Санкт-Петербургский Национальный Исследовательский университет Информационных Технологий, Механики и Оптики

Факультет прикладной математики и информатики
Кафедра Компьютерных Технологий

Отчет по курсовой работе

по дисциплине «Структуры данных (углубленный курс)»

«Задача о Динамической Связности в Графе»

студентки I курса группы M4138
Ромашкиной Вероники Игоревны

Санкт-Петербург

1 Постановка задачи

Реализовать структуру данных динамической связности в произвольном неориентированном графе, которая поддерживает операции удаления рёбер, добавления и проверки того, что две вершины находятся в одной компоненте связности.

Код написан на языке Java. Исходные файлы можно найти по ссылке: github repository

Выполняется поддержка следующих операций:

- `boolean areConnected(String v, String u);` — запрос на проверку связности ребер, выполняется за $O(\log n)$
- `void link(String v, String u);` — добавление ребра в граф, выполняется за $O(\log n)$
- `void cut(String v, String u);` — удаление ребра из графа, выполняется за $O(\log^2 n)$ (амортизированная оценка)

Для более детального понимания алгоритма и выполненой работы перечислю используемые написанные классы и алгоритмы.

В пакете `ImplicitTreap` реализовано декартово дерево по неявному ключу (декартово дерево со случайными ключами, которые не хранятся в структуре). Данная структура поддерживает операции `merge(treap1, treap2)` — слияние двух деревьев, `split(treeNode)` — разрезание дерево по узлу (подняться к родителю, найти индекс и удалить элемент по индексу), `findIndex(treapNode)` — нахождение индекса, `add(treap, index)` — добавление, реализованное с помощью одного сплитов и двух операций мерджа и `remove(index)` — удаление с конкретного индекса, которое в свою очередь происходит путем двух разрезаний и одного слияния. В действительности, можно реализовать добавление при помощи только одной операции сплит, а удалении с помощью одного слияния, что незначитеьно бы ускорило выполнение данных операций, но я решила использовать более простой способ для поставленной задачи. `fullsize(treap)` — нахождение размерности всего дерева (подняться к корню и взять размер) также написан итератор, который позволяет лениво обходить дерево. Данный класс будет ключевым звеном в построении дерева эйлерова обхода. Замечу, что вершины хранят в себе параметрические данные, что будет использовано для хранения ребер в дальнейшем.

Для проверки корректности работы неявного декартового дерева имеется набор тестов `test.ru.ifmo.ads.romashkina.treap`: простых ручных, а также автоматических тесты для большого количества вершин и различных операций в дереве, используя класс утилит.

Пакет `graph` содержит в себе структуры :

`Vertex` — класс вершин. Каждыя вершина хранит информацию об исходящих ребрах

`Edge` — класс ориентированных ребер (пары названий вершин)

`TreapEdge` — класс ребер, содержащих в себе пару `Vertex` и также ссылку на ребро в эйлеровом дереве.

`Graph` — это `Map` вершин в ребра.

В пакете `Euler` содержится класс `EulerTree`, который содержит в себе алгоритм нахождения эйлерова пути графа, построения из него эйлерова дерева, основанного на `ImplicitTreap<TreapEdge>`, а также функции `link(u, v)`, `cut(u, v)`, `areConnected(u, v)`, которые поддерживают структуру дерева, сохраняя правильный эйлеров путь графа в узлах. Таким образом поддерживается инициализация структуры `DynamicConnectivity` из сразу заданного графа оптимальным способом: поиск остовного леса с последующим поиском Эйлерова обхода леса и построением деревьев эйлерова обхода.

Основной класс проекта — `DynamicConnectivity` сожержит в себе интерфейс с тремя основными операциями, который должен быть реализованы собственно классами `FastDynamicConnectivity` и наивной реализацией `NaiveDynamicConnectivity`. Операции в наивной реализации структуры, которая представляет из себя граф, устроены довольно просто. `link(v, u)` и `cut(v, u)` просто добавляют/удаляют соответствующее ребро из словаря <Вершина, Ребра> (они хранятся как `HashMap`, поэтому работают за $\mathcal{O}(1)$. А вот операция проверки связности основана на алгоритме поиска в глубину (Depth-first search, DFS), который реализован в данном классе.

