

编程学习笔记

Awsdkl

2025 年 2 月 6 日

目录

I	字符串	3
1	字符串匹配	4
1.1	暴力做法	4
1.2	KMP 算法	4
1.2.1	前缀函数	4
1.2.2	代码	4
1.3	查找	5
1.3.1	代码	5
II	数学	6
2	数论	7
2.1	最大公约数	7
2.1.1	定义	7
2.1.2	欧几里得算法	7
2.2	裴蜀定理	8
2.2.1	内容	8
2.3	乘法逆元	8
2.3.1	定义	8
2.3.2	扩展欧几里得算法	8
III	数据结构	10
3	并查集	11
3.1	并查集	11
3.1.1	实现	11
3.2	带权并查集	11
4	线段树	12
4.1	Kinetic Tournament 树 (KTT)	12
4.1.1	解决的问题	12
4.1.2	过程	12
4.1.3	实现	13

目录	2
IV 图论	16
5 树	18
5.1 相关定义与前置知识	18
5.2 树链剖分	19
5.2.1 解决的问题	19
5.2.2 重链剖分	19
6 连通性相关	22
6.1 相关定义与前置知识	22
6.2 割点	23
6.2.1 过程	23
6.2.2 实现	24
6.3 割边	25
6.3.1 过程	25
6.3.2 实现	25
6.4 点双连通分量	26
6.4.1 实现	26
6.5 边双连通分量	26
6.5.1 实现	26
6.6 强连通分量	26
6.6.1 实现	26

Part I

字符串

Chapter 1

字符串匹配

字符串匹配又被称为模式匹配 (pattern matching)，简单来说就是在主串 s 中寻找子串 t 。而字符串 t 被成为 **模式串** (pattern)。

1.1 暴力做法

这种做法非常简单。思想主要就是从主串 s 的第一个字符开始和模式串 t 的第一个字符开始匹配，若相等，则比较二者后续的字符。若不匹配，则模式串 t 退回到第一个字符，与主串 s 的下一个字符比较。如此循环往复，直至 s 与 t 都比较完毕。

不难发现这种算法的时间复杂度在最坏情况下为 $O(|s| \cdot |t|)$ ，是非常大的。

1.2 KMP 算法

1.2.1 前缀函数

定义

给定一个长度为 m 的字符串 t ，则其前缀数组 $next_i$ 表示： t 的第 1 至 i 位中前缀和后缀相同的部分的长度。

1.2.2 代码

```
1 char s[MAXN], t[MAXN];
2 int n, m;
3 int nxt[MAXN];
4
5 void init()
6 {
7     for(int i = 2, j = 0; i <= m; i++)
8     {
9         while(j && t[i] != t[j+1]) j = nxt[j];
10        if(t[i] == t[j+1]) j++;
11        nxt[i] = j;
12    }
```

13 }

1.3 查找

1.3.1 代码

```
1 void Match()
2 {
3     for(int i = 1, j = 0; i <= n; i++)
4     {
5         while(j && s[i] != t[j+1]) j = nxt[j];
6         if(s[i] == t[j+1]) j++;
7         if(j == m)
8         {
9             //找到了
10        }
11    }
12 }
```

Part II

数学

Chapter 2

数论

2.1 最大公约数

2.1.1 定义

最大公约数 (Greatest Common Divisor), 常缩写为 gcd。一组整数的公约数, 即同时是这组数中每一个数的约数的数。 ± 1 是任意一组整数的公约数。一组整数的最大公约数, 是指所有公约数中最大的一个。我们对于不全为 0 的两个整数 a, b , 将其最大公约数记作 $\gcd(a, b)$ 。对于不全为 0 的 n 个整数 $a_1, a_2, a_3 \dots a_n$, 将其最大公约数记作 $\gcd(a_1, a_2, a_3 \dots a_n)$ 。

那么如何计算最大公约数呢? 我们先考虑两个数的情况。

2.1.2 欧几里得算法

过程

已知我们有两个数 a, b , 我们要求出其 $\gcd(a, b)$ 。

注意到: $\gcd(a, b) = \gcd(b, a \bmod b)$ 。

并且当 b 为 0 是, 两数的最大公约数就是 a , 且两数大小均减小, 我们可以写出以下式子:

