ComputerNote

Anonymous

Contents

1 算法

1.1 Disjoint Set Union(并查集:DSU 算法)

并查集，在一些有 N 个元素的集合应用问题中，我们通常是在开始时让每个元素构成一个单元素的集合，然后按一定顺序将属于同一组的元素所在的集合合并，其间要反复查找一个元素在哪个集合中。

使用并查集时，首先会存在一组不相交的动态集合 $S = \{S_1, S_2, \cdots, S_k\}$，一般都会使用一个整数表示集合中的一个元素。

每个集合可能包含一个或多个元素，并选出集合中的某个元素作为代表。每个集合中具体包含了哪些元素是不关心的，具体选择哪个元素作为代表一般也是不关心的。我们关心的是，对于给定的元素，可以很快的找到这个元素所在的集合（的代表），以及合并两个元素所在的集合，而且这些操作的时间复杂度都是常数级的。

并查集的基本操作有三个：

makeSet(s)：建立一个新的并查集，其中包含 s 个单元素集合。

unionSet(x, y)：把元素 x 和元素 y 所在的集合合并，要求 x 和 y 所在的集合不相交，如果相交则不合并。

find(x)：找到元素 x 所在的集合的代表，该操作也可以用于判断两个元素是否位于同一个集合，只要将它们各自的代表比较一下就可以了。

并查集的实现原理也比较简单，就是使用树来表示集合，树的每个节点就表示集合中的一个元素，树根对应的元素就是该集合的代表，如图 1-1-1 所示。

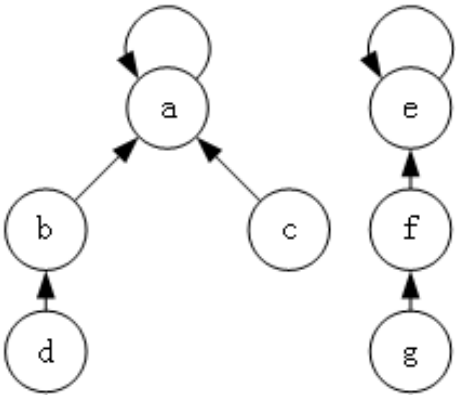

图 1-1-1 并查集的树表示

图中有两棵树，分别对应两个集合，其中第一个集合为 $\{a, b, c, d\}$，代表元素是 a；第二个集合为 $\{e, f, g\}$，代表元素是 e。

树的节点表示集合中的元素，指针表示指向父节点的指针，根节点的指针指向自己，表示其没有父节点。沿着每个节点的父节点不断向上查找，最终就可以找到该树的根节点，即该集合的代表元素。

现在，应该可以很容易的写出 makeSet 和 find 的代码了，假设使用一个足够长的数组来存储树节点（很类似之前讲到的静态链表），那么 makeSet 要做的就是构造出如图 2 的森林，其中每个元素都是一个单元素集合，即父节点是其自身：

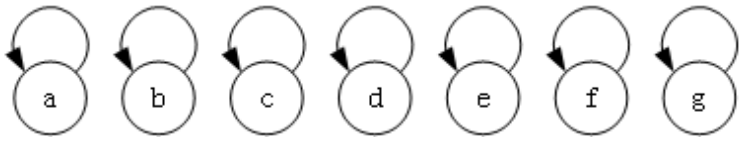

图 1-1-2 构造并查集初始化

接下来，就是 find 操作了，如果每次都沿着父节点向上查找，那时间复杂度就是树的高度，完全不可能达到常数级。这里需要应用一种非常简单而有效的策略——路径压缩。

路径压缩，就是在每次查找时，令查找路径上的每个节点都直接指向根节点，如图 1-1-3 所示。

最后是合并操作 unionSet，并查集的合并也非常简单，就是将一个集合的树根指向另一个集合的树根，如图 1-1-4 所示。

这里也可以应用一个简单的启发式策略——按秩合并。该方法使用秩来表示树高度的上界，在合并时，总是将具有较小秩的树根指向具有较大秩的树根。简单的说，就是总是将比较矮的树作为子树，添加到较高的树中。为了保存秩，需要额外使用一个与 uset 同长度的数组，并将所有元素都初始化为 0。

