

# 二分与三分

## 一、二分

二分法适用于求解具有单调性的问题。

对于在实数区间 $[L, R]$ 内递增的连续函数 $f(x)$ ，求 $[L, R]$ 内 $f(x)$ 的零点 $J$ 。

$$\forall L \leq x < J, f(x) < 0$$

$$\forall J < x \leq R, f(x) > 0$$

$$f(J) = 0$$

二分法的思想是不断将待求解区间平均分成两份，根据求解区间中点的情况来确定答案所在的区间，这样就把解的范围缩小了一半。

设当前求解区间为 $[l, r]$ ，它的中点为 $m = \frac{l+r}{2}$ ，则有：

若 $f(m) < 0$ 则 $J \in [m, r]$ ；若 $f(m) > 0$ 则 $J \in [l, m]$ ；若 $f(m) = 0$ 则 $J = m$ 。

类似上述例子的问题往往都可考虑二分求解。在用二分法解决类似问题时，类似于在考虑一个布尔表达式 $E$ ，在问题定义域 $[L, R]$ 内存在一个分界点 $J$ ：

$$\forall L \leq x \leq J, E(x) = true$$

$$\forall J < x \leq R, E(x) = false$$

若求解的问题的定义域为整数域，那么根据写法的不同，算法会在 $L = R$ 或 $L = R + 1$ 时结束，而此时对于长度为 $N$ 的求解区间，算法需要 $\log_2 N$ 次来确定出分界点。

对于定义域在实数域上的问题，可以类似于上面的方法，判断 $R - L$ 的精度是否达到要求，即 $R - L \geq eps$ ，但由于实数运算带来的精度问题，若 $eps$ 取得太小就会导致程序死循环，因此往往指定二分次数更好。

显然，二分算法的复杂度为 $O(\text{二分次数} \times \text{单次判定复杂度})$

## 二、二分写法

整数定义域上二分：

```
1. int L = 1, R = n;
2. while (L <= R) {
3.     int mid = L + R >> 1;
4.     if (check(mid)) L = mid + 1;
```

```
5.     else R = mid - 1;
6. }
```

这种写法不会陷入死循环，算法结束后会有  $L=R+1$ 。稍微麻烦的是答案的取值，需要根据具体问题的要求来确定答案是  $L$  还是  $R$ 。

实数域上二分：

```
1. double l = 0, r = n;
2. for (int t = 0; t < 60; ++t) {
3.     double mid = (l + r) / 2;
4.     if (check(mid)) l = mid;
5.     else r = mid;
6. }
```

就像上面所说，我们指定二分的次数  $t$ ，那么对于初始的求解区间长度  $L$ ，算法结束后  $r-l$  的值会为  $\frac{L}{2^t}$ ，根据这个值来判断我们的精度是否达到要求即可。

## 三、二分法常见模型

### 二分答案

最小值最大（或是最大值最小）问题，这类双最值问题常常使用二分法求解，也就是确定答案后，配合贪心或 DP 等其他算法来检验这个答案是否合法，将最优化问题转变为判定性问题。例如：将长度为  $n$  的序列  $a_i$  分成最多  $m$  个连续段，求所有分法中每段和的最大值最小能是多少。

### 二分查找

具有单调性的布尔表达式求解分界点，比如在有序数列中求解数字  $x$  的排名。

### 代替三分

有时对于一些单峰函数，我们可以通过二分导函数的方法求解函数极值，这样使用时通常定义域为整数域比较方便，因为此时  $dx$  可以直接取整数 1。

## 四、例题

题目大意：

## Aggressive cows (POJ No.2456)

农夫约翰搭了一间有  $N$  间牛舍的小屋。牛舍排在一条线上，第  $i$  号牛舍在  $x_i$  的位置。但是他的  $M$  头牛对小屋很不满意，因此经常互相攻击。约翰为了防止牛之间互相伤害，因此决定把每头牛都放在离其他牛尽可能远的牛舍。也就是要最大化最近的两头牛之间的距离。

### 限制条件

- $2 \leq N \leq 100000$
- $2 \leq M \leq N$
- $0 \leq x_i \leq 10^9$

### 分析：

类似的最大化最小值或者最小化最大值的问题，通常用二分搜索法就可以很好地解决。我们定义

$C(d)$  := 可以安排牛的位置使得最近的两头牛的距离不小于  $d$

那么问题就变成了求满足  $C(d)$  的最大的  $d$ 。另外，最近的间距不小于  $d$  也可以说成是所有牛的间距都不小于  $d$ ，因此就有

$C(d)$  = 可以安排牛的位置使得任意的牛的间距都不小于  $d$

这个问题的判断使用贪心法便可非常容易地求解。

- 对牛舍的位置  $x$  进行排序
- 把第一头牛放入  $x_0$  的牛舍
- 如果第  $i$  头牛放入了  $x_j$  的话，第  $i+1$  头牛就要放入满足  $x_j + d \leq x_k$  的最小的  $x_k$  中

对  $x$  的排序只需在最开始时进行一次就可以了，每一次判断对每头牛最多进行一次处理，因此复杂度是  $O(N)$ 。

### 代码：

```
1. #include <cstdio>
2. #include <algorithm>
3.
4. const int N = 1e5 + 3;
5. int n, m, x[N];
6.
7. inline bool check(int d) {
8.     int cow = 1;
9.     int rgt = x[1] + d;
10.    for (int i = 2; i <= n; ++i) {
11.        if (x[i] < rgt) continue;
12.        ++cow; rgt = x[i] + d;
13.    }
14.    return cow >= m;
15. }
16.
17. int main() {
18.    scanf("%d%d", &n, &m);
```

```
19. for (int i = 1; i <= n; ++i) scanf("%d", &x[i]);
20. std::sort(x + 1, x + n + 1);
21. int l = 0, r = x[n] - x[1];
22. while (l <= r) {
23.     int mid = l + r >> 1;
24.     if (check(mid)) l = mid + 1;
25.     else r = mid - 1;
26. }
27. printf("%d\n", r);
28. return 0;
29. }
```

