Dancing Links 中文版（DLXcn）

Donald E.Knuth, Stanford University

翻译 武汉武钢三中 吴豪
更正 排版 上海交通大学 隋清宇（sqybi）

目录

正文

我写这篇论文的目的，是觉得这个简单的程序技巧理应得到广泛认可。假设 x 指向双向链的一个节点；L[x]和 R[x]分别表示 x 的前驱节点和后继节点。每个程序员都知道如下操作：

$$L[R[x]] \leftarrow L[x],\ R[L[x]] \leftarrow R[x] \qquad (1)$$

是将 x 从链表删除的操作；但是只有少数程序员意识到如下操作：

$$L[R[x]] \leftarrow x,\ R[L[x]] \leftarrow x \qquad (2)$$

是把 x 重新链接到双向链中。

当然，指出这种操作以后，这个结果是显然的。但是，当我真正认识到操作（2）的作用以后，我突然感到了定义“啊哈”这个词语时候的感觉，因为，L[x]和 R[x]的值在 x 从链表中删除以后早已没有了它原来的语义。确实，一个精心设计的程序在 x 被删除后会通过把 L[x],R[x]赋值为 x 或者赋值为空值（null）来清理掉这些不用的数据结构。而让一个链外的对象指向链本身有时具有潜在的危险性。例如，指针就可以干扰垃圾回收机制的运作。

那么是什么关于操作（2）的研究促使我写一整篇论文来讨论这个问题呢？当 x 从链表删除以后；为什么还要把它放回链表中？嗯，我承认，数据结构的更新通常来说是永久性的。但是非永久性的更新也时常发生。例如，在一个交互性的程序中，用户很可能想撤销他所做的一个或一系列操作，恢复到先前的状态。另一个典型的应用是在*回溯程序*（*backtrack programs*）[16]里,回溯程序枚举约束集合里的所有解。回溯，也叫*深度优先搜索*（*depth-first search*），在之前的论文中曾经讨论到。

操作（2）的观点是 Hitotumatu 和 Noshita [22]于 1979 年提出的。他们提出 Dijkstra 提出的著名的解决 N 皇后问题 [6，第 72-82 页]的算法在使用了这个技巧后，程序的速度比不使用几乎快了 2 倍。

Floyd 关于回溯和非确定性算法 [11]之间关联的优雅论述中包含详细的数据结构更新与恢复的算法（谁能够提供这句话的准确翻译？——译者）。通常来说，回溯程序可以被认为是一种搜索，所要做的就是缩小这个任务需要搜索的范围，同时组织好用于控制搜索流程和决策的数据。对于多步的问题,解决问题的每一步操作，都将改变剩余需要解决的问题。

简单情况下，我们可以考虑维护一个栈，用来保存当前搜索树节点之前的所有相关状态信息，但是这个任务的拷贝动作需要耗时太多。因此，我们通常选用全局数据结构。这样无论搜索进行到何种程度，它都会保留相关状态信息，并且当搜索回溯的时候它都能恢复先前状态。

例如，Dijkstra 解决 n 皇后问题的递归算法将当前状态保存在三个全局布尔（Boolean）数组中，他们分别表示棋盘上的列和 2 条对角线；Hitotumatu 和 Noshita 的程序中使用双向链表来记录所有列和对角线上的可能性。当 Dijkstra 算法暂时放置一个皇后在棋盘上的时候，会把每个布尔数组里的一个数据从真改为假；回溯后又将这个数据改回真。Hitotumatu 和 Noshita 使用（1）去删除一列，使用（2）去恢复删除操作；这意味着他们可以不通过搜索便找到一个空列。程序通过这种方法记录下每个状态信息，这样替换和恢复节点使得 N 皇后问题的计算更加高效。

算法（2）的优雅之处就在于我们仅仅知道 x 的值就可以恢复（1）的操作。通常来说要恢复操作，需要我们记录下节点的左指针和它先前的值（请参阅 [11]或 [25]，268-284 页）。但是在这个实例中，我们只需要知道 x 的值，而回溯程序在做通常的操作时恰恰又很容易得到节点的值。

我们可以把（1）、（2）这对操作应用于涉及到大量操作的复杂数据结构的双向链上。这个删除元素的操作可以随时进行逆操作，因此它可以用来决定哪些元素需要被恢复（即用来恢复已经删除的元素——译者）。重建链表的恢复操作使得我们可以一直向后回溯到下一次向前递归为止。这个过程使得指针在数据结构内部被灵活运用，仿佛设计精巧的舞蹈动作。因此，我很愿意把（1）、（2）的这个技巧叫做*舞蹈链*（*Dancing Links*）。

精确覆盖问题。阐明 Dancing Links 威力的一种方法就是考虑一个能大致描述如下的一般问题：给定一个由 0 和 1 组成的矩阵，是否能找到一个行的集合，使得集合中每一列都恰好包含一个 1？例如，下面这个矩阵