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases} \quad (2.1)$$

证明

设: $a = b \times k + c$, 显然, $c = a \bmod b$ 。

设: $d|a, d|b$, 即 d 为 a, b 的公约数。

则 $c = a - b \times k$, 两边同时除以 d 后, 得: $\frac{c}{d} = \frac{a}{d} - \frac{b}{d} \times k$ 。

显然, $\frac{c}{d}$ 也是一个整数。所以 d 也是 b, c 的公约数

可得: 对于 a 和 b 的公约数, 它也会是 b 和 $a \bmod b$ 的公约数。

反过来也需要证明:

设: $d|b, d|(a \bmod b)$ 。

我们仍可以得到与之前类似的式子: $\frac{a \bmod b}{d} = \frac{a}{d} - \frac{b}{d} \times k$, 推出 $\frac{a \bmod b}{d} + \frac{b}{d} \times k = \frac{a}{d}$ 。
显然左边式子为整数, 则 $\frac{a}{d}$ 为整数。

可得: b 和 $a \bmod b$ 的公约数也是 a 和 b 的公约数。

既然两式公约数相同, 那么最大公约数也会相同。

则可得到式子 $\gcd(a, b) = \gcd(b, a \bmod b)$ 。

实现

```
1 int gcd(int a, int b)
2 {
3     return !b ? a : gcd(b, a % b);
4 }
```

2.2 裴蜀定理

2.2.1 内容

设 a, b 是不全为零的整数, 对任意整数 x, y , 满足 $\gcd(a, b) \mid ax + by$, 且存在整数 x, y , 使得 $ax + by = \gcd(a, b)$ 。

2.3 乘法逆元

2.3.1 定义

如果一个线性同余方程 $ax \equiv 1 \pmod{b}$, 则 x 称为 $a \bmod b$ 的逆元, 记作 a^{-1} 。

猜你不知道逆元有什么用。我们知道乘法是可以直接取模的, 但是在涉及到除法的时候取模会发生错误。应此就有了逆元。在模 b 的意义下除以 a , 就等于在模 b 的意义下将原数乘上 $a \bmod b$ 的逆元。逆元就相当于在模意义下的倒数, 所以才会被记作 a^{-1} 。

2.3.2 扩展欧几里得算法

扩展欧几里得算法 (Extended Euclidean algorithm, EXGCD), 常用于求 $ax + by = \gcd(a, b)$ 的一组可行性解。也可以用于求解乘法逆元。

过程

对于式子 $ax \equiv 1 \pmod{b}$, 我们可以将其改写为: $ax + by = 1$, 显然在这里, 一定有 $\gcd(a, b) = 1$, 因此求出第二个式子的解, 其中的 x 就是 a 的乘法逆元。

设:

$$ax_1 + by_1 = \gcd(a, b)$$

$$bx_2 + (a \bmod b)y_2 = \gcd(b, a \bmod b)$$

在欧几里得算法中我们知道: $\gcd(a, b) = \gcd(b, a \bmod b)$

所以 $ax_1 + by_1 = bx_2 + (a \bmod b)y_2$

又因为 $a \bmod b = a - (\lfloor \frac{a}{b} \rfloor \times b)$

所以 $ax_1 + by_1 = bx_2 + (a - (\lfloor \frac{a}{b} \rfloor \times b))y_2$

推出 $ax_1 + by_1 = ay_2 + bx_2 - \lfloor \frac{a}{b} \rfloor \times b \times y_2 = ay_2 + b(x_2 - \lfloor \frac{a}{b} \rfloor \times y_2)$

因为 $a = a, b = b$, 所以 $x_1 = y_2, y_1 = x_2 - \lfloor \frac{a}{b} \rfloor \times y_2$

所以我们可以将 x_2, y_2 带入递归中, 直至求出 gcd, 然后再递归 $x = 1, y = 0$ 回去求解。

实现

```
1 typedef long long ll;
2 void exgcd(ll a, ll b, ll &x, ll &y)
3 {
4     if(b == 0)
5     {
6         x = 1, y = 0;
7         return;
8     }
9     exgcd(b, a % b, y, x);
10    y -= a / b * x;
11 }
```