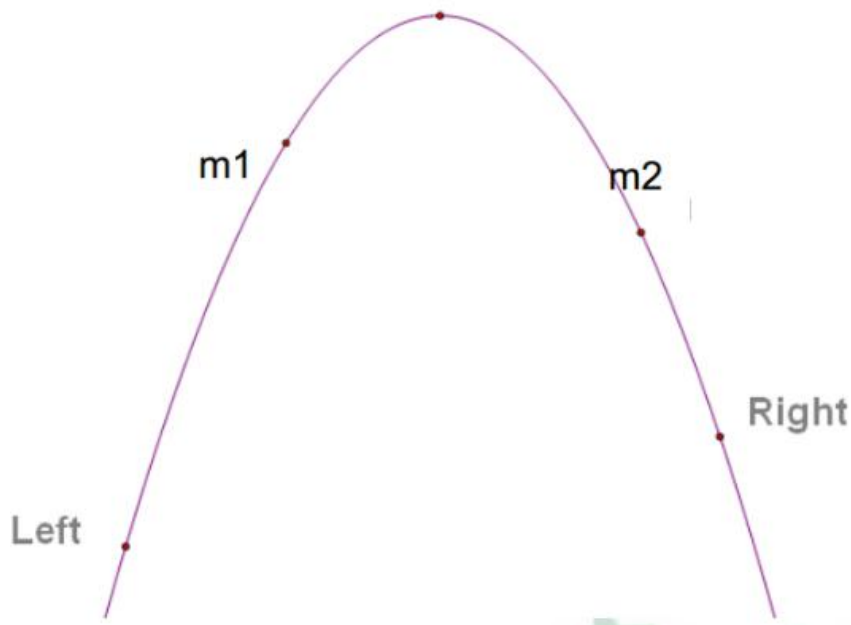
## 五、三分

三分法适用于求解单峰函数的极值问题。二次函数就是一个典型的单峰函数。

三分法与二分法一样，它会不断缩小答案所在的求解区间。二分法缩短区间利用的原理是函数的单调性，而三分法利用的则是函数的单峰性。

设当前求解的区间为 $[l, r]$ ，令 $m1 = l + \frac{r-l}{3}$ ,  $m2 = r - \frac{r-l}{3}$ ，接着我们计算这两个点的函数值 $f(m1)$ ,  $f(m2)$ ，之后我们将两点中函数值更优的那个点称为好点（若计算最大值，则 $f$ 更大的那个点就为好点，计算最小值同理），而函数值较差的那个点称为坏点。

我们可以证明，最优点与好点会在坏点同侧。如下图， $f(m1) > f(m2)$ ，则 $m1$ 是好点而 $m2$ 是坏点，因此最后的最优点会与 $m1$ 一起在 $m2$ 的左侧，即我们的求解区间由 $[l, r]$ 变成了 $[l, m2]$ 。因此根据这个结论我们可以不停缩短求解区间，直至可以得出近似解。



与二分一样，我们可以指定三分的次数，或是根据  $r - l$  的值来终止算法。

以求上凸单峰函数的最大值为例，三分的代码：

```
1. double l = 0, r = 1e9;
2. while (r - l >= 1e-3) {
3.     double m1 = l + (r - l) / 3, m2 = r - (r - l) / 3;
4.     if (f(m1) < f(m2)) l = m1;
5.     else r = m2;
6. }
```

## 六、例题

### UmBasketella (POJ 3737)

题目大意：

给定圆锥的表面积  $S$ ，求这个圆锥的最大体积  $V$ ，以及此时它的高  $h$  与底面半径  $r$ 。

分析：

$$S = \pi r l + \pi r^2$$

$$h = \sqrt{l^2 - r^2}$$

$$V = \frac{1}{3} \pi r^2 h$$

若确定了底面半径，则其他值都可以依次计算出来，同时根据上面几个有关圆锥的公式也可以看出， $V$  是关于  $r$  的一个单峰函数，因此三分底面半径求解，

代码：

```

1. #include <cmath>
2. #include <cstdio>
3.
4. double PI = acos(-1.0);
5. double S;
6.
7. inline double calc(const double &r) {
8.     double l = (S - r * r) / r;
9.     double h = sqrt(l * l - r * r);
10.    double V = PI * r * r * h / 3.0;
11.    return V;
12. }
13.
14. int main() {
15.     while (scanf("%lf", &S) != EOF) {
16.         S /= PI;
17.         double lft = 0, rgt = sqrt(S), m1, m2, r;
18.         for (int t = 0; t < 200; ++t) {
19.             m1 = lft + (rgt - lft) / 3.0;
20.             m2 = rgt - (rgt - lft) / 3.0;
21.             if (calc(m1) <= calc(m2)) lft = m1, r = m2;
22.             else rgt = m2, r = m1;
23.         }
24.         double l = (S - r * r) / r;
25.         double h = sqrt(l * l - r * r);
26.         double V = PI * r * r * h / 3.0;
27.         printf("%.2f\n%.2f\n%.2f\n", V, h, r);
28.         // POJ 中的 G++编译器输出需要使用.f, 若使用.lf 会出奇怪的错误, 原因不明
29.     }
30. }

