Применения DFS

Denis Bakin

План лекции

1. Компоненты связности
2. Поиск цикла (раскраска в 3 цвета)
3. Двудольные графы и раскраска в 2 цвета
4. Топологическая сортировка

Часть 1: Компоненты связности

Компонента связности: определение

Компонента связности: определение

Компонента связности — максимальное по включению множество вершин, в котором из любой вершины можно добраться до любой другой по рёбрам графа

Компонента связности: определение

Компонента связности — максимальное по включению множество вершин, в котором из любой вершины можно добраться до любой другой по рёбрам графа

Свойства:

- Каждая вершина принадлежит ровно одной компоненте

Компонента связности: определение

Компонента связности — максимальное по включению множество вершин, в котором из любой вершины можно добраться до любой другой по рёбрам графа

Свойства:

- Каждая вершина принадлежит ровно одной компоненте
- Компоненты не пересекаются

Компонента связности: определение

Компонента связности — максимальное по включению множество вершин, в котором из любой вершины можно добраться до любой другой по рёбрам графа

Свойства:

- Каждая вершина принадлежит ровно одной компоненте
- Компоненты не пересекаются
- Граф связен $\iff$ он имеет ровно одну компоненту

Задача: найти все компоненты

Вход: неориентированный граф $G = (V, E)$

Выход: для каждой вершины — номер её компоненты связности

Задача: найти все компоненты

Вход: неориентированный граф $G = (V, E)$

Выход: для каждой вершины — номер её компоненты связности

Ключевое наблюдение: DFS посетит все вершины в компоненте связности начальной вершины.

Задача: найти все компоненты

Вход: неориентированный граф $G = (V, E)$

Выход: для каждой вершины — номер её компоненты связности

Ключевое наблюдение: DFS посетит все вершины в компоненте связности начальной вершины.

Идея алгоритма:

1. Перебираем все вершины от 0 до $n - 1$
2. Если вершина ещё не посещена — запускаем из неё DFS
3. Все вершины, достигнутые этим DFS, образуют одну компоненту
4. Увеличиваем счётчик компонент

Компоненты связности: реализация

```
std::vector<std::vector<int>> adj(n);
std::vector<int> component(n, -1);  // номер компоненты для каждой вершины
int num_components = 0;

void dfs(int v, int comp) {
    component[v] = comp;
    for (int u : adj[v]) {
        if (component[u] == -1) {
            dfs(u, comp);
        }
    }
}
```

Компоненты связности: реализация

```cpp
std::vector<std::vector<int>> adj(n);
std::vector<int> component(n, -1);  // номер компоненты для каждой вершины
int num_components = 0;

void find_components() {
    for (int v = 0; v < n; ++v) {
        if (component[v] == -1) {
            dfs(v, num_components);
            ++num_components;
        }
    }
}
```

Пример: компоненты связности

Компоненты связности: сложность

Компоненты связности: сложность

Временная сложность: $O(n + m)$

- Каждая вершина посещается ровно один раз
- Каждое ребро просматривается ровно два раза

Пространственная сложность: $O(n)$

- Массив `component` размера n
- Глубина рекурсии до $O(n)$

Задача: есть ли цикл в графе?

Вход: ориентированный граф $G = (V, E)$

Выход:

- Есть ли в графе цикл?
- Если есть — вывести вершины цикла

Раскраска вершин: белый, серый, чёрный

Введём три состояния (цвета) для каждой вершины:

Цвет	Значение	Константа
Белый	вершина не посещена	`WHITE = 0`
Серый	вершина в процессе обработки (в стеке рекурсии)	`GRAY = 1`
Чёрный	вершина полностью обработана	`BLACK = 2`

Ключевое наблюдение: цикл существует $\iff$ есть ребро в серую вершину

Почему серая вершина означает цикл?

