浅谈部分搜索+高效算法在搜索问题中的应用

浙江省杭州第十四中学　楼天城

摘要：

本文从有位置限制的匹配问题的搜索谈起，通过对题目 Milk Bottle Data 的分析，提出了深度优先搜索的一种非常规搜索——部分搜索+高效算法。然后通过部分搜索在 Triangle Construction 和智破连环阵两题中的应用，探讨了部分搜索方法通用的主要优化方法，并从此方法本质分析其高效的原因所在和应用需要满足的要求和限制。

关键字：

部分搜索、高效算法

正文：

很多题目，如果我们可以建立数学模型，应该尽量用解析法来处理，因为简单的模型更清晰地反映了事物之间的关系。

但是，并不是所有的题目都可以建立简单的数学模型。我们这时必须使用搜索的方法，也就是枚举所有可能情况来寻找可行解或最优解。

由于搜索建立在枚举之上，所以搜索常常和低效是分不开的。有时搜索的运算量实在太大，实在是一件痛苦的事情。

于是我们需要利用很多技巧来提高效率，可行性剪枝，最优性剪枝和调整搜索顺序等方法都很有用，在它们的帮助下，我们可以大大提高搜索的效率。

而有些题目，这些常规的优化方法很难有用武之地。这是我们必须使用一些非常规的搜索方法。本文中我们将讨论非常规搜索中的一种——部分搜索+高效算法。

引题：

N 个物品与 N 个位置，给定每个物品的可能放的位置集合，要求寻找一一对应的关系，但还给出物品位置之间的限制（例如：如果 1 放在 3 则 2 不能放在 1），求一组可行解，或给每一种对应关系一个权，求满足条件的最优解。

由于事物之间的限制关系非常复杂，很难建立简单的二分图关系，或者用网络流来解决。

面对这一系列类似的问题，我们一般只有搜索，如何搜索又如何优化呢？

简单分析：

如果我们枚举每一个物品的位置，然后判断，这样的时间复杂度为 O(N!)。好像似乎也只能这样。

进一步分析：

我们看一个例子，N=6：

其它限制有 4 条(a,b,c,d)表示如果 a 放在 b 则 c 不能放在 d

1 3 5 6

2 2 5 3

3 1 4 1

3 2 6 2

后来我们发现，如果我们一旦确定了 3 和 5 的位置，其它 4 个物品的位置之间已经没有限制关系了，这样其它 4 个物品的位置可以通过匹配来解决。

这时我们可以发现一个新的搜索模式：部分搜索。

部分搜索：搜索一部分变量，使得余下的变量之间的关系简化，然后通过一些高效算法（一般有匹配、解方程、贪心、动态规划等）完成余下问题。

就本题而言：先搜索一定数量（而不是全部）的物品的位置，使问题内物品的关系简化为二分图关系，用二分图匹配来解决余下的物品。

本质：

其实，例如上面的例子，如果我们先知道了 3 和 5 的位置后，不用匹配，其实我们是在用搜索来求匹配，效率当然不会高。

通过部分搜索为其它高效算法提供条件（例如上面的例子创造二分图关系），而其它高效算法代替搜索，高效地完成余下的任务。

部分搜索的方法充分发挥了搜索和其它高效算法的优势。搜索的优势在于应用性广，可以克服复杂的情况，其他高效算法的优势在于效率高。两者相互促进，同时也弥补对方的不足。这也是这个方法的成功的关键。

部分搜索+其它高效算法已经在很多题目中得到了应用。我们通过几个例子来探讨这种搜索方法的应用和优化技巧。

先看一个应用的例子。

Milk Bottle Data **(ACM/ICPC Asia Regional Shanghai 1996)**

【问题描述】

一个被分为**N*N**个网格的盒子，每一格有可能包含一瓶牛奶或者什么都没有。史密斯先生对每行从左到右记下牛奶的情况，同样对每列从上到下记下牛奶的情况。每一条记录包含**N**的数字，**0**表示没有牛奶，**1**表示有牛奶。不幸的是，**2*N**条记录的顺序被打乱了，有些数字也模糊不清。