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \tag{3}$$

就包含了这样一个集合（第 1，4，5 行）。我们把列想象成全集的一些元素，而行看作全集的一些子集；或者我们可以把行想象成全集的一些元素，而把列看作全集的一些子集；那么这个问题就是要求寻找一批元素，它们与每个子集恰好有一个交点。不管怎么说，这都是一个很难的问题，众所周知，当每行恰包含 3 个 1 时，这是个一个 NP-完全问题 [13，第 221 页]。自然，作为首选的算法就是回溯了。

Dana Scott 完成了第一个关于回溯算法的实验。1958 年，当他作为 Princeton University 普林斯顿大学 [34]的一名研究生时，在 Hale F. Trotter 的帮助下，他在 IAS "MANIAC" 机器上首次实现 12 片 5 格骨牌拼图问题（12 片 5 格骨牌拼图问题要求把 12 片骨牌放入正方形棋盘，并且中间留有 2x2 的空格）的回溯解法。他的程序首次产生了所摆放的可能性。例如，65 种解中的一种如图 1 所示（5 格骨牌是 n 格骨牌在 n=5 时的特例；见 [15]。Scott 或许从 Golomb 的论文 [14]和 Martin Gardner 的一些深入报告 [12]中得到了灵感。）

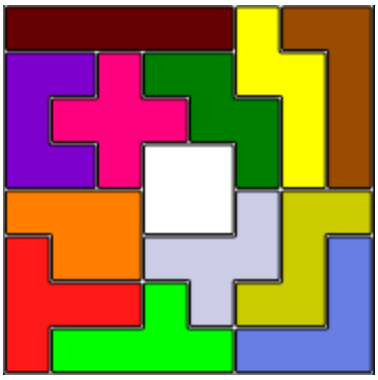


图 1 Scott 的 12 片 5 格骨牌拼图问题

这个问题是精确覆盖问题的一个特例。我们想象一个有 72 列的矩阵，其中 12 列是 12 个骨牌，剩下 60 列是六十个非中心部分的格子，构造出所有可能的行来代表在一块骨牌在棋盘上的放置方案；每行有一些‘1’，用来标识被覆盖的格子，5 个 1 标识一个骨牌放置的位置（恰有 1568 个这样的行）。依据 Golomb 对骨牌命名的介绍 [15，第 7 页]，我们将最前面的 12 列命名为 F I L P N T U V W X Y Z，并且我们可以用两个数字 ij 给矩阵中对应棋盘上第 i 行第 j 列格子的那一列命名。通过给出那些出现了‘1’的列的名字，可以很方便地表示每一行。例如，图 1 就是与下面 12 行的对应的精确覆盖。

I	11	12	13	14	15
N	16	26	27	37	47
L	17	18	28	38	48
U	21	22	31	41	42
X	23	32	33	34	43
W	24	25	35	36	46
P	51	52	53	62	63
F	56	64	65	66	75
Z	57	58	67	76	77
T	61	71	72	73	81
V	68	78	86	87	88
Y	74	82	83	84	85

解决精确覆盖问题。对于接下来的非确定性算法，由于我们没有想到更好的名字，我们将称之为 X 算法，它能够找到由特定的 01 矩阵 A 定义的精确覆盖问题的所有解。X 算法是实现试验——错误这一显而易见的方法的一段简单的语句（确实，一般来说，我想不到别的合理的方法来完成这个工作）。

如果 A 是空的，问题解决；成功终止。
否则，选择一个列 c（确定的）。
选择一个行 r，满足 A[r, c]=1 （不确定的）。
把 r 包含进部分解。

对于所有满足 A[r,j]=1 的 j，
　　从矩阵 A 中删除第 j 列；
　　对于所有满足 A[i,j]=1 的 i，
　　　　从矩阵 A 中删除第 i 行。
在不断减少的矩阵 A 上递归地重复上述算法。

对 r 不确定的选择意味着这个算法本质上把自身复制给许多独立的子算法；每个子算法继承了当前的矩阵 A，但在考虑不同行 r 的同时对其进行了删减。如果列 c 全部是 0，那么就不存在子算法而且这个过程会不成功地终止。很自然地，所有的子算法搭建了一棵搜索树，其根部就是初始问题，并且第 k 层的每个子算法对应 k 个选择的行。回溯就是前序遍历这棵树的过程，即“深度优先”。
这个程序中任意选择列 c 的体系规则都能找到所有解，但是有些规则运行起来比别的会好得多。例如，Scott [34]说他最初更倾向于先放第一张骨牌，然后再放第二张，依此类推；这就对应了在于之相符该精确覆盖问题中先选择 F 列，再选择 I 列，等等。但是他很快意识到这个方法将会变得无可救药的慢。有 192 种放置 F 的方法，对于每种又有 34 种放置 I 的方法。[24]中介绍的 Monte Carlo 计算法暗示了该方案的搜索树粗略估计会有 $2*10^{12}$ 个结点！相较之下，如果一开始选择 11 列（矩阵中对应棋盘上第 1 行第 1 列的那一列），并且大体上按照字典序选择第一个没有被覆盖的列，那么导出的搜索树仅有 9,015,751 个结点。一个更好的策略被 Scott [34]采用：他意识到 X 块本质上有 3 种不同的位置，即中心在 23，24 和 33。更进一步，如果 X 在 33 处，我们可以假定 P 块没有“翻转”，这么一来它就只能取 8 个方向中的 4 种。接着我们一次得到 65 种本质不同的解，那么全部解集有 8*65=520 种解，这些解通过旋转和对称很容易得到。X 和 P 的这些约束引导出了 3 个独立的问题，当按字典序选择列时,他们的搜索树分别：

