Модули на ACM

Владимир Милосердов, Владимир Шабанов

7 ноября 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
2.2.6 Минимальное остовное дерево O($M * log(M)$) 11
2.2.7 Паросочетания 11
2.2.8 Точки сочленения 13
2.2.9 Эйлеров путь/цикл 13

3 Строки **15**

4 ДП **17**

5 Структуры данных **19**

6 Алгебра **21**
6.1 Алгоритм Евклида 21
6.1.1 НОД, НОК (gcd, lcm) 21
6.1.2 Расширенный алгоритм Евклида 21
6.1.3 Восстановление в кольце по модулю 21
6.2 Решето Эратосфена 22
6.2.1 Классический вариант 22
6.2.2 Линейное решето 22
6.3 Разбор выражений 22

7 Разное **25**

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.

```
1 bool dfs (int v) {
2     cl[v] = 1; //colors
3     for (int i = 0; i < g[v].size(); i++) {
4         int to = g[v][i];
5         if (cl[to] == 0) {
6             p[to] = v;
7             if (dfs(to)) //if son have cycle then parent have cycle
8                 return true;
9         }
```

```
10          else if (cl[to] == 1) {
11              cycle_end = v;
12              cycle_st = to;
13              return true;
14          }
15      }
16      cl[v] = 2;
17      return false;
18  }
```

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.

```
1  bool dfs (int v, int c) {
2      cl[v] = c; //colors
3      for (int i = 0; i < g[v].size(); i++) {
4          int to = g[v][i];
5          if (cl[to] == 0) {
6              return dfs(to, max(1, (c + 1) % 2));
7          }
8          else
9              return cl[to] != c;
10     }
11 }
```

2.2.4 Компоненты связности O(n)

Поиск компонент свзязности 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.

```
1  void dfs (int v, int c) {
2      if (!used[v])
3          return;
4      cl[v] = c; //colors
5      for (int i = 0; i < g[v].size(); i++) {
6          int to = g[v][i];
7          if (cl[to]!= cl[v]) {
8              dfs(to, c);
9              return;
```

```
10          }
11  }
12  int main(){
13      ...
14      for (int i = 0; i < N; i++){
15          dfs(i, i);
16      }
17  }
```

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.

```
1   vector < vector<int> > g, gr; // граф и транспортированый граф
2   vector<char> used;
3   vector<int> order, component; // порядок топ сорта и компонента сильной связности
4
5   void dfs1 (int v) {
6       used[v] = true;
7       for (int i = 0; i < g[v].size(); i++)
8           if (!used[g[v][i]])
9               dfs1(g[v][i]);
10      order.push_back(v);
11  }
12
13  void dfs2 (int v) {
14      used[v] = true;
15      component.push_back(v);
16      for (int i = 0; i < gr[v].size(); i++)
17          if (!used[gr[v][i]])
18              dfs2(gr[v][i]);
19  }
20
21  int main() {
22      int n, m;
23      cin >> n >> m;
24      for (int i = 0; i < m; i++){
25          int a, b;
26          cin >> a >> b;
27          g[a].push_back(b);
28          gr[b].push_back(a);
29      }
30
31      used.assign(n, false);
32      for (int i = 0; i < n; i++)
33          if (!used[i])
34              dfs1(i);
35      used.assign(n, false);
36      for (int i = 0; i < n; i++) {
37          int v = order[n - 1 - i];
38          if (!used[v]) {
39              dfs2 (v);
40              //... вывод component ...
```

```
41                 component.clear();
42             }
43         }
44 }
```

2.2.6 Минимальное остовное дерево O($M * log(M)$)