```

## 练习题

二分: POJ 1064 ; POJ 3273 ; CF 549H

三分: CF 578C ; BZOJ 1857

# 哈希

## 一、问题引入

寻找长为  $n$  的字符串  $S$  中字符串  $T$ （长度为  $m$ ）出现的位置或次数的问题属于字符串匹配问题。朴素的想法是枚举所有起始位置，再直接检查是否匹配。而检查是否匹配，我们可以不使用  $O(m)$  的直接比较字符串的方法，而是比较长度为  $m$  的字符串  $S$  的子串的哈希值与  $T$  的哈希值是否相等，这也就是哈希算法用于这个问题的原理。

## 二、算法流程

哈希算法的思想就是通过某个函数，将一个字符串转成一个数字，而比较字符串是否相等就只需要比较转化后的数字是否相等，而这个函数我们称其为哈希函数。

虽然即使哈希值相等字符串也未必相等，但如果我们的哈希函数所生成的哈希值是随机分布的话，不同的字符串哈希值相等的概率是很低的，在竞赛中我们常常就认为这种情况不会发生。实际上根据生日悖论，对于哈希值在  $[0, n)$  内均匀分布的哈希函数，它首次发生冲突（也就是不同字符串哈希值相等）的期望步数是  $O(\sqrt{n})$ ，这在选择哈希函数时可以作为效率与正确性的参考。

回到原来的问题，若我们用  $O(m)$  的复杂度计算长为  $m$  的字符串的哈希值，则总时间复杂度并没有改观。这里就需要用到一个叫做滚动哈希的优化技巧。我们选取两个合适的互素常数  $b$  和  $h$  ( $b < h$ )，假设字符串  $C = c_1c_2 \cdots c_m$ ，那么我们定义哈希函数：

$$H(C) = (c_1b^{m-1} + c_2b^{m-2} + \cdots + c_mb^0) \bmod h$$

其中  $b$  是基数，相当于把字符串看做是  $b$  进制数。这样，字符串  $S = s_1s_2s_3 \cdots s_n$  从位置  $k+1$  开始的长度为  $m$  的字符串子串  $S[k+1 \dots k+m]$  的哈希值，就可以利用从位置  $k$  开始的长度为  $m$  的字符串子串  $S[k \dots k+m-1]$  的哈希值计算：

$$H(S[k+1 \dots k+m]) = (H(S[k \dots k+m-1]) \times b - s_kb^m + s_{k+m}) \bmod h$$

于是我们就能在  $O(n)$  时间内得到所有长为  $m$  的字符串子串哈希值，从而在  $O(n+m)$

的时间内完成字符串匹配。在实现时，可以利用 32 位或 64 位无符号整数计算哈希值，并取  $h = 2^{32}$  或  $h = 2^{64}$ ，通过自然溢出省去求模运算。

代码：

```
1.  const int b = 31;
2.  const int h = 1e9 + 7;
3.
4.  int count(char *s, char *t) {
5.      int n = strlen(s);
6.      int m = strlen(t);
7.      int h1 = 0;
8.      for (int i = 0; i < m; ++i)
9.          h1 = (1ll * h1 * b + t[i] - 'a') % h;
10.     int h2 = 0, cnt = 0;
11.     for (int i = 0; i < n; ++i) {
12.         h2 = (1ll * h2 * b + s[i] - 'a') % h;
13.         if (i - m >= 0) h2 = (h2 - pow(b, m) * (s[i - m] - 'a') % h + h) % h;
14.         if (i >= m - 1 && h1 == h2) ++cnt;
15.     }
16.     return cnt;
17. }
```

### 三、哈希表

有时我们需要做的工作是维护一个数据结构来存储数字集合，支持插入数字以及询问数字集合中是否包含某个数。

若数字大小不大，则我们可以通过数组来完成，而当数字大小很大，无法用数字下标表示时，我们考虑设计一个哈希函数，将数字转化成一个较小的数字，并将其进行存储。哈希函数可以类似上面字符串匹配中那样，将数字看成是十进制数进行哈希。

这样做的代价是不能进行自然溢出，或是将  $h$  设为很大的数字，因为我们需要用哈希值当做下标来存储数字，这样无疑会增大产生冲突的概率，解决方法是我们对每个哈希值都新建一个链表，存储所有哈希值等于它的数字。这样当数字计算完哈希值后，需要遍历一遍对应的链表，来判断这个数是否存在于集合中。

通常我们仍然要使  $b, h$  互质，这样才能保证每个链表里的元素个数都是常数个，从而不影响复杂度。

有些时候我们直接将不超过 64 位的整数模  $h$  后存入哈希表，而此时一般尽量保证  $h$  是



素数，这样通常冲突的概率会减小。因为平常我们所存储的数据并不是真随机数，而是伪随机数，他们的生成方式也常常会有规律可循的（就比如上面的哈希，它在不停地乘以一个常数），那么根据同余方面知识，模数取素数能使得冲突概率最小。

搜索时的状态判重也常常用这种哈希表来实现。

哈希表代码：

```
1. namespace Hash {
2.     const int N = 50000;
3.     const int H = 999979;
4.     int tot, adj[H], nxt[N], num[N];
5.     int top, stk[N];
6.
7.     void init() {
8.         tot = 0;
9.         while (top) adj[stk[top--]] = 0;
10.    }
11.
12.    void insert(int x) {
13.        int h = x % H;
14.        for (int e = adj[h]; e; e = nxt[e]) {
15.            if (num[e] == x) return ;
16.        }
17.        if (!adj[h]) stk[++top] = h;
18.        nxt[++tot] = adj[h], adj[h] = tot;
19.        num[tot] = x;
20.    }
21.
22.    bool query(int x) {
23.        int h = x % H;
24.        for (int e = adj[h]; e; e = nxt[e])
25.            if (num[e] == x) return true;
26.        return false;
27.    }
28. }
```

