

# 08-hw-PA

vladrus13rus

November 2021

## 1 MIS



Рисунок 1. Мем \_без\_ названия.png

Итак, вспомним какойнибудь обычный алгоритм, решающий эту задачу. Конечно же, это жадность. Мы берем каждую вершину, и, если соседей нет в сете нет, то мы добавляем эту вершину в сет. Теперь мы хотим что то веселое. У нас есть какие то этапы reserve, commit и мы должны делать это не последовательно, а какими то последовательными кучами.

Где то в глубине сознания мы понимаем, что наш алгоритм не должен далеко отходить от последовательного. Только в чем разница? Конечно же, у нас будет точно входящие в сет вершины, вершины, которые точно в этот сет не попадут, вершины, которые хотят попасть в сет на данном этапе, и вершины, которые вообще никогда не были рассмотрены. Назовем их DONE, FAILED, PROGRESS, NONE соответственно. Изначально все вершины помечены как NONE. В конце они будут помечены как DONE или FAILED.

Как только мы рассматриваем вершину (reserve), то мы пробегаемся по всем ребрам этой вершины, концами которых являются вершины меньших номеров. Если мы там нашли вершину, которая хочет добавляться (PROGRESS), то мы тоже хотим попробовать добавиться. Если там есть уже добавленная (DONE) вершина, то мы немедленно говорим, что мы не можем участовать в игре (FAILED). В любом ином случае мы решаем, что мы все таки можем добавить себя (DONE). Конечно же, мы себя еще не добавили.

При этапе коммита (commit) мы утвердим свой статус в случаях, если мы точно добавлены (то есть рядом нет вершин (PROGRESS)) или мы уже точно не будем в сете (рядом есть вершина (DONE)).

Итого:

```

enum class Status {DONE, FAILED, PROGRESS, NONE};

class MIS {
    public Status status = Status.NONE;
    private Node node;

    boolean reserve(int i) {
        // представим, что у нас есть функция fold, которая
        // последовательно применяет к объекту функцию из
        // объекта и элемента из списка
        // Пример: (1, 2, 3).fold(X, (x, y) -> x + y) = ((X + 1) + 2) + 3
        (is_some_in, is_some_live) = node.childes // берем всех детей
            .filter(it -> it.number < node.number) // отсеиваем всех детей
                                            // чей номер больше
            .fold(pair(false, false), (current, it) ->
                pair(current.first | it.status == Status.IN,
                    current.second | it.status == Status.LIVE,
                )
            );
        // То есть теперь - у нас is_some_in отвечает за то, что есть
        // ли сосед с IN, is_some_live отвечает за то, есть ли у нас
        // сосед с LIVE
        if (is_some_in) {
            status = Status.FAILED;
            return 1;
        }
        if (is_some_live) {
            status = Status.LIVE;
            return 1;
        }
        statis = Status.IN;
        return 1;
    }

    boolean commit(int i) {
        \\ по хорошему, надо давать bonk illegal state при flag = NONE
        return (flag == IN | flag == OUT);
    }
}

```



Рисунок 2. Когда понял, что делает fold в решении

## 2 MST

Нам сразу говорят - ребра отсортированы, что *очень тонко* намекает нам на то, что нужно использовать алгоритм Крускала. В этом алгоритме нам последовательно предлагаю брать ребра и проверять, можно ли пройти по нему, и если да, к нему есть доступ, то мы забираем его.

Возьмем классическую реализацию spanning tree с лекции, ведь она несомненно поможет нам! Просто применим ее к нашему массиву, он же и так отсортирован по весам. В чем же подвох? Нам что, дали задачу без подвоха? (evil hw be like)

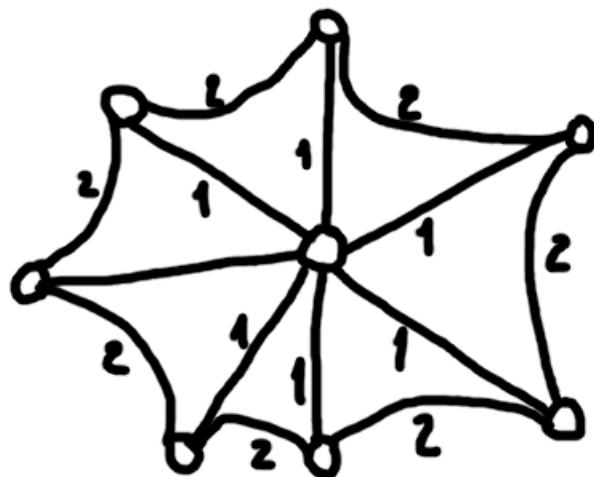


Рисунок 3. Время пауков! Здесь все хорошо

Конечно же нет, давайте искать его! Конечно же, посмотрим, что может нарушиться. Это последовательность добавления ребер. Мы можем выбрать, какие ребра добавляем, и

добавить их так, что мы могли выбрать на этом шаге оптимальнее. Почему мы могли так сделать? Потому что допустим у нас были ребра  $i, j, k$ , выгодным у нас были  $i, j$ , а мы выбрали  $i, k$ , потому что  $j$  почему то был недоступен (например, уже участвует где то), и мы закоммитили  $i, k$ . А вот и пример!

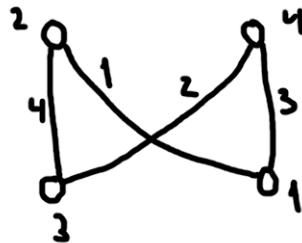


Рисунок 4. Стартовая часть

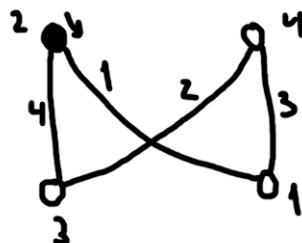


Рисунок 4. Берем 2-1, оно весит 1

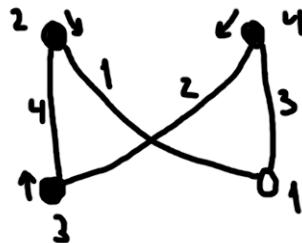


Рисунок 4. Берем 4-3, оно весит 2, но не берем 1-4, так как 4 занята. Берем 3-2, и нещадно проигрываем

Как же это исправить? Нельзя позволять вершине брать новые вершины, если она уже и так присоединена куда либо. Что же это значит? Что мы должны просто зарезервировать не только  $v$ , но и  $u$ ! В нашем примере это будет значить, что мы закоммитим изменения после первых двух ребер и сделаем все правильно. Почему это приведет к верному решению? Допустим, что у нас должно быть ребро  $a-b$ , но произошел коммит  $c-d$ . Это значит, что мы пропустили ребро  $a-b$ , из-за того, что минимум из них был уже забран. Но если  $c-d$  доступно только в этом случае, значит, что  $a-b$  мешает  $c-d$ , то есть, между этими двумя ребрами есть только одно ребро. Его мы и заблокировали