Алгоритм Кр(а|у)скала. Составим список ребер, отсортируем их по возрастанию весов. Распределим все вершины в соответствующие им множества, для работы с ними воспользуемся СНМ. Будем добавлять ребра по порядку, но только, если они соединяют вершины из разных множеств. После этого объединим их множества.

```
1  vector<int> p (n);//Множества
2
3  int dsu_get(int v) {
4      return(v == p[v]) ? v : (p[v] = dsu_get(p[v]));
5  }
6
7  void dsu_unite(int a, int b) {
8      a = dsu_get(a);
9      b = dsu_get(b);
10     if (rand() & 1)
11         swap(a, b);
12     if (a != b)
13         p[a] = b;
14 }
15
16 int main(){
17                 .....
18 sort (g.begin(), g.end()); // список ребер <pair<вес, pair<начало, конец> > >
19 p.resize (n);
20 for (int i = 0; i < n; i++)
21     p[i] = i;
22 for (int i = 0; i < m; i++) {
23     int a = g[i].second.first,  b = g[i].second.second,  l = g[i].first;
24     if (dsu_get(a) != dsu_get(b)) {
25         cost += l; // вес каркаса
26         res.push_back(g[i].second); // каркас
27         dsu_unite(a, b);
28     }
29 }
```

2.2.7 Паросочетания

Наибольшее паросочетание O($N * M$)

Введем новое понятие понятия:

- Увеличивающая цепь

  Простой путь, ребра которого поочередно входят/невходят в паросочетание, а первая и последняя вершина не пренадлежат паросочетанию

Теорема *Просочетание является максимальным тогда и только тогда, когда не существует увеличивающих его цепей*

В основе алгоритма лежит эта теорема, позволяющая "улучшать"любое паросочетание. Будем считать, что граф уже разбит на две доли. Для каждой вершины из первой доли делаем следующее: если у текущей вершины уже есть ребро включенное в паросочетание, то переходим к следующей вершине, иначе запускаем поиск увеличивающей цепи из этой вершины. Поиск увеличивающей цепи.

Для каждого ребра из этой вершины делаем следующее: пусть смежная с текущим ребром вершина не принадлежит паросочетанию, значит увеличивающая цепь найдена, добавим это ребро к паросочетанию, иначе ищем увеличивающую цепь из новой вершины. Максимальное паросочетание будет найдено после проверки всех вершин из первой доли.

Т.к. на каждом шаге алгоритм улучшает текущее паросочетание, то асимптотику можно значительно улучшить, если за начальное паросочетание брать не пустое, а любое другое, полученное любым эвристическим методом.

Следует заметить, что номера вершин разных долей в данной реализации независимы.

```
1  int n, k; // число вершин в первой и второй доли соответственно
2  vector < vector<int> > g; // список ребер из вершин первой доли
3  vector<int> mt; // mt[i] номер вершины первой доли связанной ребром с вершиной второй
       доли под номером i
4  vector<char> used;
5
6  bool try_kuhn (int v) {
7      if (used[v])  return false;
8      used[v] = true;
9      for (int i = 0; i < g[v].size(); i++) {
10         int to = g[v][i];
11         if (mt[to] == -1 || try_kuhn (mt[to])) {
12             mt[to] = v;
13             return true;
14         }
15     }
16     return false;
17 }
18 int main() {
19     //... чтение графа ...
20
21     mt.assign (k, -1);//текущее паросочетание пусто
22     vector<char> used1 (n);
23     for (int i = 0; i < n; i++)
24         for (int j = 0; j < g[i].size(); j++)
25             if (mt[g[i][j]] == -1) {
26                 mt[g[i][j]] = i; // строим какое-нибудь паросочетание
27                 used1[i] = true;
28                 break;
29             }
30     for (int i = 0; i < n; i++) {
31         if (used1[i])  continue;
32         used.assign (n, false);
33         try_kuhn (i);
34     }
35
36     for (int i = 0; i < k; i++)
37         if (mt[i] != -1)
38             printf ("%d␣%d\n", mt[i]+1, i+1);
39 }
```