Part III

数据结构

Chapter 3

并查集

3.1 并查集

3.1.1 实现

解释

fa : 表示第 i 个点的父亲。

代码

fa 数组记得初始化。

```
1  int fa[(int)1e4+5];
2
3  int Find(int x)
4  {
5      return x == fa[x]?x:fa[x] = Find(fa[x]);
6  }
7
8  int merge(int x,int y)
9  {
10     int a = Find(x);
11     int b = Find(y);
12     if(a != b)
13     {
14         fa[b] = a;
15     }
16 }
```

3.2 带权并查集

Chapter 4

线段树

4.1 Kinetic Tournament 树 (KTT)

4.1.1 解决的问题

先来看 Luogu5693。省流：区间加，查询最大子段和。

会发现它和 Luogu4513 的模板最大子段和是有差别的（废话，不然怎么会是紫题）。这道题多了区间加，是无法直接做的。因此就有毒瘤想到了分块。但是我们不会，于是就有大佬搞出了 KTT。

4.1.2 过程

该做法的原理在 EI 的博客 中有详细阐述。这里会用更加通俗易懂（至少我觉得）的语言来描述。

首先考虑经典不带区间修的最大子段和线段树做法，每个结点维护 lm, mm, rm, sum 分别表示前缀最大子段和，最大子段和，后缀子段和，区间总和。于是有：

$$lm = \max(lp.lm, lp.sum + rp.sum)$$

$$mm = \max(lp.mm, rp.mm, lp.rm + rp.lm)$$

$$rm = \max(rp.rm, rp.sum + lp.rm)$$

$$sum = lp.sum + rp.sum$$

再来看修改，如果这一次区间修改**没有**造成决策的改变（及上述式子中 \max 的取值没有改变），则该区间的答案会增大 kx 。（其中 k 是原先被选择子段的长度， x 是本次修改每个点加的值）

因此我们可以**将原来维护的每个值看成一次函数**，每次区间加操作后值符合函数 $y = kx + b$ ，其中 b 为该值原本的大小，而 y 就是修改后值的大小。

在这一情况下，当所加的 x 极小时， \max 的取值并不会发生变化。

我们假设现在在 \max 函数里左右两个一次函数分别为 y_1, y_2 ，根据其图像得（图像懒得画了）：当 x 较小时， \max 的值取其中一条直线；当 x 大于一定值时（这个值就是两图像交点对应的横坐标）， \max 的取值改为另一条直线。我们称这是一次“击败（defeat）”事件。在这种情况下我们向下暴力递归到该节点并重新选取当下 x 值最大的直线。

应此我们需要多记录一个当前节点发生击败事件所需最小的 x (通俗来讲, 就是当前节点及其子树的所有节点的众多 \max 中的一对函数两两相交的节点的横坐标的最小值)。当未发生击败事件时, 将这个 x 减小增加的值。当发生时, 向下递归, 如上述操作。

看似很暴力, 通过复杂的势能分析, EI 得到复杂度为 $O((n+m)\log^3 n + q\log n)$ 。接下来我们分析时间复杂度:

可以发现作者不会, 分析完毕。

4.1.3 实现

```

1 struct Func//一次函数
2 {
3     int k; ll b;
4     Func operator+(const Func &_)const
5     {
6         return Func{k + _.k, b + _.b};
7     }
8     void add(ll v) { b += k * v; }
9 };
10
11 /// @brief 对两个一次函数取  $x = 0$  时的最值, 同时给出  $\max$  的选取不发生变化的最大值。
12 /// @param a 第一个一次函数
13 /// @param b 第二个一次函数
14 /// @return 一个 pair。
15 /// @return 第一个是当  $x$  较小 (这里  $x = 0$ ) 时值更大的函数。
16 /// @return 第二个是当  $x$  大于这个值时, 会发生一次 “击败事件”, 及另一个函数的值更大。
17 pair<Func, ll> max(Func a, Func b)
18 {
19     if(a.k<b.k||a.k==b.k&& a.b<b.b) swap(a,b);
20     if(a.b>=b.b) return make_pair(a,INF);
21     return make_pair(b,(b.b-a.b)/(a.k-b.k));
22 }
23
24 struct Node //线段树的节点
25 {
26     Func lm, mm, rm, sum;
27     ll x;
28     ll tag;
29     Node operator+(const Node &_)const
30     {
31         Node re;re.tag = 0;
32         pair<Func, ll> tmp;
33         re.x = min(x, _.x);
34         tmp = max(lm, sum + _.lm);
35         re.lm = tmp.first; re.x = min(re.x, tmp.second);
36         tmp = max(_.rm, _.sum + rm);