现在史密斯先生请你恢复原来盒子的牛奶情况。

输入：

第一行：一个整数**N**，然后的**2N**行，每行有**N**个数字，**0**表示一定没有牛奶，**1**表示一定有牛奶，**2**表示不能确定。

样例：

input

5
01210
21120
21001
12110
12101
12101
00011
22222
11001
10010

output

9 8 6 2 7

4 1 0 1 1 0
10 1 0 0 1 0
1 0 1 1 1 0
3 0 1 0 0 1
5 1 1 1 0 1

数据范围：**1<=N<=10**

初步分析：

行列之间的限制关系非常复杂，很难找到多项式算法。（网络流！？）

可以用**(2N)!**的深度优先搜索，依次枚举每行和每列的记录编号，然后判断是否产生了矛盾。

判断方法：

设：第 **i** 条记录的第 **j** 个数字为 **A[i,j]**。

如果第 **a** 行选择第 **x** 条记录，第 **b** 列选择第 **y** 条记录，那么矛盾的条件就是：

((A[x,b]<>2) and (A[y,a]<>2) and (A[x,b]<>A[y,a]))

性能分析（1）：

因为 **N<=10**，**(20)!*10** 高达 **2432902008176640000**。

试一试几个基本的优化：

可行性剪枝：除了提前判断，也没有什么别的优化。

最优性剪枝：不是最优性问题。

调整搜索顺序：如果每次搜索可能性最少的，效果还是不错的。但是最坏情况需要计算**(20)!**的运算量，实在不能说是个好的可行算法。

部分搜索＋匹配算法

我们可以发现行与行之间和列与列之间不存在约束关系，所有的约束关系都在行列之间。我们可不可以只搜索行呢？答案是肯定的。

如果我们已经知道了每行的值，列与列之间已经没有约束关系。我们当然可以用匹配来求出一组可行解。

建图方法：

二分图左边 **N** 个点表示 **N** 个没有被选择的 **N** 条记录，右边 **N** 个点表示 **N** 列。如果将左边第 **i** 个结点对应编号的记录放入第 **j** 列，在第 **j** 列的每一行都不会产生矛盾，就在左边第 **i** 个结点和右边第 **j** 结点之间添一条边。

二分图匹配可以在 **$O(N^3)$**的时间内解决，比盲目搜索的效果好很多。

性能分析（2）：

这样 **N=10** 时的运算量为 **P(20,10)*(20*10*10)=13408850145600000**。效率与简单搜索相比已有很大的提高。因为只要找到一组解就可以结束程序，所以可以很快地通过所有测试数据。

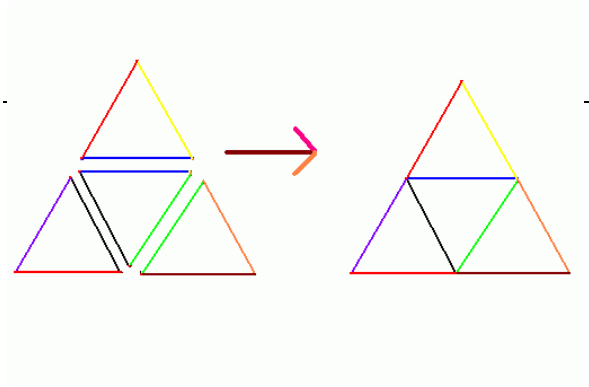
总结：

我们可以这样比较两个算法，如果普通的搜索是先搜索行再搜索列，那么在搜索完行以后，它是在用搜索来找匹配。搜索的复杂度为 **O(N!)**，而匹配的复杂度为 **$O(N^3)$**。搜索的效率不言而喻**.**

本搜索算法依然可以使用几个常规优化。特殊的搜索方法和常规的优化还是不矛盾的。

本算法还有很多独特优化，我们结合两个例子讲起。

例一：Triangle Construction(ZJU Monthly Contest)