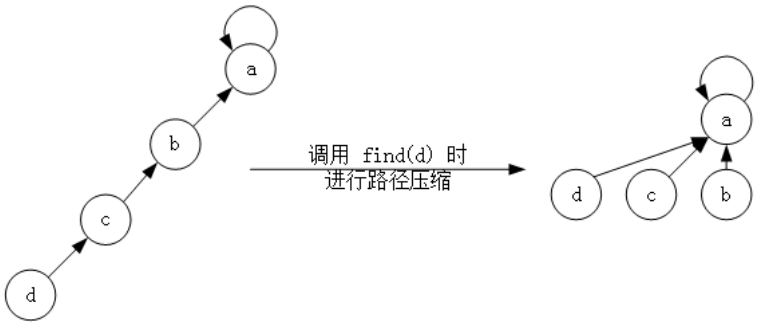

图 1-1-3 路径压缩

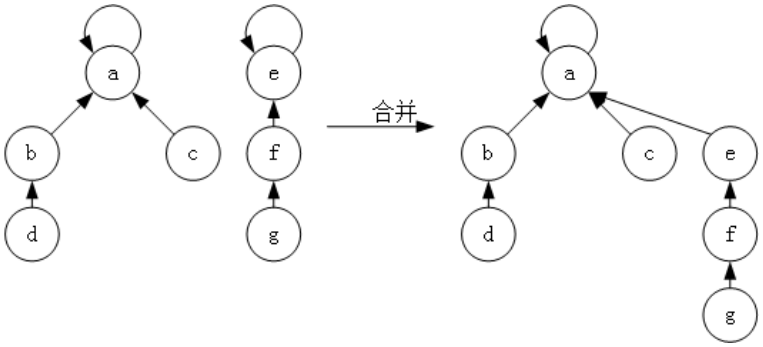

图 1-1-4 并查集的合并

除了按秩合并，并查集还有一种常见的策略，就是按集合中包含的元素个数（或者说树中的节点数）合并，将包含节点较少的树根，指向包含节点较多的树根。这个策略与按秩合并的策略类似，同样可以提升并查集的运行速度，而且省去了额外的 rank 数组。

并查集的空间复杂度是 $O(n)$ 的，这个很显然，如果是按秩合并的，占的空间要多一些。find 和 unionSet 操作都可以看成是常数级的，或者准确来说，在一个包含 n 个元素的并查集中，进行 m 次查找或合并操作，最坏情况下所需的时间为 $O(m\alpha(n))$，这里的 α 是 Ackerman 函数的某个反函数，在极大的范围内（比可观察到的宇宙中估计的原子数量 10^{80} 还大很多）都可以认为是不大于 4 的。

Description:

若某个家族人员过于庞大，要判断两个是否是亲戚，确实还很不容易，给出某个亲戚关系图，求任意给出的两个人是否具有亲戚关系。规定：x 和 y 是亲戚，y 和 z 是亲戚，那么 x 和 z 也是亲戚。如果 x,y 是亲戚，那么 x 的亲戚都是 y 的亲戚，y 的亲戚也都是 x 的亲戚。

Input:

第一行：三个整数 n,m,p，（$n \leq 5000, m \leq 5000, p \leq 5000$），分别表示有 n 个人，m 个亲戚关系，询问 p 对亲戚关系。以下 m 行：每行两个数 M_i M_j，$1 \leq M_i$ $M_j \leq N$，表示 M_i M_j 具有亲戚关系。接下来 p 行：每行两个数 P_i P_j 询问 P_i 和 P_j 是否具有亲戚关系。

Output:

P 行，每行一个'Yes'或'No'。表示第 i 个询问的答案为“具有”或“不具有”亲戚关系。

1.2 Meet-in-the-middle(split and merge)

Meet in the middle(有时候也叫作 split and merge) 是一种用以获取足够高效解决方案的灵巧的思想。和分治思想非常类似，它将问题分割成两个部分，然后试着合并这两个子问题的结果。好处在于通过使用一点额外的空间，你可以解决两倍规模的原来可以解决的问题。