## 练习题

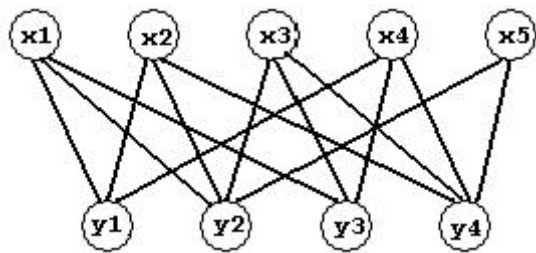
POJ 3690 ; CF 25E

# 二分图匹配

## 一、二分图的原始模型及相关概念

二分图又称作二部图，是图论中的一种特殊模型。

设  $G=(V,E)$  是一个无向图。如顶点集  $V$  可分割为两个互不相交的子集，并且图中每条边依附的两个顶点都分属两个不同的子集。则称图  $G$  为二分图。我们将上边顶点集合称为  $X$  集合，下边顶点结合称为  $Y$  集合，如下图，就是一个二分图。



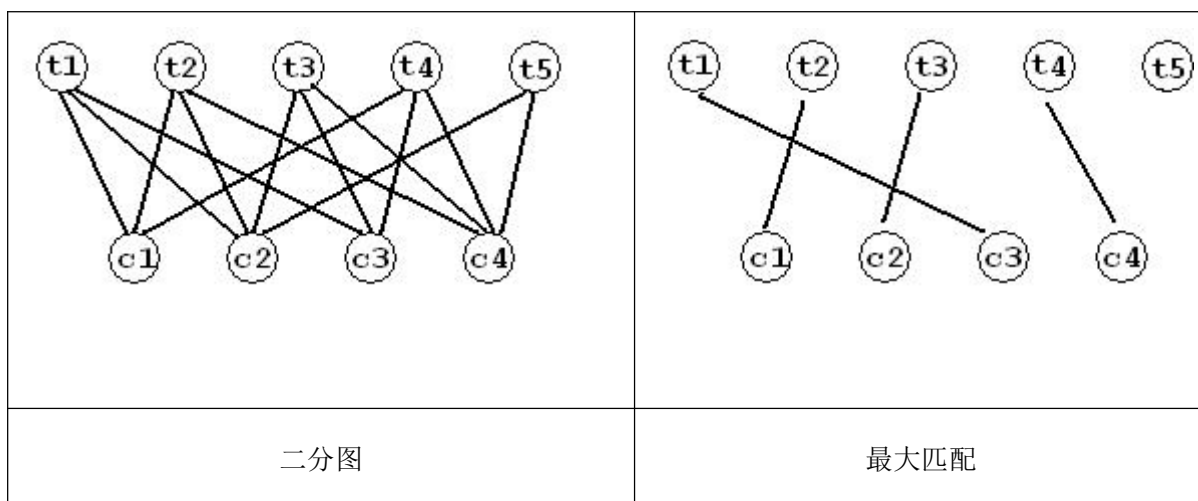
二分图的匹配：

给定一个二分图  $G$ ，在  $G$  的一个子图  $M$  中， $M$  的边集  $E$  中的任意两条边都不依附于同一个顶点，则称  $M$  是一个匹配。

$M$ 是图 $G$ 的一个匹配	$M$ 不是图 $G$ 的一个匹配

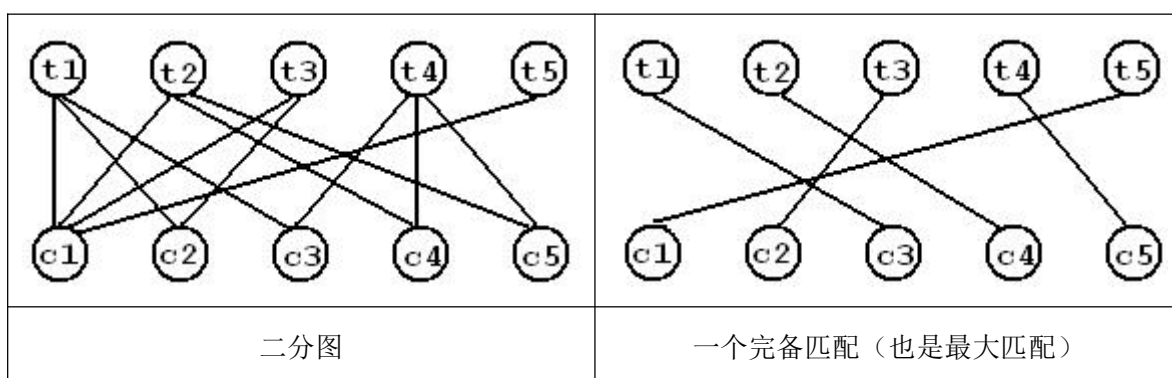
最大匹配：

在二分图  $G$  中所有的匹配  $M$  中，边数最多的匹配，称为二分图的最大匹配。



### 完全匹配：

如果一个匹配中，图中的每个顶点都和图中某条边相关联，则称此匹配为完全匹配，也称作完备匹配。显然，完备匹配必然是一个最大匹配。



由完备匹配的定义可知：一个二分图有完备匹配，那么这个二分图的顶点个数必然为偶数，且它的两个顶点集合的个数相等。

### 最佳匹配：

如果二分图  $G$  的每条边带权的话，权和最大的匹配叫做最佳匹配。

### 最佳完备匹配：

在加权二分图的所有完备匹配中，边权和最大的称为最佳完备匹配。

### 一般图最大匹配：

对于无向图  $G = (V, E)$ ，图中满足两两不含公共端点的边集合  $M \subseteq E$  称为这张图的一个匹配，集合大小  $|M|$  最大的匹配称为最大匹配。

