы, Ижевск: НИЦ "РХД 2001, 288 стр.

[15] M. A. Bender and M. Farach-Colton. "The LCA Problem Revisited."LATIN, pages 88-94, 2000