【问题描述】

一个边长为 N 的等边三角形可以被分为 N*N 个边长为 1 的小的等边三角形，如图：

我们注意到相邻的两条边必须是同一种颜色，现在给出这 n*n 个三角形的三条边的颜色，请你判断是否可以用这 n*n 个三角形构成一个边长为 n 的等边三角形。

这里只考虑 n=4 的情况。

分析：

直接搜索运算量达 16!=20922789888000，实在太高了。

其实，这个问题也就是一个 16 个物体的有位置限制的匹配问题。与引题的区别就是：位置之间的限制是确定的，我们必须充分利用这一特点。

使用**部分搜索＋匹配**的算法，我们如何搜索呢？我们当然应该选择效率最高的方法。

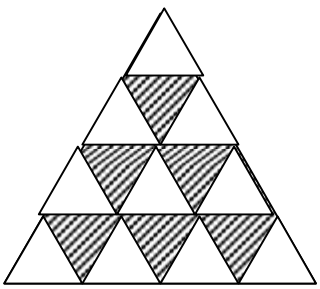
假设枚举的三角形的数量为 T，则运算量为 P(16,T)*256*(16-T)。

T	运算量	T	运算量
0	4096	9	7439214182400
1	61440	10	44635285094400
2	860160	11	223176425472000
3	11182080	12	892705701888000
4	134184960	13	2678117105664000
5	1476034560	14	5356234211328000
6	14760345600	15	5356234211328000
7	132843110400	16	?
8	1062744883200		

结论：运算量随 T 的增加而递增。

此类问题，一般情况下，枚举的变量数量越少越好。

优化：选择好的枚举变量，使枚举的变量数量尽可能少。



对于此题，我们可以枚举阴影部分的 6 块，其它 10 块都可以建立简单的二分图关系。

运算量为 P(16,6)*256*10=14760345600。加上常规的优化效率有了很大的提高。

总结：

搜索变量的选择：一般以搜索变量的数目尽可能少为原则，因为高效算法的效率是多项式级别的，但是搜索是 NP 的。

选择方法：有时可以通过分析，例如上述两题；但有的题目需要程序来确定搜索的变量。

例二：智破连环阵(NOI2003)（一个部分搜索法优化的经典例子。）

【问题描述】

B 国在耗资百亿元之后终于研制出了新式武器——连环阵(Zenith Protected Linked Hybrid Zone)，并声称这是一种无敌的自发性智能武器。但 A 国经侦察发现，连环阵其实是由 M 个独立武器组成的。这 M 个武器编号为 1，2，…，M。每件武器有两种状态：无敌自卫状态和攻击状态。最初，1 号武器处于攻击状态，其他武器都处在无敌自卫状态。以后，一旦第 I(1 ≤ i<M）号武器被消灭，1 秒钟以后第 i+1 号武器就自动从无敌自卫状态变成攻击状态。当第 M 号武器被消灭以后，这个造价昂贵的连环阵就被彻底摧毁了。

为了打败 B 国，A 国军事部长打算用最廉价的武器——炸弹来消灭连环阵。经过长时间的精密探测，A 国的军事家们掌握了连环阵中 M 个武器的平面坐标，然后依此选择了 n 个点，并在这些点上安放了特殊的定时炸弹。这 n 个炸弹编号为 1，2，…，n。每个炸弹的作用半径均为 k，且会持续爆炸 5 分钟。在这 5 分钟内，每枚炸弹都可以在瞬间消灭离它直线距离不超过 k 的、处在攻击状态的 B 国武器。和连环阵类似，最初 a_1 号炸弹持续引爆 5 分钟时间，然后 a_2 号炸弹持续引爆 5 分钟时间，接着 a_3 号炸弹引爆……以此类推，直到连环阵被摧毁。在每个炸弹爆炸的时候，其它尚未引爆的炸弹都处于地下隐蔽处，不会被己方的炸弹摧毁。

显然，选好 a_1、a_2、a_3...十分重要。好的序列可以在仅使用较少炸弹的情况下就能将连环阵摧毁；坏的序列可能在使用完所有炸弹后仍无法将连环阵摧毁。现在，请你决定一个序列 a_1、a_2、a_3…使得在第 a_x 号炸弹引爆的时间内连环阵被摧毁。这里的 x 应当尽量小。