```

```

37     re.rm = tmp.first; re.x = min(re.x, tmp.second);
38     tmp = max(mm, _.mm);
39     re.x = min(re.x, tmp.second);
40     tmp = max(tmp.first, rm + _.lm);
41     re.mm = tmp.first; re.x = min(re.x, tmp.second);
42     re.sum = sum + _.sum;
43     return re;
44 }
45 };
46
47 struct KTT
48 {
49     #define lp (p<<1)
50     #define rp (p<<1|1)
51     Node tree[MAXN<<2];
52     ll nums[MAXN];
53     int len;
54     //这个函数将节点 p 加上 v
55     void update(int p, ll v)
56     {
57         tree[p].tag += v;
58         tree[p].x -= v;
59         tree[p].lm.add(v);
60         tree[p].mm.add(v);
61         tree[p].rm.add(v);
62         tree[p].sum.add(v);
63     }
64     void push_up(int p)
65     {
66         tree[p] = tree[lp] + tree[rp];
67     }
68     void push_down(int p)
69     {
70         if(!tree[p].tag) return;
71         update(lp, tree[p].tag);
72         update(rp, tree[p].tag);
73         tree[p].tag = 0;
74     }
75     void build(int s, int t, int p)
76     {
77         if(s == t)
78         {
79             Func tmp = {1, nums[s]};
80             tree[p] = Node{tmp, tmp, tmp, tmp, INF, 0};
81             return;
82         }
83         int mid = (s + t) >> 1;

```

```

84     build(s, mid, lp);
85     build(mid+1, t, rp);
86     push_up(p);
87 }
88 //发生击败事件, 向下递归直到 v < 节点的 x
89 void defeat(int s, int t, int p, ll v)
90 {
91     if(v > tree[p].x)
92     {
93         int mid = (s + t) >> 1;
94         ll t = tree[p].tag + v;
95         tree[p].tag = 0;
96         defeat(s, mid, lp, t);
97         defeat(mid+1, t, rp, t);
98         push_up(p);
99     }
100     else update(p, v);
101 }
102 void add(int l, int r, int s, int t, int p, ll v)
103 {
104     if(l <= s && t <= r)
105     {
106         defeat(s, t, p, v);
107         return;
108     }
109     push_down(p);
110     int mid = (s + t) >> 1;
111     if(l <= mid) add(l, r, s, mid, lp, v);
112     if(r > mid) add(l, r, mid+1, t, rp, v);
113     push_up(p);
114 }
115 Node query(int l, int r, int s, int t, int p)
116 {
117     if(l <= s && t <= r)
118     {
119         return tree[p];
120     }
121     int mid = (s + t) >> 1;
122     push_down(p);
123     if(r <= mid) return query(l, r, s, mid, lp);
124     if(l > mid) return query(l, r, mid+1, t, rp);
125     return query(l, r, s, mid, lp) + query(l, r, mid+1, t, rp);
126 }
127 };

```


Part IV

图论

语文不好，结点节点乱用了。

一些常用概念：

度数，与一个顶点 v 关联的边的条数称作该顶点的 **度 (degree)**，记作 $d(v)$ 。特别的，对于边 (v, v) ，则每条边对 $d(v)$ 要产生 2 的贡献。

时间戳，表示**每个点被第一次访问的时间**（可以简单理解为是第几个被访问的点）。我们用 **dfn**（dfs number）数组记录每个点的时间戳，即 dfn_i 表示第一次访问结点 i 的时间。若结点 i 未被访问，则 $\text{dfn}_i = 0$ ，因此通常也可以用 **dfn** 数组判断一个结点是否被访问过。

Chapter 5

树

5.1 相关定义与前置知识

图论中的树和现实中的树很像。

树上有几种节点：

- 根结点 (root)：在有根树上指定的一个节点，无特殊要求。
- 叶结点 (leaf node)：对于无根树，即为度数不超过 1 的结点；对于有根树，即为没有子结点的结点。