4 和问题（流行的面试问题）

给定一个整数数组 A，问数组中是否存在 4 个数，使得这 4 个数的和是 0（同一个元素可以被多次使用）。例如：数组 A = [2, 3, 1, 0, -4, -1]，一种可能的方案是 3 + 1 + 0 - 4 = 0 或 0 + 0 + 0 + 0 = 0。

朴素的算法是判断所有可能的 4 个数的组合，这种方案需要计算 $O(N^4)$ 次。

一个些微改进的算法是暴力搜索所有的可能的 n^3 个 3 个数的组合,并且用 hash 表来判断-(a+b+c) 是否在原始的数组中。这个算法的复杂度是 $O(n^3)$ 到目前为止, 你可能想知道 meet in the middle 在这里要怎么应用，最关键的洞察来自于改写 $a+b+c+d=0$ 成 $a+b=-(c+d)$。现在我们可以存储 n^2 个 a+b 的和在一个 hash 表 S 中，然后可以枚举所有可能 c 和 d 的 n^2 种组合并且判断 S 中是否包括-(c+d)。

这个算法的时间复杂度和空间复杂度都是 $O(n^2)$，这个问题没有已知的更快的算法

双向搜索

1.3 字符串匹配算法

1.3.1 KMP

1.4 排序算法

1.4.1 快速排序

1.4.2 拓扑排序

定义：将有向图中的顶点以线性方式进行排序。即对于任何连接自顶点 u 到顶点 v 的有向边 uv，在最后的排序结果中，顶点 u 总是在顶点 v 的前面。

一个有向图能被拓扑排序的充要条件就是它是一个有向无环图 (DAG：Directed Acyclic Graph)。

复杂度分析：复杂度同 DFS 一致，即 O(E+V)。具体而言，首先需要保证图是有向无环图，判断图是 DAG 可以使用基于 DFS 的算法，复杂度为 O(E+V)，而后面的拓扑排序也是依赖于 DFS，复杂度为 O(E+V)

1.5 动态规划

1.6 贪心算法

2 数据结构

2.1 链表

2.2 队列

2.3 栈

2.4 树

2.4.1 B-Tree

2.4.2 AVL Tree

2.4.3 Red-Black Tree

2.4.4 VEb Tree

2.4.5 Dancing Linker

https://www.jianshu.com/p/93b52c37cc65

2.4.6 Spanning Tree

Spanning Tree:

In the mathematical field of graph theory, a spanning tree T of an undirected graph G is a subgraph that is a tree which includes all of the vertices of G, with minimum possible number of edges. In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree (but see Spanning forests below). If all of the edges of G are also edges of a spanning tree T of G, then G is a tree and is identical to T (that is, a tree has a unique spanning tree and it is itself).

Minium Spanning Tree:

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted (un)directed graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.

2.5 图

2.6 堆

2.7 散列表

3 编程语言

3.1 C/C++

3.1.1 内存分配

3.1.2 STL

3.2 JAVA

3.3 Python

3.3.1 装饰器

4 例题

4.1 求数组的最大子序列和

给定整数 $A_1, A_2, \ldots, A_N$(可能有负数)，求 $\sum_{k=i}^{j} A_k$ 的最大值（为方便起见，如果所有的整数均为负数，则最大子序列和为 0。

代码如下, 时间复杂度为 $O(N)$。

```
int MaxSubsequenceSum(const int A[], int N) {
    int ThisSum, MaxSum, j;
    ThisSum = MaxSum = 0;
    for (j = 0; j < N; j++) {
        ThisSum += A[j];
        if (ThisSum > MaxSum) {
            MaxSum = ThisSum;
        } else if (ThisSum < 0) {
            ThisSum = 0;
        }
    }
    return MaxSum;
```

```
}
```

4.2 Binary Search

给定一个整数 X 和整数 $A_0, A_1, \ldots, A_{N-1}$, 后者已经预选排序好，求使得 $A_i = X$ 的下标 i, 如果 X 不在数据中，则返回 i = -1。

4.3 A + B

Write a function that add two numbers A and B. You should not use + or any arithmetic operators.