【输入文件】

输入文件 zplhz.in 第一行包含三个整数：M、n 和 k（1 ≤ M，n≤ 100，1≤ k≤ 1000），分别表示 B 国连环阵由 M 个武器组成，A 国有 n 个炸弹可以使用，炸弹攻击范围为 k。以下 M 行，每行由一对整数 x_i，y_i（0≤ x_i，y_i ≤ 10000）组成，表示第 i（1≤ i≤ M）号武器的平面坐标。再接下来 n 行，每行由一对整数 u_i，v_i(0 ≤ u_i，v_i≤ 10000)组成，表示第 i（1≤ i≤ n）号炸弹的平面坐标。输入数据保证无误和有解。

测试数据中的 x_i、y_i、u_i、v_i 是随机生成的。

【输出文件】

输出文件 zplhz.out 的第一行包含一个整数 x，表示实际使用的炸弹数。第二行包括 x 个整数，依次表示 a_1，a_2，…，a_x。

初步分析：

A国炸弹I可以炸到B国武器J的条件：$(u[i]-x[j])^2+(v[i]-y[j])^2<=r^2$

结论：很难找到求最优解的多项式算法。

面对此类问题，一般只有搜索策略。

进一步分析：

每一颗炸弹必定炸掉B国武器中编号连续的一段。

5分钟只是表明每一颗炸弹可以炸掉任意多个编号连续的B国武器。

部分搜索：

此题使用部分搜索的算法需要一些转化：如果已经将B国武器根据编号分为x段，$[S_i,T_i]$ $(S_1=1,T_i>=S_i,T_i+1=S_{i+1})$。然后判断是否可以从A国的N颗炸弹中选出x颗，分别可以炸掉其中的一段。

其实我们把搜索分为了两部分，先通过搜索将B国武器根据编号分为x段，再

通过搜索判断是否可以从A国的N颗炸弹中选出x颗，分别可以炸掉其中的一段。其实第二部分可以用匹配来解决。

C[S][T][I] 表示A国炸弹I是否可以炸到B国武器S,S+1..T-1,T。

$C[S][S][I]=((u[I]-x[S])^2+(v[I]-y[S])^2<=R^2)$

C[S][T][I]=C[S][T-1][I] & C[T][T][I] (S<T)

求C的时间复杂度为$O(N^3)$。

建图：左边x个点，表示B国武器根据编号分为的x段，右边N个点，表示A国的N颗炸弹。左边第i个点到右边第j个点有边的条件即：C[Si][Ti][j]。

搜索的任务就是将B国武器根据编号划分为若干段+二分图匹配判断。

性能分析（1）：

搜索的基本框架已经建立，虽然数据是随机生成的，但是M个B国武器的划分方案还是非常多的，有时可能高达2^m。时间上很难承受，如果使用卡时，正确性受到影响，效果不会很好。

只有4个数据可以在时限内出解，另外6个如果卡时，有1个也可以得到最优解。

优化：

优化可以通过可行性和最优性两方面分析。

优化一（最优性）：如果A国炸弹可以重复使用，设：

Dist[i]=炸掉B国武器i－m的最少使用炸弹数。可以用动态规划计算Dist值，状态转移方程如下：

Dist[m+1]=0。

Dist[i]=Min(Dist[j]+1 | Can[i][j-1][k](1<=k<=n)) (1<=i<=n) (i<j<=n+1)

