Модули на АСМ

Владимир Милосердов, Владимир Шабанов

23 октября 2014 г.

Оглавление

1 Геометрия **5**

2 Графы **7**

2.1 Основы . . . 7

2.2 Идентификация графа . . . 8

2.2.1 Поиск цикла и проверка на ацикличность . . . 8

2.2.2 Гамильтонов граф . . . 9

2.2.3 Двудольный граф O(n) . . . 9

2.2.4 Компоненты связности O(n) . . . 9

2.2.5 Компоненты сильной связности O(n + m) . . . 10

3 Строки **13**

4 ДП **15**

5 Алгебра **17**

5.1 Алгоритм Евклида . . . 17

5.1.1 НОД, НОК (gcd, lcm) . . . 17

5.1.2 Расширенный алгоритм Евклида . . . 17

5.1.3 Восстановление в кольце по модулю . . . 17

5.2 Решето Эратосфена . . . 18

5.2.1 Классический вариант . . . 18

5.2.2 Линейное решето . . . 18

5.3 Разбор выражений . . . 18

6 Разное **21**

ОГЛАВЛЕНИЕ

Глава 1

Геометрия

Глава 2

Графы

2.1 Основы

Граф - множество вершин и ребер(заданных явно или не явно). Понятия используемые в дальнейшем:

- Ациклический граф

  Граф без циклов

- Влентность/Степень вершины

  Количество ребер входящих/выходящих в вершину

- Взвешенный граф

  Граф в котором у каждого ребра есть стоимость

- Висячая вершина

  Вершина со степенью один

- Гамильтонов путь

  Путь в графе содержащий каждую вершину ровно один раз

- Гамильтонов цикл

  Цикл содержащий каждую вершину ровно один раз

- Двудольный граф

  Граф в котором можно выделить два множества такие, что между любыми двумя вершинами одного множества нет ребер.

- Компонента связности

Множество вершин и ребер графа такое, что из каждой его вершины достижима любая другая вершина этого множества

- Компонента сильной связности

  Множество вершин и ребер ориентированного графа такое, что из каждой его вершины достижима любая другая вершина этого множества

- Кратные ребра

  Ребра связывающие одну и ту же пару вершин

- Минимальный каркас

  Множество ребер соеденяющих все вершины графа без циклов и имеющее минимальный суммарный вес

- Паросочетания

  Множество попарно не смежных ребер

- Точка сочленения

  Вершина после удаления которой количество компонент связности возрастает

- Эйлеров путь

  Путь в графе содержащий каждое ребро ровно один раз

- Эйлеров цикл

  Цикл содержащий каждое ребро ровно один раз

2.2 Идентификация графа

2.2.1 Поиск цикла и проверка на ацикличность

Воспользуемся поиском в глубину. Окрасим все вершины в белый. Запускаясь от вершины перекрашиваем ее в серый, а выходя из нее красим в черный. Если dfs попытается пойти в серую вершину, значит мы нашли цикл, который сможем вывести с помощью массива предков, иначе граф циклов не имеет. Далее реализация на списках смежности.

```
bool dfs (int v) {
    cl[v] = 1; //colors
    for (int i = 0; i < g[v].size(); i++) {
        int to = g[v][i];
        if (cl[to] == 0) {
            p[to] = v;
            if (dfs(to)) //if son have cycle then parent have cycle
                return true;
        }
```

```
        else if (cl[to] == 1) {
            cycle_end = v;
            cycle_st = to;
            return true;
        }
    }
    cl[v] = 2;
    return false;
}
```

2.2.2 Гамильтонов граф

Пусть n количесттво вершин графа, а δ минимальная степень вершины в графе, тогда граф имеет гамильтонов цикл если $n \geq 3$ и $\delta \geq \frac{n}{2}$

2.2.3 Двудольный граф O(n)

Так как граф является двудольным тогда и только тогда, когда все циклы четны, определить двудольность можно за один проход в глубину. На каждом шаге обхода в глубину помечаем вершину. Допустим мы пошли в первую вершину — помечаем её как 1. Затем просматриваем все смежные вершины и если не помечена вершина, то на ней ставим пометку 2 и рекурсивно переходим в нее. Если же она помечена и на ней стоит та же пометка, что и у той, из которой шли (в нашем случае 1), значит граф не двудольный.

```
bool dfs (int v, int c) {
    cl[v] = c; //colors
    for (int i = 0; i < g[v].size(); i++) {
        int to = g[v][i];
        if (cl[to] == 0) {
            return dfs(to, max(1, (c + 1) % 2));
        }
        else
            return cl[to] != c;
    }
}
```

2.2.4 Компоненты связности O(n)

Поиск компонент связности можно осуществить многими методами, в том числе и поиском в глубину. Окрасим все вершины в индивидуальные цвета. Запуская $[dfs]$ от каждой вершины будем перекрашивать в ее цвет все вершины в этой компоненте. В конце алгоритма у нас останется столько цветов, сколько компонент связности в графе, а цвет каждой вершины будет идентифицировать ее компоненту.

```
void dfs (int v, int c) {
    if (!used[v])
        return;
    cl[v] = c; //colors
    for (int i = 0; i < g[v].size(); i++) {
        int to = g[v][i];
        if (cl[to]!= cl[v]) {
            dfs(to, c);
            return;
```

```
        }
}
int main(){
    ...
    for (int i = 0; i < N; i++){
        dfs(i, i);
    }
}
```