一个没有固定根节点的树称为 **无根树 (unrooted tree)**。无根树也有形式化的定义：

- 有 n 个结点， $n - 1$ 条边的连通无向图。
- 无向无环连通图。
- 任意两个结点之间有且仅有一条简单路径的无向图。

在无向图的基础上，若指定一个点为根（若需要，任何点都可被指定为根），则形成一棵 **有根树 (rooted tree)**。有根树在很多时候和无根树一样，只是根规定了结点的上下关系。

- 森林 (forest)：多个树凑一起就是森林（是不是很形象？）。同时，一棵树也是森林。

对于有根树（对于具体题目中的无根树，通常会任意指定一个点作为根）：

- 父亲 (双亲结点 parent)：从该结点到根的路径上的第二个点。特别的，根结点没有父结点。
- 祖先 (ancestor)：从该点到根的路径上处该点以外的所有结点均为该点的祖先。
- 子结点 (孩子结点 child node)：若 u 为 v 的父亲，则 v 是 u 的子结点。子结点的顺序一般不区分，但有时二叉树会分为左子结点和右子结点。
- 深度 (depth)：该点到根结点的路径上的边数。
- 高度 (height)：所有结点的深度最大值。

不难发现，每个轻子节点都是链顶，且每条链都是一个轻子节点后面跟着一大坨的重子节点。

实现

重链剖分的实现分为了两个 dfs 的过程。

对于第一个 dfs，需要记录每个点的父节点，深度，子树大小，重子节点。

```

1  const int MAXN = 1e5+5;
2  int dep[MAXN],siz[MAXN],fa[MAXN],hson[MAXN];
3  //siz 是子树大小; hson 是重子节点
4  void dfs1(int u, int d)
5  {
6      siz[u] = 1;    //千万别忘了把 size 初始赋为 1
7      dep[u] = d;
8      for(int i = e[u].size()-1;i >= 0;i--)
9      {
10         int v = e[u][i];
11         if(v == fa[u]) continue;
12         fa[v] = u;
13         dfs1(v, d+1);
14         if(siz[v] > siz[hson[u]]) hson[u] = v; //取 size 最大的为重子节点
15         siz[u] += siz[v];
16     }
17 }
```

第二个 dfs 记录每个点所在链的链顶（一般来说要初始化为该点本身）、dfn，即 dfs 序、rnk，即 dfs 序对应的编号。

dfs 时，要求优先遍历重子节点。这样就可以保证一条链上的 dfn 是连续的。

```

1  const int MAXN = 1e5+5;
2  int top[MAXN],dfn[MAXN],rnk[MAXN],dfncnt;
3  void dfs2(int u, int t)
4  {
5      top[u] = t; //记录链顶
6      dfn[u] = ++dfncnt; //记录 dfn 序
7      rnk[dfncnt] = u;
8      if(hson[u]) dfs2(hson[u], t); //先遍历重子节点，注意一定要判断是否有重子节点
9      for(int i = e[u].size()-1;i >= 0;i--)
10     {
11         int v = e[u][i];
12         if(v == hson[u] || v == fa[u]) continue;
13         //到这里 v 一定是轻子节点
14         dfs2(v, v); //上面说过了，轻子节点一定是链顶，应此第二个参数填 v
15     }
16 }
```

重链剖分的变量还是挺多的，有时候还会忘记操作。因此可以先把数组都打好，然后对着有的数组来检查操作是否都做完了。亲测好用。

重链剖分性质

树上每个节点都属于且仅属于一条重链。

Chapter 6

连通性相关

6.1 相关定义与前置知识

无向图的连通性主要研究割点和割边。

- 割点：在无向图中，删去该点后会使得连通分量数增加的点为 **割点**。
- 割边：在无向图中，删去该边后会使得连通分量数增加的边为 **割边**，也称作 **桥**。

孤立点和孤立边上的端点都不是割点，但孤立边是割边（这其实是显然的）。

显然，割点和割边重要的原因就是删去他们后相较于非割点和非割边对图的连通性有更大的影响。下面还有几个概念，都是基于刚才的割点和割边的。

- 点双连通图：不存在割点的无向连通图被称为 **点双连通图**。孤立点和孤立边均为点双连通图。
- 边双连通图：不存在割边的无向连通图被称为 **边双连通图**。孤立点是边双连通图，孤立边不是。
- 点双连通分量：一张图的极大点双连通子图称为 **点双连通分量 (V-BCC)**，简称 **点双**。
- 边双连通分量：一张图的极大边双连通子图称为 **边双连通分量 (E-BCC)**，简称 **边双**。

