О временной сложности алгоритма решения LA

Daniil Oreshnikov

16 September 2020

1 Введение

Задача LA (level ancestor) — поиска вершины в подвешенном дереве, находящейся выше заданной на определенное число, широко используется в различных сферах программирования и является важной частью теории графов в информатике. Для решения задачи LA существует множество алгоритмов, которые чаще всего являются улучшениями друг друга некоторыми неординарными способами.

Мы будем отдельно указывать время работы предподсчета и ответа на запрос, чтобы можно было сравнивать, как эти два значения изменялись с каждой следующей оптимизацией. Всего оптимизаций будет выделено пять:

1. двоичные подъемы
2. декомпозиция длинного пути
3. лестничная декомпозиция
4. объединение последних двух
5. macro-micro-tree

2 Алгоритм

2.1 Наивный алгоритм

Запустим обход в глубину и посчитаем для каждой вершины `parent`, чтобы `parent[v]` возвращал родителя вершины `v`

При запросе `la(a, x)` будем подниматься из `a` в `parent[a]` ровно `x` раз. Итоговое время работы — $\mathcal{O}(n)$ на предподсчет, $\mathcal{O}(n)$ на ответ на запрос.

2.2 Двоичные подъемы

Помимо подсчитанных родителей в прошлом случае, посчитаем также двоичные подъемы `up[i][v]` — вершину, которая на 2^i выше `v` или корень, если 2^i больше ее глубины. Это достаточно несложно сделать: `up[0] = parent`, а `up[i][v] = up[i - 1][up[i - 1][v]` (так как можно подняться на 2^{i-1} дважды), что можно посчитать в цикле по `i`.

Используя двоичные подъемы, можно при запросе `la(a, x)` сделать следующее:

```
for (h = (int_log2(n) + 1) downto 0) {
    if ((x >> h) & 1 == 1) {
        a = up[h][a]
    }
}
// return a
```

Заметим, что если разложить `x` в двоичной системе, то этот цикл просто собирает это число по единичкам, идя от большей степени к меньшей. Итоговое время работы — $\mathcal{O}(n \log n)$ на предподсчет массива `up`, $\mathcal{O}(\log n)$ на ответ на запрос.

2.3 Декомпозиция длинных путей

Несложно из `dfs` возвращать длину самого длинного пути из текущей вершины до какого-либо листа, а так же ребенка вершины на этом пути. Сделаем декомпозицию дерева на эти самые пути: возьмем самый длинный путь из корня и выделим его, а для оставшихся его детей повторим эту операцию:

```
fun build(v, path):
    path.append(v)
    for (u in children[v]) {
        if (longestPath[u] == longestPath[v] - 1) {
            build(u, path)
        } else {
            val newPath = Path()
            build(u, newPath)
        }
    }

build(root)
```

Теперь для каждой вершины достаточно запомнить `pathNo[v]` и `pathIndex[v]` — номер ее компоненты и ее индекс в ней (номер пути задается автоинкрементом в вызове `Path()`, индекс можно запомнить как `path.size()` в момент добавления).

Рассмотрим следующую операцию: подняться в корень (самую высокую вершину) текущего пути и еще на 1 выше. Давайте заметим, что так мы точно попадем в вершину в другом пути, а эта операция выполняется за $\mathcal{O}(1)$, так как мы знаем расстояние от вершины до корня пути (`pathIndex[v]`) и знаем всех родителей. Давайте повторять эту операцию, пока `x` > `pathIndex[v]`, а как только это перестанет выполняться, это будет означать, что искомая вершина находится на текущем пути, и надо вернуть `paths[pathNo[v]][pathIndex[v] - x]`.

Можно доказать, что операций перехода между разными путями будет не более $\mathcal{O}(\sqrt{n})$. Для этого достаточно заметить, что когда мы поднимаемся в другой путь, тот факт, что это самый длинный путь из этой вершины, означает, что он хотя бы той же длины, что и путь, из которого мы только что пришли (без учета новой текущей вершины, а значит на деле строго больше хотя бы на 1). Таким образом, количество оставленных «снизу» вершин растет не медленнее, чем $1 + 2 + 3 + \ldots$, то есть квадратично от числа шагов. Итоговое время работы — $\mathcal{O}(n)$ на предподсчет, $\mathcal{O}(\sqrt{n})$ на запрос.