Серая вершина = вершина в текущем стеке рекурсии

Если из u есть ребро в серую вершину v:

- Существует путь $v \to \ldots \to u$ (по дереву DFS)
- Существует ребро $u \to v$
- Значит, есть цикл $v \to \ldots \to u \to v$

Пример: граф с циклом

Поиск цикла: реализация

```cpp
enum Color { WHITE = 0, GRAY = 1, BLACK = 2 };

std::vector<std::vector<int>> adj(n);
std::vector<Color> color(n, WHITE);
std::vector<int> parent(n, -1);
int cycle_start = -1, cycle_end = -1;
```

Поиск цикла: реализация

```
bool dfs(int v) {
    color[v] = GRAY;
    for (int u : adj[v]) {
        if (color[u] == GRAY) {
            // Нашли цикл: ребро v -> u, где u серая
            cycle_start = u;
            cycle_end = v;
            return true;
        }
        if (color[u] == WHITE) {
            parent[u] = v;
            if (dfs(u)) return true;
        }
    }
    color[v] = BLACK;
    return false;
}
```

Восстановление цикла

```cpp
std::vector<int> get_cycle() {
    std::vector<int> cycle;

    // Идём от cycle_end к cycle_start по parent
    int v = cycle_end;
    while (v != cycle_start) {
        cycle.push_back(v);
        v = parent[v];
    }
    cycle.push_back(cycle_start);

    // Разворачиваем, чтобы получить порядок обхода
    std::reverse(cycle.begin(), cycle.end());

    return cycle;
}
```

Запуск поиска цикла

```
bool has_cycle() {
    for (int v = 0; v < n; ++v) {
        if (color[v] == WHITE) {
            if (dfs(v)) {
                return true;
            }
        }
    }
    return false;
}
```

Запускаем DFS из всех непосещённых вершин — граф может быть несвязным

Поиск цикла: сложность

Поиск цикла: сложность

Временная сложность: $O(n+m)$

- Каждая вершина меняет цвет: белый $\rightarrow$ серый $\rightarrow$ чёрный
- Каждое ребро просматривается один раз

Пространственная сложность: $O(n)$

- Массивы `color`, `parent`
- Глубина рекурсии

Раскраска графа: определение

Раскраска графа в k цветов — это функция $c : V \to \{1, 2, \dots, k\}$, такая что для любого ребра $(u, v) \in E$ выполняется $c(u) \neq c(v)$

Раскраска графа: определение

Раскраска графа в k цветов — это функция $c : V \to \{1, 2, \dots, k\}$, такая что для любого ребра $(u, v) \in E$ выполняется $c(u) \neq c(v)$

Иначе говоря: смежные вершины должны иметь разные цвета

Хроматическое число $\chi(G)$ — минимальное k, для которого существует раскраска в k цветов

Раскраска графа: примеры

Граф	$\chi(G)$	Пояснение
Пустой граф (без рёбер)	1	Все вершины одного цвета
Дерево	2	Чередуем цвета по уровням
Цикл чётной длины	2	Чередуем цвета
Цикл нечётной длины	3	Два цвета недостаточно
Полный граф K_n	n	Все вершины попарно смежны

Раскраска графа: сложность задачи

Задача: дан граф G и число k. Можно ли раскрасить G в k цветов?

Раскраска графа: сложность задачи

Задача: дан граф G и число k. Можно ли раскрасить G в k цветов?

Сложность:

- $k = 1$: тривиально (граф без рёбер)
- $k = 2$: решается за $O(n + m)$ — **сегодня разберём**
- $k \geq 3$: **NP-полная задача!**

Даже для $k = 3$ не известен полиномиальный алгоритм

Двудольный граф: определение

Двудольный граф (bipartite graph) — граф, вершины которого можно разбить на два множества L и R так, что все рёбра идут между L и R (внутри множеств рёбер нет)

Двудольный граф: определение

Двудольный граф (bipartite graph) — граф, вершины которого можно разбить на два множества L и R так, что все рёбра идут между L и R (внутри множеств рёбер нет)

Эквивалентные определения:

- Граф можно раскрасить в 2 цвета
- $\chi(G) \leq 2$
- Граф не содержит циклов нечётной длины

Проверка двудольности: идея

Алгоритм:

1. Запускаем DFS/BFS из произвольной вершины
2. Красим стартовую вершину в цвет 0
3. Всех соседей красим в противоположный цвет (1)
4. Соседей соседей — снова в цвет 0
5. Если встретили уже покрашенную вершину того же цвета — граф не двудольный