在连通性这一块，我们用的均为 Tarjan 算法。因此，还需要知道 **DFS 生成树**。

有向图的 DFS 生成树主要有 4 种边（不一定全部出现）：

- 树边 (tree edge)：绿色边，每次搜索找到一个还没有访问过的结点时就形成了一条树边。
- 返祖边 (back edge)：黄色边，即指向祖先结点的边（Tarjan 中用栈来判断）。
- 横插边 (cross edge)：红色边，在搜索过程中访问过且不在栈中的结点的边。
- 前向边 (forward edge)：蓝色边，搜索中遇到子树中的结点的边。

对于无向图，不存在横插边和前向边。

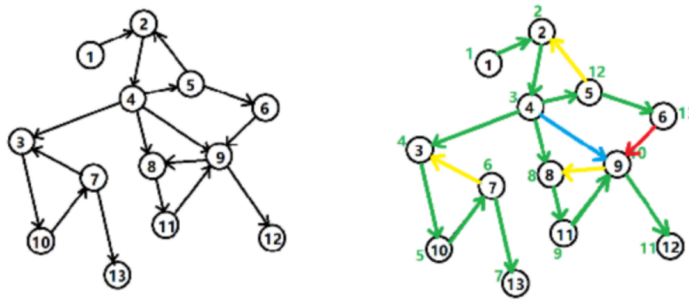


图 6.1: DFS 生成树图例

其实说实话这些东西知道了也没啥用。

6.2 割点

6.2.1 过程

不难想到可以尝试删除一个点，然后判断此图的连通性。但这样做复杂度会极高。因此我们需要用到 tarjan 算法。

设点 $x \in V$ 的子树为 x 在 DFS 树上的子树，包含 x 本身，记其为 $T(x)$ 。记 $\mathcal{C}_V T(x) = T'(x)$ 。

对于非根节点的判定：

设 x 不为 DFS 生成树的根，则 $T'(x) \neq \emptyset$ 。

若 x 为割点，则删去后，对于 $z \in T'(x)$ ，存在 y 使 y, z 不连通，而删除 x 后，所有 $T'(x)$ 的边均存在，则 $y \in T(x)$ 。如果 y 与 z 不连通，因为 $T'(x)$ 的连通性不变，既均连通，则 y 与所有 $T'(x)$ 中的点都不连通。

推出：若存在 $y \in T(x)$ 在不经 x 的情况下与 $T'(x)$ 中所有点均不连通，则 x 就是割点。

现在我们思考怎么判断对于一个 $y \in T(x)$ ，是否在不经 x 的情况下与 $T'(x)$ 中所有点都不连通。

注意到，如果 $y \in T(x)$ 不经 x 就和 $T'(x)$ 连通，就一定存在从 y 到 $T'(x)$ 的一条路径。我们称这条路径在 $T'(x)$ 的第一个结点为 v ，倒数第二个，也就是最后一个在 $T(x)$ 上的结点为 u 。如果 (u, v) 是树边，那么 $u = x$ ，矛盾。则 v 应该是 u 的祖先。又因为 x 是 u 的祖先， $v \in T'(x)$ ，则 v 也是 x 的祖先。

这时我们发现， x 不是割点，那么就会有一条非树边使 $y \in T(x)$ 连向 x 的祖先。这个可以用时间戳来判断。设 f_x 表示与 x 通过非树边相连的点的时间戳的最小值，则可写为 $f_x < dfn_x$ 。

因为 x 的不同儿子子树之间没有非树边（子树独立性），设 x 的儿子 y' 的子树包含 y ，即 $y \in T(y')$ 。

- 对于 $T(y')$, 如果存在 $u \in T(y')$ 满足 $f_u < dfn_u$, 那么删除 x 后 $T(y')$ 的每个点与 $T'(x)$ 均连通。这时 u 与 $T'(x)$ 中的某个点通过非树边直接连通, $T(y')$ 中的所有点均通过树边连通。此时 x 不是割点。
- 反之, 如果不存在 $u \in T(y')$ 满足 $f_u < dfn_u$, 那么删除 x 后 $T(y')$ 的每个点与 $T'(x)$ 均不连通。此时 x 是割点。且只需要出现一个 y' 满足这个条件就可以判断出 x 是割点。