2.4 Лестничная декомпозиция

Идея является прямой продолжением прошлой — оставим декомпозицию на пути, но «продлим» каждый из них в два раза вверх. При этом `pathNo` и `pathIndex` останутся без изменений, но в каждый `path` добавим последовательность вершин от него вверх до корня той же длины, что и он сам.

- во-первых, предподсчет все еще работает $\mathcal{O}(n)$, так как суммарная длина всех путей ровно n, а обходить вершины, поднимаясь вверх, можно с помощью `parent`
- во-вторых, теперь будем подниматься не в верх пути, а в верхнюю известную ему вершину. Заметим, что тогда эта вершина точно принадлежит другому пути, а значит, что он длиной хотя бы в два раза больше предыдущего, соответственно, из него можно подняться на хотя бы в два раза большую длину

Таким образом, высота вершины (считая снизу) растет экспоненциально, и LA решается за $\mathcal{O}(n)$ на предподсчет и $\mathcal{O}(\log n)$ на ответ на запрос.

2.5 Объединение идей

Посчитаем и лестничную декомпозицию, и двоичные подъемы. Найдем `la(a, x)` за два шага — прыгнем с помощью двоичных подъемов на максимальную степень двойки, не большую `x`, а затем сделаем один подъем в лестницах. Поскольку лестницы позволяют нам подняться не менее, чем на саммый длинный путь вниз, то мы сможем подняться на $2\times$ первый прыжок, то есть больше, чем на `x`.

Таким образом, мы получаем предподсчет за $\mathcal{O}(n \log n)$ и ответ на запрос за $\mathcal{O}(1)$.

2.6 Алгоритм Macro-Micro-Tree

Заметим, что можно посчитать двоичные подъемы не во всех вершинах, и соптимизировать предподсчет. А точнее, сделаем (примерно) следующее:

- посчитаем двоичные подъемы только в вершинах относительно вверху дерева
- предподсчитаем все поддеревья маленького размера
- дадим ответ на задачу в 3 шага: если мы достаточно снизу, используем предподсчитанные на небольших деревьях данные и прыгнем вверх, а затем оттуда применим двоичный подъем и лестничный подъем, как в предыдущей версии алгоритма

Более формально — рассмотрим вершины, размер поддерева которых меньше $\frac{\log n}{4}$ и удалим их из графа. Для всех листьев нового дерева посчитаем двоичные подъемы, сделав `dfs` и запоминая путь от корня до текущей вершины в стеке. Заметим, что по принципу Дирихле таких вершин не более $\frac{4n}{\log n}$, значит подсчет двоичных подъемов работает за $\mathcal{O}(n)$.

Заметим, что при этом количество различных деревьев размера $\frac{\log n}{4}$ (которые по сути были удалены) не превышает $\mathcal{O}(\sqrt{n})$, поскольку дерево можно задать тем, где в его обходе `dfs` спуски, а где подъемы, а это дает меньше $2^{\frac{\log n}{2}}$, то есть $\sqrt{n}$ вариантов. Поэтому можно для каждой возможной конфигурации дерева предподсчитать высоту в таком дереве от каждой вершины до ее корня за $\mathcal{O}(\sqrt{n} \cdot \frac{\log n}{4})$.

Итого — если высота поддерева заданной вершины меньше $\frac{\log n}{4}$, то поднимаемся в корень ее «маленького» дерева, еще на 1 вверх, делаем двоичный подъем и подъем в лестницу. Если же мы изначально оказались выше, давайте заранее еще предподсчитаем для каждой вершины ее любой лист, спустимся в него и из него вышеописанными шагами поднимемся на нужную высоту.

Суммарное время работы — $\mathcal{O}(n)$ на предподсчет и $\mathcal{O}(1)$ на ответ на запрос.