1. Алгоритм поиска в глубину

1.1. Неформальное описание

Поиск в глубину (Depth-first search, DFS) — один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти "вглубь" графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин. Таким образом, если красить каждую просмотренную компоненту в разные цвета, можно разбивать граф на компоненты связности.

1.2. Формальное описание

Пусть $G = (V, E)$. Предположим, что в начальный момент времени все вершины графа окрашены в *белый цвет*. Тогда выполним следующие действия.

```
void dfs(Vertex* v) {
    v->color = grey;
    for (Vertex* w : adjacent[v]) { // edge (v, w) ∈ E
        if (w->color == white)
            dfs(w);
    }
    v->color = black;
}

int main() {
    ...
    for (Vertex* v : graph) { // ∀v ∈ V
        if (v->color == white)
            dfs(v);
    }
}
```

Приведённый алгоритм часто пишут с двухцветной меткой, не используя промежуточную покраску в серый цвет. Тем, кто понял, как работает этот алгоритм, предлагается написать его нерекурсивную версию.

1.3. Поиск в глубину с метками времени. Классификация рёбер

1.3.1. Время входа и время выхода

Для каждой из вершин $u \in V$ установим два числа – время входа `entry[u]` и время выхода `leave[u]`. Модифицируем нашу процедуру

```
size_t Time = 0; //current time
void dfs(Vertex* v) {
    ++Time;
    entry[v] = Time;
```

```
        v->color = grey;
        for (Vertex* w : adjacent[v]) { // edge (v, w) ∈ E
            if (w->color == white)
                dfs(w);
        }
        v->color = black;
        ++Time;
        leave[v] = Time;
    }
```

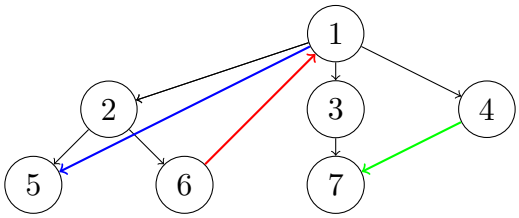
Считаем, что граф ориентированный. Очевидно, для любой вершины u, из которой мы не вышли в момент t, $entry[u] \leqslant t < leave[u]$. Теперь мы хотим классифицировать все рёбра (u, v) нашего графа (Тут есть два способа построения определений – на основе цветов или на основе только что введённых величин. Однако эти определения являются эквивалентными.).

1.3.2. Классификация рёбер

Рассмотрим подграф предшествования обхода в глубину $G_p = (V, E_p)$, где $E_p = \{(p_u, u) | u \in V\}$, где в свою очередь p_u — вершина, от которой был вызван dfs(u). Подграф предшествования поиска в глубину образует лес обхода в глубину, который состоит из нескольких деревьев обхода в глубину. С помощью полученного леса можно классифицировать ребра графа G:

1. Ребрами дерева назовем те ребра из G, которые вошли в G_p.
2. Ребра (u, v), соединяющие вершину u с её предком v в дереве обхода в глубину назовем обратными ребрами (для неориентированного графа предок не должен быть родителем, так как иначе ребро будет являться ребром дерева).
3. Ребра (u, v), не являющиеся ребрами дерева и соединяющие вершину u с её потомком v в дереве обхода в глубину назовем прямыми ребрами (в неориентированном графе нет разницы между прямыми и обратными ребрами, поэтому все такие ребра считаются обратными).
4. Все остальные ребра назовем перекрестными ребрами – такие ребра могут соединять вершины одного и того же дерева обхода в глубину, когда ни одна из вершин не является предком другой, или соединять вершины в разных деревьях.

Придумайте как классифицировать рёбра прямо во время обхода графа в глубину (используйте цвета).



На рисунке выше красным обозначены обратные рёбра, зелёным – перекрёстные, синим – прямые, а чёрным – рёбра дерева обхода в глубину.

1.4. Время работы DFS

Оценим время работы обхода в глубину. Процедура dfs вызывается от каждой вершины не более одного раза, а внутри процедуры рассматриваются все инцидентные ей рёбра. Но таких рёбер суммарно $\sum\limits_{v \in V} \deg(v) = 2E$, то есть время работы можно ограничить величиной $O(V + E)$.