求Dist的时间复杂度为$O(N^3)$。

从而产生了一个最优性剪枝条件：

if 当前已经使用的炸弹数+Dist[当前已经炸掉的B国武器数+1]>=当前找到的最优解 then 剪枝；

优化二（可行性）：

此搜索方法一般都可以用两个效果很好的可行性优化：

(1)提前判断是否可以匹配成功，避免多余的搜索。

(2)每次匹配可以从以前的匹配开始扩展，不需要重新开始。

如果当前的划分方法已经无法匹配成功，就没有搜索下去的必要了，只要每搜索新的一段时立即通过匹配判断即可。

每次求匹配只要从原来的基础上扩展就可以了。

通过上述两个优化，程序的效率有了很大的提高。

性能分析（2）：

虽然通过上述两个优化，程序的效率较原来的搜索有了很大的提高。10个测试数据中有8个可以在时限内出解，另外2个如果卡时，有1个也可以得到最优解。

进一步优化：

优化二虽然排除了许多不必要的划分，但是在判断时浪费了不少时间。

因此，在枚举划分长度时，可以通过以前的划分和匹配情况（被匹配的边），用$O(n^2)$的时间复杂度的宽度优先搜索计算出下一个划分的最大长度maxL，显然下一个划分的长度在[1，maxL]都一定可以找到可行的匹配。这样既节省了判断的时间，又可以使每次划分长度从长到短枚举，使程序尽快逼近最优解，同时增

强了剪枝条件一的效果。

这一部分的实现，首先需要求**MaxT**。

MaxT[I][S]=炸弹**i**，从**S**开始炸，可以炸到的最大编号。

如果，炸弹**i**炸不到**S**，则**MaxT[I][S]=S-1**。

求 **MaxT[I][S]**可以用动态规划的方法解决。状态转移方程为：

MaxT[I][S]= 炸弹 **i** 炸不到 **S**　**S-1**

炸弹 **i** 炸得到 **S**　**MaxT[i][S+1]**

MaxT[i][m+1]=m

求**MaxT**的时间复杂度为$O(N^2)$。

具体实现方法，考虑二分图右边的**n**个结点（**n**颗炸弹），如果结点**i**没有匹配，**i**被认为可以使用。对于一个已经匹配的结点**i**，如果从任何一个没有匹配的结点出发存在一条到达**i**，而且**i**为外点的交错路，**i**也被认为可以使用。

计算所有从没有匹配点出发的交错路（没有匹配点**i**出发的交错路没有被匹配点**i**一定为外点）所能到达的匹配的结点，只要从每一个没有匹配的结点出发，宽度优先搜索，只要$O(N^2)$的时间。注意判断重复（如果一个已经匹配的结点已经被确定为可以使用，那么不需要对它再扩展一次，因为当把这个已经匹配的结点确定为可以使用的结点的时候，已经从这个结点扩展过，如果再扩展必将产生无谓的重复）

所以**MaxL=Max(MaxT[I][S] | i**可以使用**)**；

性能分析（3）：

通过以上的优化，所有数据都是瞬间出解，并且所有结果都是最优解。

甚至对 **n=200** 的随机数据，也可以在瞬间出解，可见程序的效率有了很大的提高。

精益求精：

另外，还有两个优化，但是有时效果不好，但也值得一提。

优化三：分支定界。这样可以增强剪枝条件一的效果，但是当最优解与**Dist[1]**相差比较远的时候，会浪费一定的时间。

优化四：优化一中 **Dist[i]**的值有时并不是最优的，通过测试，发现如果**Dist[i]**的值与最优值相差 **1**，特别是当 **i** 小的时候，程序的速度都会有明显的影响。所以，可以通过同样的搜索来计算 **Dist[i]**，（本题的答案就是 **Dist[1]**）。这样做可以增强剪枝条件一的效果，但是同时对每个 **i** 都要搜索也浪费了一定的时间。

程序结果比较：

	1	2	3	4	5	6	7	8	9	10
最简单的搜索	0.00	0.00	0.50	Time Over	0.65	Time Over	Time Over	Time Over	Time Over	Time Over
优化的搜索	0.01	0.01	0.10	Time Over	0.50	0.80	Time Over	0.55	0.30	0.80
进一步优化的搜索	0.01	0.01	0.02	0.03	0.00	0.02	0.02	0.01	0.02	0.02