原本我们需要判断 $T(x)$ 上的所有点去检查 $f_u < dfn_u$ 的情况是否出现。现在令 low_x 表示与 x 及其子树上的所有点通过非树边相连接的点的时间戳的最小值, 则可以只通过 low_x 判断即可。这样我们就得到了在 x 为非根节点情况下的割点判定方法:

对于一个非根节点 x , x 是割点当且仅当存在 y 是 x 的子节点, 有 $low_y \geq dfn_x$ 。

对于根节点的判定:

根节点的判定相对简单。

设点 x 是 DFS 树上的根节点。

若 x 有大于一个子节点, 根据子树独立性, 删去 x 后其子节点的子树各不相通, 所以 x 是割点。反之, 若 x 有小于等于一个子节点, 删去后剩余部分通过树边连通, x 不是割点。

这样我们就得到了判断割点的完整方法。

6.2.2 实现

解释

dfn : 某个点被遍历到的时间戳。

low : 某个点通过非树边到达最早的点的时间戳。

is_cut : 是否为割点。

$cutcnt$: 统计割点个数。

在 dfs 过程中求出 dfn 和 low , 然后进行对割点的判断。

代码

此代码用 vector 存图。

```

1 vector<int> e[MAXN];
2 int dfn[MAXN];
3 int low[MAXN];
4 int cnt;
5 bool is_cut[MAXN];
6 int cutcnt;
7
8 void tarjan(int u, int f, int root)
9 {
10     dfn[u] = low[u] = ++cnt;
11     int son = 0; //用于统计子节点个数

```

```

12     for(int i = e[u].size()-1; i >= 0; i--)
13     {
14         int v = e[u][i];
15         if(v == f) continue;
16         if(!dfn[v])//说明这个 v 是在 x 的子树中的
17         {
18             son++;
19             tarjan(v,u,root);
20             low[u] = min(low[u],low[v]);
21             if(dfn[u] <= low[v])
22             {
23                 if(!is_cut[u] && u != root)
24                 {
25                     //非根节点的情况
26                     cutcnt++;
27                     is_cut[u] = true;
28                 }
29             }
30         }
31         else low[u] = min(low[u],dfn[v]);
32         //这个 else 中 v 不再 x 的子树当中。
33         //而我们求的是 x 子树当中 low 的最小值。
34         //因此 min 的第二个参数是 dfn[v] 而不是 low[v]。
35     }
36     if(u == root && son >= 2) is_cut[root] = true, cutcnt++;
37     //根节点的情况
38 }

```

6.3 割边

6.3.1 过程

和割点差不多。

树边能使整张图连通，因此割掉非树边后不影响连通性。因此 $e = (u, v)$ 是割边的必要条件是 e 为树边。

不妨设 v 是 u 的儿子。若割掉 e 后图不连通，根据求割点的经验，不难发现当 $low_v > dfn_u$ 时， e 为割边。

6.3.2 实现

还没写...

6.4 点双连通分量

6.4.1 实现

6.5 边双连通分量

6.5.1 实现

6.6 强连通分量

6.6.1 实现

解释

sta: 栈。

top: 栈顶。

insta: 判断是否在栈中。

belong: 判断点 *i* 在哪个强连通分量中。

代码

此代码用 vector 存图。

```
1 vector<int> e[MAXN];
2 int low[MAXN],dfn[MAXN],dfncnt,belong[MAXN],sc;
3 int sta[MAXN],top;bool insta[MAXN];
4 void tarjan(int u)
5 {
6     low[u] = dfn[u] = ++dfncnt;
7     sta[++top] = u,insta[u] = true;
8     for(int i = e[u].size()-1;i >= 0;i--)
9     {
10         int v = e[u][i];
11         if(!dfn[v])
12         {
13             tarjan(v);
14             low[u] = min(low[u],low[v]);
15         }
16         else if(insta[v])
17         {
18             low[u] = min(low[u],low[v]);
19         }
20     }
21
22     if(dfn[u] == low[u])
23     {
24         sc++;int now;
25         do
```

```
26     {
27         now = sta[top];
28         insta[now] = false;
29         belong[now] = sc;
30         sz[sc]++;
31     } while (sta[top--] != u);
32 }
33 }
```