## 二、求解二分图最大匹配

## 使用网络流算法：

实际上，可以将二分图最大匹配问题看成是最大流问题的一种特殊情况。

用网络流算法思想解决最大匹配问题的思路：

首先：建立源点  $s$  和汇点  $t$ ，从  $s$  向  $X$  集合的所有顶点引一条边，容量为 1，从  $Y$  集合的所有顶点向  $T$  引一条边，容量为 1。

然后：将二分图的所有边看成是从  $X_i$  到  $Y_j$  的一条有向边，容量为 1。

求最大匹配就是求  $s$  到  $t$  的最大流。

最大流图中从  $X_i$  到  $Y_j$  有流量的边就是匹配集合中的一条边。

## 使用匈牙利算法：

利用所有边的容量都是 1 以及二分图的性质，我们还可以将二分图最大匹配算法更简单地实现：

**增广路的定义：（增广轨或交错轨）**

假设  $M$  是二分图  $G$  的一个匹配，则称  $M$  中边所依附的顶点是已匹配的顶点，若  $P$  是图  $G$  中一条连通两个未匹配顶点的路径，并且不属于  $M$  的边和属于  $M$  的边（即待匹配的边和已匹配边）在  $P$  上交替出现，则称  $P$  为相对于  $M$  的一条增广路径。

由增广路我们可以得到如下几点推论：

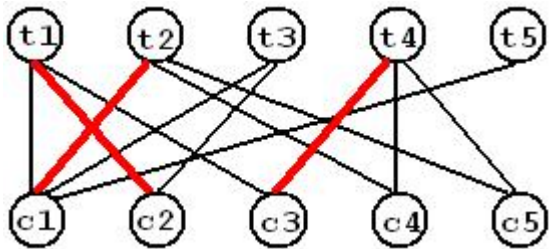
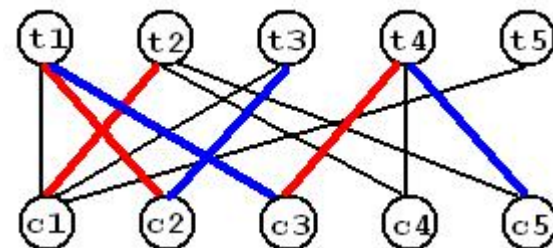
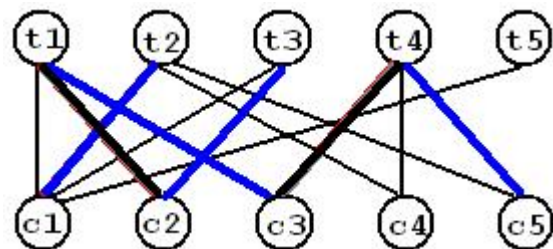
(1)  $P$  的路径长度为奇数，且起点在  $X$  集合、终点在  $Y$  集合，起点和终点都是未匹配的顶点。

(2) 如果我们将  $P$  的路径看着边的集合，我们令  $M' = M \text{ xor } P$ ，则  $M'$  是比  $M$  多一条边的匹配，即更大的匹配  $M'$  包含了，或者属于  $M$ ，或者属于  $P$ ，但不同时属于  $M$  和  $P$  的边。

(3) 当找不到增广路的时候，则这时候的  $M$  是最大匹配。

这个不停寻找增广路的算法就是匈牙利算法。

下面图示出增广路定义及其结论的含义：

	<p>图 G 的一个匹配：  <math>M = \{(t1, c2), (t2, c1), (t4, c3)\}</math>          图中红色粗边标示出的。</p>
	<p>增广路径 <math>P = t3 - c2 - t1 - c3 - t4 - c5</math>;          用集合表示为:  <math>\{(t3, c2), (c2, t1), (t1, c3), (c3, t4), (t4, c5)\}</math></p>
	<p><math>M' = M \text{ xor } p</math>  <math>= \{(t2, c1), (t3, c2), (t1, c3), (t4, c5)\}</math>          得到更大的匹配</p>

由此我们得到匈牙利算法的框架：

- (1) 置 M 为空
- (2) 寻找一条相对于 M 的增广路 P
- (3) 令  $M = M \text{ xor } P$
- (4) 重复步骤(2)、(3)，直到不存在增广路为止。

代码：

```

1. // 尝试从 u 开始寻找一条增广路
2. // mateR 表示 Y 集合中已匹配点所匹配的 X 集合点
3. bool Hungry(int u) {
4.     for (int e = adj[u]; e; e = nxt[e])
5.         if (!mateR[go[e]]) return mateR[go[e]] = u, true;
6.     for (int e = adj[u]; e; e = nxt[e]) {
7.         if (vst[go[e]] == vt) continue;
8.         vst[go[e]] = vt;
9.         if (Hungry(mateR[go[e]]))

```

```

10.     return mateR[go[e]] = u, true;
11. }
12. return false;
13. }

```

对  $X$  部所有点尝试进行一次增广，因此总复杂度为  $O(nm)$ 。

### 三、常见模型

上面已经提到了图的匹配的概念，此外还有几个相关的有用的概念，在此我们再介绍除匹配之外的三个概念：

记图  $G = (V, E)$ 。

匹配：在  $G$  中两两没有公共端点的边集合  $M \subseteq E$ 。

边覆盖：  $G$  中的任意顶点都至少是  $F$  中某条边的端点的边集合  $F \subseteq E$ 。

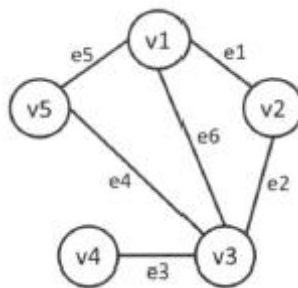
独立集：在  $G$  中两两互不相连的顶点集合  $S \subseteq V$ 。