Пример: двудольный граф

Двудольность: сложность

Двудольность: сложность

Временная сложность: $O(n + m)$

- Стандартный DFS/BFS

Пространственная сложность: $O(n)$

- Массив цветов

Топологическая сортировка: определение

Топологическая сортировка ориентированного графа — это линейный порядок вершин $v_1, v_2, \dots, v_n$ такой, что для каждого ребра $(v_i, v_j) \in E$ выполняется $i < j$

Топологическая сортировка: определение

Топологическая сортировка ориентированного графа — это линейный порядок вершин $v_1, v_2, \dots, v_n$ такой, что для каждого ребра $(v_i, v_j) \in E$ выполняется $i < j$

- если есть ребро из u в v, то u стоит раньше v в порядке
- вершины можно пронумеровать таким образом, что ребра будут идти только из вершин с меньшим номером в вершины с большим

Топологическая сортировка: когда существует?

Теорема: Топологическая сортировка существует $\iff$ граф ациклический

Топологическая сортировка: когда существует?

Теорема: Топологическая сортировка существует $\iff$ граф ациклический

Доказательство ($\Rightarrow$):

- Пусть есть цикл $v_1 \to v_2 \to ... \to v_k \to v_1$
- Тогда $v_1 < v_2 < ... < v_k < v_1$ — противоречие

Доказательство ($\Leftarrow$): конструктивно — алгоритм

Применения топологической сортировки

- **Компиляция:** порядок сборки модулей с зависимостями
- **Расписание:** порядок выполнения задач с prerequisite
- **Курсы в университете:** какие предметы брать сначала
- **Makefile / build systems:** порядок сборки
- **Spreadsheet:** порядок вычисления ячеек с формулами

Топологическая сортировка через DFS

Ключевое наблюдение:

При DFS вершина получает $tout$ только после обработки всех достижимых из неё вершин

Топологическая сортировка через DFS

Ключевое наблюдение:

При DFS вершина получает $tout$ только после обработки всех достижимых из неё вершин

Алгоритм:

1. Запустить DFS из всех вершин
2. Записывать вершины в список при выходе (когда присваиваем $tout$)
3. Развернуть список — это и есть топологический порядок

Почему это работает?

Пусть есть ребро $u \to v$. Возможны случаи:

Ситуация	$tout[u]$ vs $tout[v]$
v белая при входе в u	$tout[u] > tout[v]$ OK
v чёрная при входе в u	$tout[u] > tout[v]$ OK
v серая при входе в u	цикл! (граф не DAG)

Во всех корректных случаях $tout[u] > tout[v]$

После разворота: u раньше v ⊠

Топологическая сортировка: реализация

```
std::vector<std::vector<int>> adj(n);
std::vector<bool> visited(n, false);
std::vector<int> order;  // результат

void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u]) {
            dfs(u);
        }
    }
    order.push_back(v);  // добавляем при выходе
}
```

Пример: топологическая сортировка

Топологическая сортировка: сложность

Топологическая сортировка: сложность

Временная сложность: $O(n+m)$

- Стандартный DFS + разворот массива $O(n)$

Пространственная сложность: $O(n)$

- Массивы `visited`/`color`, `order`
- Глубина рекурсии

Итоги: компоненты связности

- Запускаем DFS из каждой непосещённой вершины
- Каждый запуск — новая компонента
- Сложность: $O(n+m)$

Итоги: поиск цикла

- Раскраска: белый $\to$ серый $\to$ чёрный
- Цикл $\iff$ ребро в серую вершину (back edge)
- Восстановление через массив `parent`
- Сложность: $O(n+m)$

Итоги: двудольность

- Двудольный $\iff$ раскрашивается в 2 цвета $\iff$ нет нечётных циклов
- Алгоритм: DFS с чередованием цветов
- Конфликт = вершина того же цвета
- Сложность: $O(n+m)$

Итоги: топологическая сортировка

- Линейный порядок: если $u \to v$, то u раньше v
- Существует только для DAG (ациклических графов)
- DFS: обратный порядок $tout$
- Кан: удаление вершин с нулевой входящей степенью
- Сложность: $O(n + m)$