Структура `FastDynamicConnectivity` устроена более интересным способом. Она представляет собой $\log n$ уровней `Level`, каждый из которых хранит пару графов — остовный лес и граф, содержащий в себе ребра данного уровня, не принадлежащие остовному лесу. Алгоритмы `link(v, u)` и `cut(v, u)` реализованы на основании статьи и видеолекции Маврина П.Ю..

Тестирование структуры данных происходит по следующему алгоритму. Генерируется граф из n вершин и в него добавляется разное число случайных ребер изначально из заданного множества вершин. После проверяемая структура инициализируется списком уровней Level. Остовный граф на основании построенного эйлерова дерева строится с помощью алгоритма Краскала с использованием структуры данных «система непересекающихся множеств» (DSU).

Тестирование корректности работы структуры проверяется сравнением списков ребер на конечном шаге у наивного и основного алгоритмов. Также после каждого вызова функции `areConnected` результат быстрого алгоритма сравнивается с наивным. Также проведен анализ работы алгоритмов по времени. Соответствующие результаты представлены в таблицах и графиках ниже (в миллисекундах).

n	$5n$	$\frac{n(n-1)}{128}$	$\frac{n(n-1)}{64}$
10	104	213	117
100	74	110	64
1000	84	70	75
10000	29	80	50 ∞

n	$5n$	$\frac{n(n-1)}{128}$	$\frac{n(n-1)}{64}$
10	57	104	57
100	479	541	530
1000	2262	2734	3370
10000	24184	164237	$+\infty$

Рис. 1: FastDynamicConnectivity vs NaiveDynamicConnectivity

На графиках видно, что Naive реализация работает сильно медленней Fast, хотя в таблицах при маленьких значениях числа вершин наивная работает быстрее, что обусловлено маленькой константой наивной реализации. Быстрая реализация не строго возрастает, потому что бенчмарки строят рандомизированные графы и тесты, и может получиться, что некоторые операции срабатывают быстрее из-за сильной разреженности графов на уровнях. К тому же, из-за сложного алгоритма трудно предсказать поведение и написать тест, который бы проверял структуру данных максимально честно.