Code:

```
int aplusb(int a, int b) {
    // 主要利用异或运算来完成
    // 异或运算有一个别名叫做：不进位加法
    // 那么a ^ b就是a和b相加之后，该进位的地方不进位的结果
    // 然后下面考虑哪些地方要进位，自然是a和b里都是1的地方
    // a & b就是a和b里都是1的那些位置，a & b << 1 就是进位
    // 之后的结果。所以：a + b = (a ^ b) + (a & b << 1)
    // 令a' = a ^ b, b' = (a & b) << 1
    // 可以知道，这个过程是在模拟加法的运算过程，进位不可能
    // 一直持续，所以b最终会变为0。因此重复做上述操作就可以
    // 求得a + b的值。
    while (b != 0) {
        int _a = a ^b;
        int _b = (a & b) << 1;
        a = _a;
        b = _b;
    }
    return a;
}
```

4.4 Trailing Zeros

Write an algorithm which computes the number of trailing zeros in n factorial.

Code:

```cpp
long long trailingZeros(long long n) {
    long long anw = 0;
    while (n >= 5) {
        anw += n / 5;
        n = n / 5;
    }
    return anw;
}
```

4.5 ugly number II

Ugly number is a number that only have factors 2, 3 and 5.

Design an algorithm to find the nth ugly number. The first 10 ugly numbers are 1, 2, 3, 4, 5, 6, 8, 9, 10, 12...

O(nlogn) or O(n) time.

Notice:

Note that 1 is typically treated as an ugly number.

```cpp
#define min(a, b) ((a) < (b)? (a):(b))
#define min3(a, b, c) (min(min(a, b),min(a, c)))
int nthUglyNumber(int n) {
    // write your code here
    int i = 1;
    int p2 = 0;
    int p3 = 0;
    int p5 = 0;
    int uglyNum[6048] = {0};
    uglyNum[0] = 1;
    while (i < n) {
        uglyNum[i] = min3(uglyNum[p2] * 2, uglyNum[p3] * 3, uglyNum[p5] * 5);
        if (uglyNum[i] == uglyNum[p2] * 2) {
            p2++;
        }
        if (uglyNum[i] == uglyNum[p3] * 3) {
            p3++;
```

```
        }
        if (uglyNum[i] == uglyNum[p5] * 5) {
            p5++;
        }
        i++;
    }
    return uglyNum[n - 1];
}
```

5 其他

5.1 位移运算

符号	描述	运算规则
&	与	两个位都为 1 时，结果才为 1
|	或	两个位都为 0 时，结果才为 0
∧	异或	两个位相同为 0，相异为 1
~	取反	0 变 1，1 变 0
≪	左移	各二进位全部左移若干位，高位丢弃，低位补 0
≫	右移	各二进位全部右移若干位，对无符号数，高位补 0，有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补 0（逻辑右移）

表 1: 常见位移运算

位移运算 注意以下几点：

1．在这 6 种操作符，只有 ~ 取反是单目操作符，其它 5 种都是双目操作符。

2．位操作只能用于整形数据，对 float 和 double 类型进行位操作会被编译器报错。

3．对于移位操作，在微软的 VC6.0 和 VS2008 编译器都是采取算术称位即算术移位操作，算术移位是相对于逻辑移位，它们在左移操作中都一样，低位补 0 即可，但在右移中逻辑移位的高位补 0 而算术移位的高位是补符号位。

4．位操作符的运算优先级比较低，因为尽量使用括号来确保运算顺序，否则很可能会得到莫明其妙的结果。比如要得到像 1，3，5，9 这些 2^i+1 的数字。写成 int a = 1 « i + 1; 是不对的，程序会先执行 i + 1，再执行左移操作。应该写成 int a = (1 « i) + 1;

5．另外位操作还有一些复合操作符，如 &=、|=、∧=、«=、»=。

位移运算技巧

判断奇偶 只要根据最未位是 0 还是 1 来决定，为 0 就是偶数，为 1 就是奇数。因此可以用 if ((a & 1) == 0) 代替 if (a % 2 == 0) 来判断 a 是不是偶数。

交换两数 交换 a 和 b

```
void Swap(int &a, int &b){
  if (a != b){
    a ^= b;
    b ^= a;
    a ^= b;
  }
}
```