2.2.8 Точки сочленения

Алгоритм для связного графа. Запустим поиск в глубину от произвольной вершины. Основная идея алгоритма: вершина является точка сочлинения, тогда и только тогда, когда невозможно попасть в предков этой вершины из её сыновей. Исключение составляет лишь вершина старта dfs, ведь у нее нет предков. Она будет точкой сочленения, если у нее больше одного сына, выевленного поиском в глубину

```
1  vector<int> g[MAXN];
2  bool used[MAXN];
3  int timer, tin[MAXN], fup[MAXN];
4
5  void dfs (int v, int p = -1) {
6      used[v] = true;
7      tin[v] = fup[v] = timer++;
8      int children = 0;
9      for (int i = 0; i < g[v].size(); i++) {
10         int to = g[v][i];
11         if (to == p)  continue;
12         if (used[to])
13             fup[v] = min (fup[v], tin[to]);
14         else {
15             dfs (to, v);
16             fup[v] = min (fup[v], fup[to]);
17             if (fup[to] >= tin[v] && p != -1)
18                 IS_CUTPOINT(v);
19             children++;
20         }
21     }
22     if (p == -1 && children > 1)
23         IS_CUTPOINT(v);
24 }
25
26 int main() {
27     int n;
28
29     timer = 0;
30     for (int i = 0; i < n; i++)
31         used[i] = false;
32     dfs (0);
33 }
```

2.2.9 Эйлеров путь/цикл

Необходимым условием Эйлерово циклаи пути является наличие не более одной компоненты связности с ненулевым числом ребер. Если выполняется это условие и степень каждой вершины четна, то в графе существует Эйлеров цикл, а если степень всех вершин, кроме двух четна, то существует Эйлеров путь.Алгоритм напоминает поиск в глубину. Главное отличие состоит в том, что пройденными помечаются не вершины, а ребра графа. Начиная со стартовой вершины v строим путь, добавляя на каждом шаге не пройденное еще ребро, смежное с текущей вершиной. Вершины пути накапливаются в стеке S. Когда наступает такой момент, что для текущей вершины w все инцидентные ей ребра уже пройдены, записываем вершины из S в ответ, пока не встретим вершину,

которой инцидентны не пройденные еще ребра. Далее продолжаем обход по не посещенным ребрам. Для поиска Эйлерова пути необходимо начинать с вершины у которой степень нечетна, и добавить ребро между вершинами с нечетными степенями, а из найденного цикла удалить добавленное ребро. Псевдокод:

```
1  findPath(v):
2     S.clear()
3     S.add(v)
4     while not S.isEmpty():
5       w := S.top()
6       if E contains(w, u):
7         S.add(u)
8         remove(w, u)
9       else:
10        S.pop()
11        print w
```

Глава 3

Строки

Глава 4

ДП

Глава 5

Структуры данных

Глава 6

Алгебра

6.1 Алгоритм Евклида

6.1.1 НОД, НОК (gcd, lcm)

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

6.1.2 Расширенный алгоритм Евклида

```
1  int gcdex (int a, int b, int & x, int & y) {
2      if (a == 0) {
3          x = 0; y = 1;
4          return b;
5      }
6      int x1, y1;
7      int d = gcd (b%a, a, x1, y1);
8      x = y1 - (b / a) * x1;
9      y = x1;
10     return d;
11 }
```

Функция возвращает нод и коеффициенты x, y по сслыкам

6.1.3 Восстановление в кольце по модулю

```
1 int x, y;
2 int g = gcdex (a, m, x, y);
3 if (g != 1)
4     cout << "no␣solution";
5 else {
6     x = (x % m + m) % m;
7     cout << x;
8 }
```

6.2 Решето Эратосфена

6.2.1 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;
```

6.2.2 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

Глава 7

Разное

7.1 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()
```

```
5           return res
6       else:
7           putc(c)
8           return E4()
```

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

Теперь рассмотрим считывание числа. Ничего сложного:

```
1  def E4():
2      res = 0
3      while True:
4          c = getc()
5          if str.isdigit(c):
6              res = res * 10 + int(c)
7          else:
8              putc(c)
9              return res
```