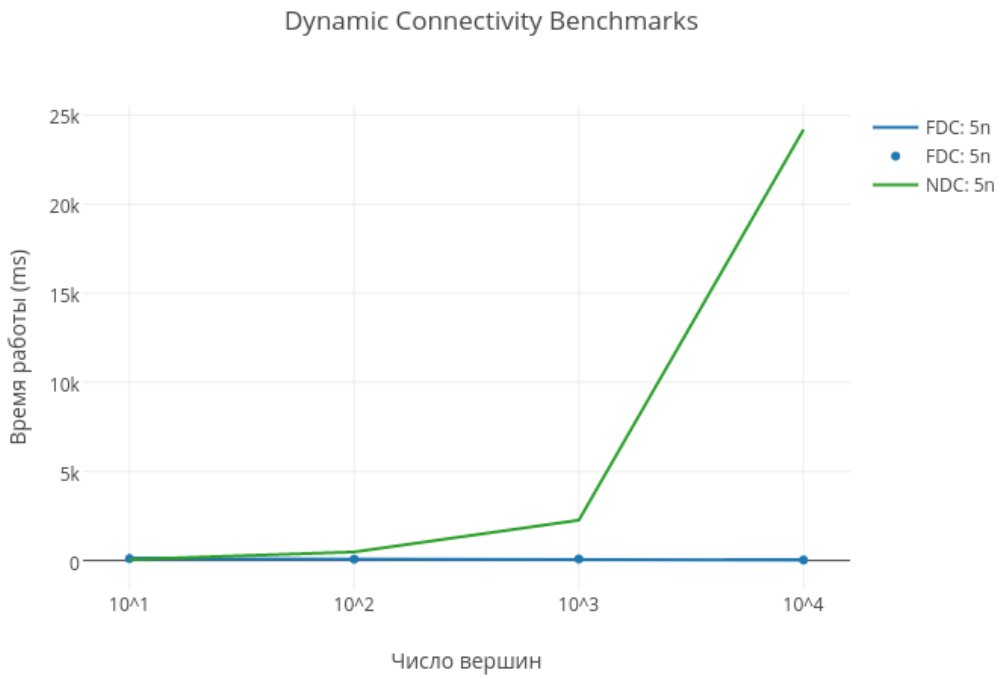

Рис. 2: График сравнения времени работы

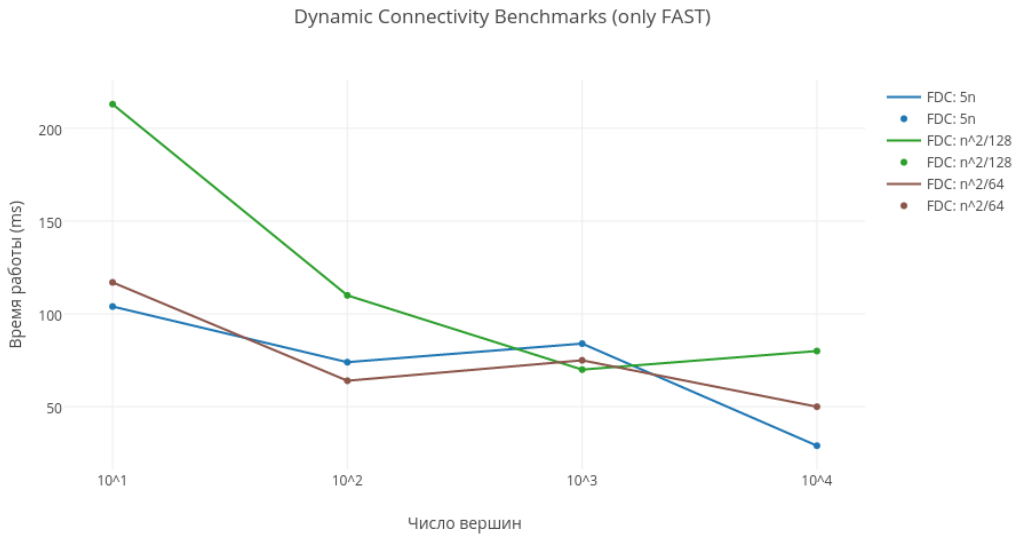

Рис. 3: График сравнения времени работы

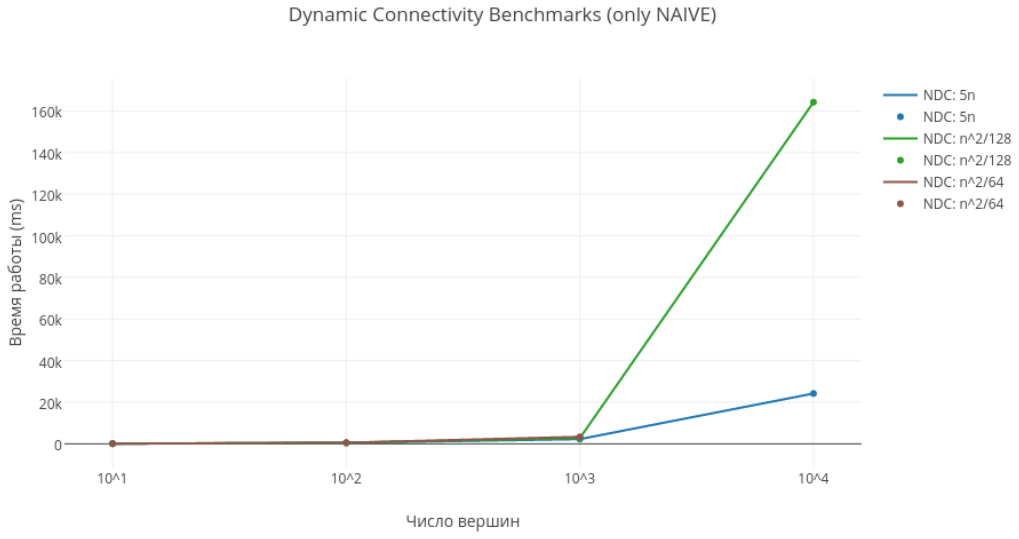

Рис. 4: График сравнения времени работы