顶点覆盖：  $G$  中的任意边都有至少一个端点属于  $S$  的顶点集合  $S \subseteq V$ 。

相应的也有：最大匹配，最小边覆盖，最大独立集，最小顶点覆盖。

例如下图中，最大匹配为  $\{e_1, e_3\}$ ，最小边覆盖为  $\{e_1, e_3, e_4\}$ ，最大独立集为  $\{v_2, v_4, v_5\}$ ，

最小顶点覆盖为  $\{v_1, v_3\}$ 。



此外，它们之间还满足：

(1) 对于不存在孤立点的图， $|\text{最大匹配}| + |\text{最小边覆盖}| = |V|$

(2)  $|\text{最大独立集}| + |\text{最小顶点覆盖}| = |V|$

《挑战》中给的简要证明：

证明并不复杂,读者不妨试着思考一下。(a)中可以通过向最大匹配中加边而得到最小边覆盖。而(b)中有 $X \subseteq V$ 是 $G$ 的独立集 $\Leftrightarrow V \setminus X$ 是 $G$ 的顶点覆盖。

对于二分图而言,还有如下等式成立:

$$(3) \quad |\text{最大匹配}| = |\text{最小顶点覆盖}|。$$

对于这些等式,我们知道其中的一个量,就可以求解等式中另一个量。

## 四、例题

题目大意:

### Asteroids (POJ No.3041)

在 $N \times N$ 的网格中有 $K$ 颗小行星。小行星 $i$ 的位置是 $(R_i, C_i)$ 。现在有一个强力武器能够用一发光束将一整行或一整列的小行星轰为灰烬。想要利用这个武器摧毁所有的小行星最少需要几发光束?

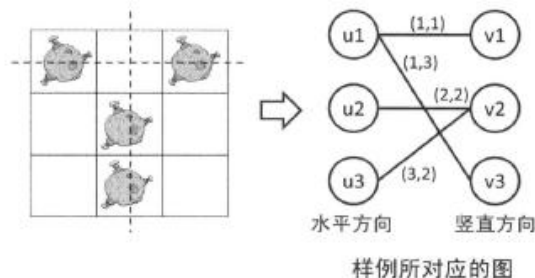
限制条件

- $1 \leq N \leq 500$
- $1 \leq K \leq 10000$
- $1 \leq R, C \leq N$

分析:

光束的攻击选择可以是横坐标从 $x=1$ 到 $x=N$ 和纵坐标从 $y=1$ 到 $y=N$ ,一共 $2N$ 种。显然,同样的选择没有必要执行多次,而攻击的顺序对结果没有影响,所以总的攻击方案共有 $2^{2N}$ 种。我们只要在这个解空间中,寻找能够摧毁所有小行星的最小的解就可以了。要破坏某个小行星,只能通过对应水平方向或竖直方向的光束的攻击。利用攻击方法只有两种这一点,我们可以将问题按如下方法转换为图。

把光束当作图的顶点,而把小行星当作连接对应光束的边。这样转换之后,光束的攻击方案即对应一个顶点集合 $S$ ,而要求攻击方案能够摧毁所有的小行星,也就是图中的每条边都至少有一个属于 $S$ 的端点。这样一来,问题就转为了求最小的满足上述要求的顶点集合 $S$ 。



这正是最小顶点覆盖的问题。之前我们已经介绍过,最小顶点覆盖问题通常是NP困难的,不过在二分图中等于最大匹配,因而可以高效地求解。事实上,本题中所有顶点可以分成水平方向和竖直方向的攻击选择两类,而每颗小行星所对应的边都分别与一个水平方向和一个竖直方向的顶点相连,所以是二分图。因此,只要运用二分图最大匹配算法,问题就会迎刃而解了。

代码:

```

1. #include <cstdio>
2.
3. const int N = 505;
4. const int M = 1e4 + 5;
5. int n, m, vt, vst[N], mateR[N];
6. int ecnt, adj[N], nxt[M], go[M];
7.
8. inline void addEdge(int u, int v) {
9.     nxt[++ecnt] = adj[u], adj[u] = ecnt, go[ecnt] = v;
10. }
11.
12. bool Hungry(int u) {
13.     for (int e = adj[u]; e; e = nxt[e])
14.         if (!mateR[go[e]]) return mateR[go[e]] = u, true;
15.     for (int e = adj[u]; e; e = nxt[e]) {
16.         if (vst[go[e]] == vt) continue;
17.         vst[go[e]] = vt;
18.         if (Hungry(mateR[go[e]]))
19.             return mateR[go[e]] = u, true;
20.     }
21.     return false;
22. }
23.
24. int main() {
25.     scanf("%d%d", &n, &m);
26.     for (int i = 1; i <= m; ++i) {
27.         int x, y;
28.         scanf("%d%d", &x, &y);
29.         addEdge(x, y);
30.     }
31.     int ans = 0;
32.     for (int i = 1; i <= n; ++i) {
33.         ++vt;
34.         if (Hungry(i)) ++ans;
35.     }
36.     printf("%d\n", ans);
37. }

```

## 练习题

POJ 1422 ; POJ 1486 ; POJ 2724



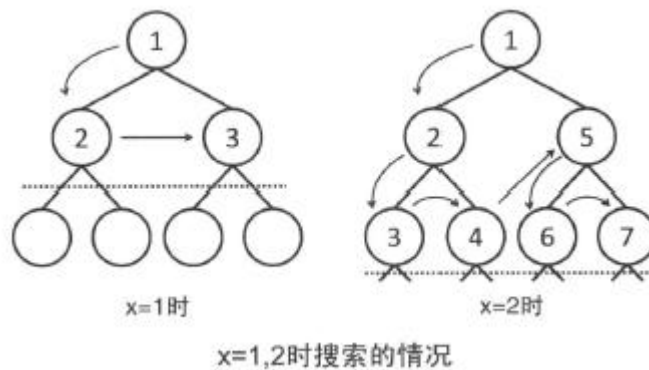
# 迭代加深搜索

## 一、概念

对于在用搜索求解一类步数最少的最优解问题时，我们常常会使用剪枝，其中有一种剪枝叫做最优性剪枝，即：当前搜索的答案大于已找到的最优解，那么就舍弃当前搜索的局面，直接回溯。然而这种剪枝在没有找到较优解时效果并不明显。