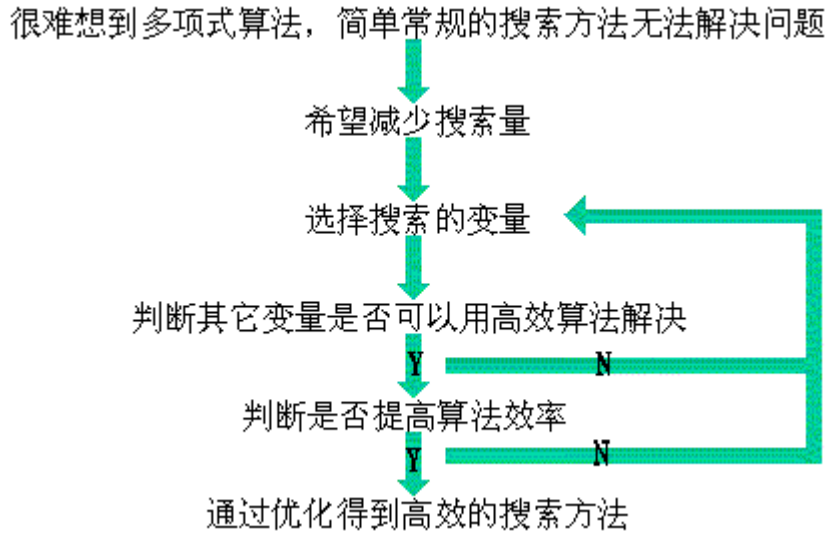
此搜索方法一般都可以用两个效果很好的可行性优化：

(1)提前判断是否可以成功，避免多余的搜索。

(2)每次判断尽量多利用以前的判断结果。

总结：

本文中的几个例子都可以应用部分搜索的方法高效解决，它们在思想上有着明显的相同点。一般的思维过程如下：



一般的优化包括：（1）选择好的枚举变量，使枚举的变量数量尽可能少。（2）尽可能充分利用以前的结果，提前判断，避免多余的搜索。

部分搜索同样可以和解方程、贪心、动态规划等高效算法结合。

部分搜索＋高效算法体现了搜索与其他方法的有机结合，充分发挥两者的长处，相互弥补对方的不足，这就是其高效的主要原因所在。因此，在搜索问题中灵活地应用部分搜索的方法，往往可以创造出奇效。

值得注意的是，部分搜索来解决搜索问题作为一种非常规的搜索方法。虽然在本文的例子中，部分搜索有着很多的过人之处，但是并不能认为常规方法一定不如非常规方法。大多数的搜索问题还是适合用常规的搜索方法的，所以只有充分把握部分搜索的特点，使之与常规的搜索融会贯通，才能真正得到高效的搜索算法。

参考文献：

ACM/ICPC Asia Regional Shanghai 1996（Milk Bottle Data)
ZJU Monthly Contest(Triangle Construction)
NOI2003(智破连环阵)

感谢：

Zhejiang University Online Judge 提供(Milk Bottle Data)和(Triangle Construction)的题目描述。

附录：

智破连环阵参考程序：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
const int maxm=100+2;
const int maxn=100+2;

int n,m,dist[maxm],MaxT[maxm][maxn];
bool reachable[maxm][maxn],*can[maxm][maxm];
/*
m=B 国武器数。
n=A 国炸弹数。
dist[i]=如果 A 国炸弹可以重复使用，炸掉 B 国武器 I~m 的最少使用炸弹数。
MaxT[s][i]=炸弹 I，从 S 开始炸，可以炸到的最大编号，
          如果炸弹 I 炸不到 S，则 MaxT[I][S]=S-1。
reachable[i][j]=A 国炸弹 i 是否可以炸到 B 国炸弹 j。
can[s][t][i]=表示 A 国炸弹 I 是否可以炸到 B 国武器 S,S+1..T-1,T。
*/
int answer,bestv[maxn];
/*
answer=最少需要的 A 国炸弹数。
bestv=记录取最优解 A 国炸弹的使用序列。
*/
int a[maxm],b[maxn];
bool vis[maxn],*g[maxm];
/*
a[i],b[i]用于匹配，分别记录左（右）第 i 个点的匹配边的另一个点的编号，如果没有匹配则为 0。
vis[i]用于匹配和宽度优先搜索时判重。
g[i][j]=左边第 i 个点到右边第 j 个点是否有边。
*/
void init()
{
    //读入数据并计算 reachable。
    int x1[maxm],y1[maxm],x2[maxn],y2[maxn],i,j,R;
    scanf("%d%d%d",&m,&n,&R);
    for (i=1;i<=m;i++)
        scanf("%d%d",&x1[i],&y1[i]);
    for (i=1;i<=n;i++)
        scanf("%d%d",&x2[i],&y2[i]);
    for (i=1;i<=m;i++)
        for (j=1;j<=n;j++)

    reachable[i][j]=((x1[i]-x2[j])*(x1[i]-x2[j])+(y1[i]-y2[j])*(y1[i]
-y2[j])<=R*R);
}
void preprocess()
```