2.2.5 Компоненты сильной связности O(n + m)

Решим эту задачу за несколько обходов в глубину. Сначала топологически (по времени выхода) отсортируем граф, затем обойдем транспонированый(инвертированый) граф в этом порядке. Найденые компоненты связности этого графа и будут компонентами сильной связности исходного графа.

```
vector < vector<int> > g, gr; // граф и транспортированый граф
vector<char> used;
vector<int> order, component; // порядок топ сорта и компонента сильной связности

void dfs1 (int v) {
    used[v] = true;
    for (int i = 0; i < g[v].size(); i++)
        if (!used[g[v][i]])
            dfs1(g[v][i]);
    order.push_back(v);
}

void dfs2 (int v) {
    used[v] = true;
    component.push_back(v);
    for (int i = 0; i < gr[v].size(); i++)
        if (!used[gr[v][i]])
            dfs2(gr[v][i]);
}

int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; i++){
        int a, b;
        cin >> a >> b;
        g[a].push_back(b);
        gr[b].push_back(a);
    }

    used.assign(n, false);
    for (int i = 0; i < n; i++)
        if (!used[i])
            dfs1(i);
    used.assign(n, false);
    for (int i = 0; i < n; i++) {
        int v = order[n - 1 - i];
        if (!used[v]) {
            dfs2 (v);
            //... вывод component ...
```

```
                component.clear();
            }
        }
}
```

Глава 3

Строки

Глава 4

ДП

Глава 5

Алгебра

5.1 Алгоритм Евклида

5.1.1 НОД, НОК (gcd, lcm)

$$gcd(a, b) = \begin{cases} a & \text{если } a = 0 \\ b & \text{иначе} \end{cases}$$
$$lcm(a, b) = \frac{a \cdot b}{gcd(a, b)}$$

5.1.2 Расширенный алгоритм Евклида

```
int gcdex (int a, int b, int & x, int & y) {
    if (a == 0) {
        x = 0; y = 1;
        return b;
    }
    int x1, y1;
    int d = gcd (b%a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
```

Функция возвращает нод и коеффициенты x, y по сслыкам

5.1.3 Восстановление в кольце по модулю

```
int x, y;
int g = gcdex (a, m, x, y);
if (g != 1)
    cout << "no␣solution";
else {
    x = (x % m + m) % m;
    cout << x;
}
```

5.2 Решето Эратосфена

5.2.1 Классический вариант

Дано число n. Требуется найти все простые в отрезке $[2; n]$. Решето Эратосфена решает эту задачу за $O(n \log \log n)$
Запишем ряд чисел $1 \dots n$, и будем вычеркивать сначала все числа, делящиеся на 2, кроме самого числа 2, затем деляющиеся на 3, кроме самого числа 3, затем на 5 и так далее ...

```
int n;
vector<char> prime (n+1, true);
prime[0] = prime[1] = false;
for (int i=2; i<=n; ++i)
    if (prime[i])
        if (i * 1ll * i <= n)
            for (int j=i*i; j<=n; j+=i)
                prime[j] = false;
```

5.2.2 Линейное решето

Чуть быстрее по времени, чем классическое $O(N)$. Цена вопроса - оверхед по памяти. Пусть $lp[i]$ – минимальный простой делитель числа $i, 2 \leq i \leq n$.

- $lp[i] = 0$ – число i – простое
- $lp[i] \neq 0$ – число i – составное

```
const int N = 10000000;
int lp[N+1];
vector<int> pr;

for (int i=2; i<=N; ++i) {
    if (lp[i] == 0) {
        lp[i] = i;
        pr.push_back (i);
    }
    for (int j=0; j<(int)pr.size() && pr[j]<=lp[i] && i*pr[j]<=N; ++j)
        lp[i * pr[j]] = pr[j];
}
```

Вектор $lp[\ ]$ можно как-то использовать для факторизации чисел

5.3 Разбор выражений

Дано выражение. Начинаем парсить его функцией E1, которая обрабатывает самые низкоприоритетные операции (в нашем случае '+', '-').

```
def E1():
    res = E2()
    while True:
        c = getc()
        if c == '+':
            res += E2()
```

```
        elif c == ’-’:
            res -= E2()
        else:
            putc(c)
            return res
```

Сначала происходит обработка атома, стоящего слева от знака (’+’, ’-’), затем обработка каждого атома между знаками. Обработку этих атомов производит функция $E2()$. Это функция более низкого ранга, которая обрабатывает более приоритетные операции (в нашем случае ’*’ и ’/’).

```
def E2():
    res = E3()
    while True:
        c = getc()
        if c == ’*’:
            res *= E3()
        elif c == ’/’:
            res /= E3()
        else:
            putc(c)
            return res
```

Функция работает аналогично $E1()$. Таким образом мы можем поддерживать сколько угодно операций различных приоритетов, добавляя функции.

Перейдем к обработке скобок:

```
def E3():
    if c == ’(’:
        res = E1()
        c = getc()
        return res
    else:
        putc(c)
        return E4()
```

Берём символ, если он скобка, тогда обрабатываем выражение внутри и считываем закрывающую скобку.

Теперь рассмотрим считывание числа. Ничего сложного:

```
def E4():
    res = 0
    while True:
        c = getc()
        if str.isdigit(c):
            res = res * 10 + int(c)
        else:
            putc(c)
            return res
```

Глава 6

Разное