此时我们可以改变思路，不去直接求最优解，而是改成通过搜索的办法来判断是否有不超过某个  $x$  的解。把  $x$  从 0 开始每次增加 1，那么首次找到解时的  $x$  便是最优解。这样搜索过程中就不会盲目访问搜索深度过深的劣解结点，也不会搜索到深度比最优解更大的结点。同时还能配合使用上前面我们所说的最优性剪枝。

这个算法会像 BFS 一样，按照距离初始状态的远近顺序访问各个状态，这种搜索就被称为迭代加深搜索(IDDFS)。



而像下面这样，通过估算下界提前剪枝优化后的算法则称为 IDA\*，它通常表述如下：

- (1) 给出状态  $v$  到目标状态的距离下界的估价函数  $h^*(v)$
- (2) 令  $x=0$
- (3) 对满足  $d(v) + h^*(v) \leq x$  的状态进行 DFS，判断是否有不超过  $x$  的解。其中  $d(v)$  表示初始状态到  $v$  的实际距离。
- (4) 如果找到解，则  $x$  为最优解，程序结束
- (5) 否则将  $x+1$  并回到第三步。

IDA\*中所访问的所有状态总是满足  $d(v) + h^*(v) \leq \text{最优解}$ ，而  $h^*(v)$  的估值越接近到目标状态的实际距离，则搜索经过的状态数越少，效率越高，但要保证  $h^*(v)$  的估值小于真实

的值，否则会导致错误的剪枝。

IDA\*可以看做是 DFS 的优化，尽管它在搜索过程中会重复访问某些深度较小的状态，但由于通常来说随着搜索深度增加，搜索空间的大小是呈指数级增长的，所以 IDA\*总的访问状态数与最后一次所访问的状态数是同一级别的。

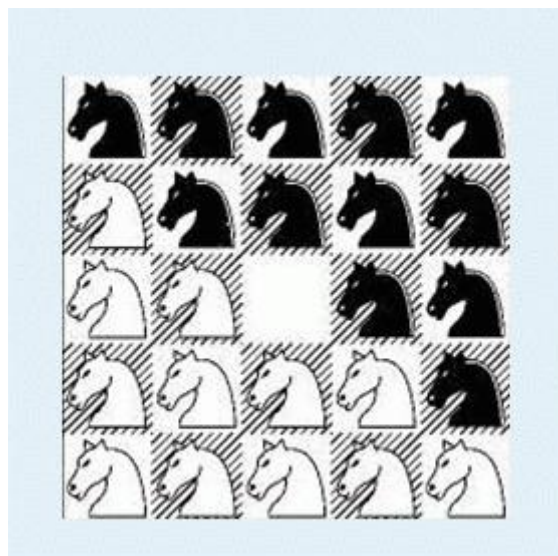
这个算法特别擅长处理深度不大，但每个搜索局面后继较多，且局面容易进行估价的搜索问题。

## 二、例题

### 骑士精神 (BZOJ 1085)

题目大意：

在一个  $5 \times 5$  的棋盘上有 12 个白色的骑士和 12 个黑色的骑士， 且有一个空位。在任何时候一个骑士都能按照骑士的走法（它可以走到和它横坐标相差为 1，纵坐标相差为 2 或者横坐标相差为 2，纵坐标相差为 1 的格子）移动到空位上。 给定一个初始的棋盘，怎样才能经过移动变成如下目标棋盘：



求最少的步数。

分析：

首先这道题的步数并不会很多，并且它的局面可以有一个简单的估价，就是还没有走到正确位置上的棋子的个数，因此我们直接套用 IDA\*算法求解。

代码：

```
1. #include <algorithm>
```

```

2. #include <iostream>
3. #include <cstring>
4. #include <cstdio>
5. #include <cmath>
6. using namespace std;
7.
8. const int N = 5;
9. const char Des[5][5] = {{ '1', '1', '1', '1', '1'},
10.                          {'0', '1', '1', '1', '1'},
11.                          {'0', '0', '*', '1', '1'},
12.                          {'0', '0', '0', '0', '1'},
13.                          {'0', '0', '0', '0', '0'}};
14. const int Move[8][2] = {{1, -2}, {2, -1}, {2, 1}, {1, 2}, {-1, 2}, {-2, 1},
    {-2, -1}, {-1, -2}};
15. int T_T, Dep, stx, sty;
16. char cur[N][N];
17.
18. inline bool check(const int &x, const int &y) {
19.     return x >= 0 && x <= 4 && y >= 0 && y <= 4;
20. }
21.
22. inline int H() {
23.     int res = -1;
24.     for (int i = 0; i <= 4; ++i)
25.         for (int j = 0; j <= 4; ++j)
26.             if (cur[i][j] != Des[i][j]) ++res;
27.     return res;
28. }
29.
30. inline bool Dfs(const int &step, const int &x, const int &y) {
31.     if (step > Dep) {
32.         if (H() == -1) return true;
33.         return false;
34.     }
35.     if (H() + step - 1 > Dep) return false;
36.     for (int i = 0; i <= 7; ++i) {
37.         int dx = x + Move[i][0], dy = y + Move[i][1];
38.         if (check(dx, dy)) {
39.             swap(cur[x][y], cur[dx][dy]);
40.             if (Dfs(step + 1, dx, dy)) return true;
41.             swap(cur[x][y], cur[dx][dy]);
42.         }
43.     }
44.     return false;