2. Нахождение цикла в ориентированном графе

2.1. Формальная постановка задачи и алгоритм решения

Формально задача поиска цикла в орграфе записывается так. Пусть $G = (V, E)$ – ориентированный граф. Требуется найти и вывести цикл в G, если он есть, или сказать, что его нет. Эту задачу решает DFS. Произведём серию обходов. То есть из каждой вершины, в которую мы ещё ни разу не приходили, запустим поиск в глубину, который при входе в вершину будет красить её в серый цвет, а при выходе из нее – в чёрный. И, если алгоритм пытается пойти в серую вершину, то это означает, что цикл найден. Для восстановления самого цикла достаточно при запуске поиска в глубину из очередной вершины добавлять эту вершину в стек. Когда поиск в глубину нашел вершину, которая лежит на цикле, будем последовательно вынимать вершины из стека, пока не встретим найденную еще раз. Все вынутые вершины будут лежать на искомом цикле.

2.2. Доказательство корректности алгоритма

Рассмотрим выполнение процедуры поиска в глубину от некоторой вершины v. Так как все серые вершины лежат в стеке рекурсии, то для них вершина v достижима, так как между соседними вершинами в стеке есть ребро. Тогда, если из рассматриваемой вершины v существует ребро в серую вершину u, то это значит, что из вершины u существует путь в v и из вершины v существует путь в u состоящий из одного ребра. И так как оба эти пути не пересекаются, то цикл существует.

Докажем теперь, что если в графе G существует цикл, то $\mathrm{dfs}(G)$ его всегда найдет. Пусть v – первая вершина принадлежащая циклу, рассмотренная поиском в глубину. Тогда существует вершина u, принадлежащая циклу и имеющая ребро в вершину v. Так как из вершины v в вершину u существует белый путь (они лежат на одном цикле), т.е. путь по белым вершинам, то во время выполнения процедуры поиска в глубину от вершины u, вершина v будет серой. Так как из u есть ребро в v, то это ребро в серую вершину. Следовательно $\mathrm{dfs}(G)$ нашел цикл.

3. Топологическая сортировка.

3.1. Определение

Топологическая сортировка (Topological Sort) ориентированного ациклического графа $G = (V, E)$ представляет собой упорядочивание вершин таким образом, что для любого ребра $(u, v) \in E$ номер вершины u меньше номера вершины v.

3.2. Применение

Топологическая сортировка применяется в самых разных ситуациях, например при создании параллельных алгоритмов, когда по некоторому описанию алгоритма нужно составить граф зависимостей его операций и, отсортировав его топологически, определить, какие из операций являются независимыми и могут выполняться параллельно (одновременно). Примером использования топологической сортировки может служить создание карты сайта, где имеет место древовидная система разделов. Также топологическая сортировка применяется при обработке исходного кода программы в некоторых компиляторах и IDE, где строится граф зависимостей между сущностями, после чего они инициализируются в нужном порядке, либо выдается ошибка о циклической зависимости.

3.3. Теорема о существовании

Теорема. *Для любого ациклического ориентированного графа существует топологическая сортировка. То есть*

$$\exists \varphi \colon V \to \{1 \dots n\}, (u, v) \in E \Rightarrow \varphi(u) < \varphi(v)$$

Доказательство. Определим $leave[u]$ как порядковый номер окраски вершины u в черный цвет в результате работы алгоритма dfs. Рассмотрим функцию $\varphi = n + 1 - leave[u]$. Очевидно, что такая функция подходит под критерий функции φ из условия теоремы, если выполняется следующее утверждение:

$$\forall (u, v) \in E \Rightarrow leave[u] > leave[v]$$

Это утверждение легко доказывается от обратного (здесь мы будем пользоваться тем, что граф ациклический). Поэтому мы оставим доказательство. Функция φ существует. □

3.4. Алгоритм

Доказательство предыдущей теоремы порождает мгновенно алгоритм топологической сортировки

```
vector<size_t> ans;
vector<size_t> topological_sort(graph G) {
    vector<bool> visited(G->number, false);
    if (cyclic_graph(G))
        return ans;
    for (Vertex* v : G)
        if (!v->visited)
            dfs(v);
    return ans;
}
void dfs(Vertex* v) {
    v->visited = true;
    for (Vertex* u : adjacent[v]) {
            if (!u->visited)
            dfs(u);
    }
    ans.push_back(u);
}
```