```
{
    //初始化，计算 can,MaxT。
    int s,t,i;
    for (i=1;i<=m;i++)
        g[i]=new bool[maxn];
    for (s=1;s<=m;s++)
        for (t=s;t<=m;t++)
            can[s][t]=new bool[maxn];
    for (s=1;s<=m;s++)
    {
        for (i=1;i<=n;i++)
            can[s][s][i]=reachable[s][i];
        for (t=s+1;t<=m;t++)
            for (i=1;i<=n;i++)
                can[s][t][i]=can[s][t-1][i] && reachable[t][i];
        for (i=1;i<=n;i++)
        {
            MaxT[s][i]=s-1;
            for (t=s;t<=m;t++)
                if (can[s][t][i])
                    MaxT[s][i]=t;
        }
    }
    //计算 dist
    dist[m+1]=0;
    for (s=m;s>=1;s--)
    {
        t=s-1;
        for (i=1;i<=n;i++)
            if (MaxT[s][i]>t)
                t=MaxT[s][i];
        dist[s]=1+dist[t+1];
    }
}
bool find(int v)
{
    //匈牙利算法找可增广路。
    for (int i=1;i<=n;i++)
        if (g[v][i] && !vis[i])
        {
            vis[i]=true;
            if (b[i]==0 || find(b[i]))
            {
                a[v]=i;
```

```
                b[i]=v;
                return true;
            }
        }
    return false;
}
void search(int used,int s)
{
    //状态：已经使用了 used 个 A 国炸弹，编号在 s 之前的 B 国武器都已经炸毁。

    if (used+dist[s]>=answer)//优化一：最优性剪枝
        return;
    if (s==m+1)
    {
        //如果 B 国武器已经全部炸毁，更新最优解，回溯。
        answer=used;
        memcpy(bestv,a,sizeof(a));
        return;
    }
    int t,maxL,tempa[maxm],tempb[maxn],op,cl,queue[maxn],k,i;
    //宽度优先搜索计算出下一个划分的最大长度 maxL
    memset(vis,false,sizeof(vis));
    maxL=s-1;
    op=cl=0;
    for (i=1;i<=n;i++)
        if (b[i]==0)
        {
            vis[i]=true;
            queue[++op]=i;
        }
    while (cl<op)
    {
        k=queue[++cl];
        if (MaxT[s][k]>maxL)
            maxL=MaxT[s][k];
        for (i=1;i<=used;i++)
            if (g[i][k] && !vis[a[i]])
            {
                vis[a[i]]=true;
                queue[++op]=a[i];
            }
    }
    if (maxL==s-1)
        return;
```

```
    used++;
    memcpy(tempa,a,sizeof(a));
    memcpy(tempb,b,sizeof(b));
    memset(vis,false,sizeof(vis));
    g[used]=can[s][maxL];
    //扩展交错路。
    find(used);
    //从大到小枚举下一段的长度。
    for (t=maxL;t>=s;t--)
    {
        g[used]=can[s][t];
        search(used,t+1);
    }
    memcpy(a,tempa,sizeof(a));
    memcpy(b,tempb,sizeof(b));
}
void out()
{
    //输出结果。
    printf("%d\n",answer);
    for (int i=1;i<=answer;i++)
        printf("%d ",bestv[i]);
    printf("\n");
}
int main()
{
    freopen("zplhz.in","r",stdin);
    freopen("zplhz.out","w",stdout);
    init();
    preprocess();
    answer=1000000000;
    memset(a,0,sizeof(a));
    memset(b,0,sizeof(b));
    search(0,1);
    out();
    return 0;
}
```