```

```
45. }
46.
47. int main() {
48.     scanf("%d", &T_T);
49.     while (T_T--) {
50.         for (int i = 0; i <= 4; ++i)
51.             scanf("%s", cur[i]);
52.         for (int i = 0; i <= 4; ++i)
53.             for (int j = 0; j <= 4; ++j)
54.                 if (cur[i][j] == '*') {
55.                     stx = i, sty = j;
56.                     break;
57.                 }
58.         for (Dep = 0; Dep <= 15; ++Dep)
59.             if (Dfs(1, stx, sty)) break;
60.         if (Dep > 15) puts("-1");
61.         else printf("%d\n", Dep);
62.     }
63.     return 0;
64. }
```

## 练习题

POJ 2286 ; ZOJ 1937

# 折半搜索

## 一、介绍

折半搜索，也称 **meet in the middle** (MITM)，它是一种常用的搜索优化技巧，有部分的搜索问题可以使用这个技巧大幅优化时间复杂度。

它是一种双向搜索，就如它的字面意思，“在中间相遇”。

它的核心思想是平衡搜索中的深度，使得能从初始状态与目标状态同时出发进行搜索。

## 二、问题模型

### 有向图模型

考虑这样一个问题：给定一张有向图  $G$ ，图中所有边的长度均为 1，现在求从图中的点  $A$  到点  $B$  之间有多少条长度为  $L$  的不同的路径。两条路径不同当且仅当某一步它们所走的边不相同。我们设图中点的最大出度为常数  $D$ 。

朴素的想法是我们从  $A$  开始 DFS 走  $L$  步，若最后停在了  $B$  点则我们可以记录进答案。这个做法的复杂度是  $O(D^L)$ 。

我们换个角度考虑，从点  $A$  到点  $B$  长度为  $L$  的路径，可以看做是点  $A$  到某点  $u$  长度为  $\frac{L}{2}$  的路径加上点  $u$  到点  $B$  长度为  $\frac{L}{2}$  的路径，也就是原图每条边方向反向后得到的反向图中，点  $B$  到点  $u$  长度为  $\frac{L}{2}$  的路径。相当于我们从点  $A$  正向走，从点  $B$  反向走，然后当它们都恰好走了  $\frac{L}{2}$  的长度，且它们在“中间相遇”时，我们就可以说，我们找到了一条在原图中从点  $A$  到点  $B$  长度为  $L$  的路径。

从点  $A$  正向 DFS  $\frac{L}{2}$  步，对每个点记  $P_u$  表示点  $A$  正向走到它的不同路径数。

从点  $B$  反向 DFS  $\frac{L}{2}$  步，对每个点记  $Q_u$  表示点  $B$  反向走到它的不同路径数。

根据加法原理与乘法原理以及上面的分析我们可知最后的答案为：

$$\sum_u P_u \cdot Q_u$$

容易发现，这个做法的复杂度就降为了  $O(D^{\frac{L}{2}})$ 。

这就是常用折半搜索的第一种模型，即有向图模型，从这个模型的做法我们也能看出，折半搜索是一种双向搜索。

## 方程模型

考虑这样一个问题，给定一个整数的集合  $S$ ，求有多少有序六元组  $(a, b, c, d, e, f)$  满足  $a, b, c, d, e, f \in S, d \neq 0$  且  $\frac{ab+c}{d} - e = f$ ， $|S| \leq 100$ 。

考虑朴素的算法是  $O(|S|^6)$  枚举，并按上述式子计算，判定是否符合条件。

我们将原式移项为  $ab + c = (e + f) \times d$ ，此时方程两边都有 3 个元素，因此我们考虑先用  $O(|S|^3)$  的时间枚举左边的三个元素，算出结果，再用  $O(|S|^3)$  的时间枚举右边的三个元素，也算出对应的结果，那么我们现在的问题是给出两个多重数集，问有多少个数同时在两个多重数集中出现了。这个问题可以使用哈希表来解决。这里我们就简单地认为哈希表插入与查询的复杂度均为  $O(1)$ 。那么这个算法的总时空复杂度均为  $O(|S|^3)$ 。

这就是折半搜索第二种常用的模型，方程模型，其核心就是通过移项，将方程两边的变量数进行平衡，之后再暴力搜索每一边，所得到的结果利用数据结构进行存储查询，或是利用其他的一些算法进行合并，从而优化复杂度。

## 三、算法总结

我们注意到不管是有向图模型还是方程模型，朴素算法都是一个指数级算法，因此搜索的深度越大，算法越慢，而折半搜索正是通过将路径拆开、将方程移项等方式，将搜索的深度减半，并利用数据结构等其他工具来合并两个搜索所得到的信息，从而优化搜索。这也正是算法的关键与核心：将问题分为两部分，分别进行搜索，通过优化合并过程来减少运算量，降低复杂度。

通常折半搜索能将时间复杂度从  $O(D^L)$  变为  $O(D^{\frac{L}{2}} \times \text{合并复杂度})$ 。尽管时间复杂度仍然是指数级别，但指数减小一半，在一些问题中就能使得我们在时限内出解。并且该算法实现简单，算法使用的搜索过程与朴素算法无异，合并时也只需要用到哈希表等简单的数据结构。

## 练习题

POJ 2549 ; CF 585D ; POJ 3977

上述内容参考：

1.挑战程序设计竞赛（第二版）

2.2013 年国家集训队论文，搜索问题中的 meet in the middle 技巧，乔明达