有 103,005 个结点和 19 组解 (X 在 23 处)
有 106,232 个结点和 20 组解 (X 在 24 处)
有 126,636 个结点和 26 组解 (X 在 33 处,P 没有翻转)。

Golomb 和 Baurnert [16]建议，在每个回溯的过程中，选择能够导出最少分支的子问题，任何时候这都是可以被有效完成的。在精确覆盖问题中，这意味着我们希望每步都选择在当前 A 中包含 1 最少的列。幸运的是我们将看到 dancing links 技术让我们相当好地做到这一点；使用这个技术后，Scott 的骨牌问题的搜索树将分别仅有：

10,421 个结点 (X 在 23 处)
12,900 个结点 (X 在 24 处)
14,045 个结点 (X 在 33 处,P 没有翻转)。

舞蹈步骤。一个实现 X 算法的好方法就是将矩阵 A 中的每个 1 用一个有 5 个域 L[x]、R[x]、U[x]、D[x]、C[x]的*数据对象*（*data object*）x 来表示。矩阵的每行都是一个经由域 L 和 R(“左”和“右”)双向连接的环状链表；矩阵的每列是一个经由域 U 和 D(“上”和“下”)双向连接的环状链表。每个列链表还包含一个特殊的数据对象，称作它的*表头*（*list header*）。
这些表头是一个称作*列对象*（*column object*）的大型对象的一部分。每个列对

象 y 包含一个普通数据对象的 5 个域 L[y]、R[y]、U[y]、D[y]和 C[y]，外加两个域 S[y]（大小）和 N[y]（名字）；这里“大小”是一个列中 1 的个数，而“名字”则是用来标识输出答案的符号。每个数据对象的 C 域指向相应列头的列对象。表头的 L 和 R 连接着所有需要被覆盖的列。这个环状链表也包含一个特殊的列对象称作“根”，h，它相当于所有活动表头的主人。而且它不需要 U[h]、D[h]、C[h]、S[h]和 N[h]这几个域。

举个例子，（3）中的 0-1 矩阵将用这些数据对象来表示，就像图 2 展示的那样，我们给这些列命名为 A、B、C、D、E、F 和 G（这个图表在上下左右处“环绕扭曲”。C 的连线没有画出，因为他们会把图形弄乱；每个 C 域指向每列最顶端的元素）。

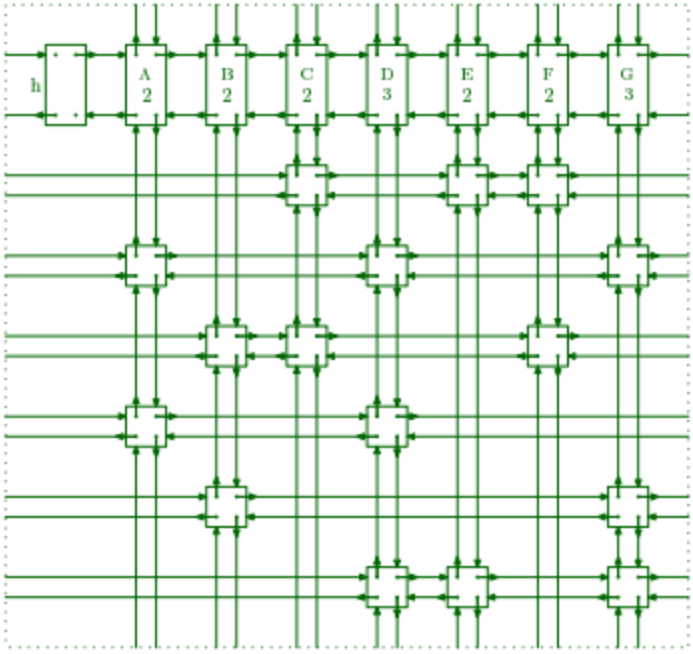


图 2 完全覆盖问题（3）的四方向连接表示法

我们寻找所有精确覆盖的不确定性算法现在可以定型为下面这个明析、确定的形式，即一个递归过程 search(k)，它一开始被调用时 k=0：

如果 R[h]=h ，打印当前的解（见下）并且返回。
否则选择一个列对象 c（见下）。
覆盖列 c（见下）。
对于每个 r←D[c]，D[D[c]]，……，当 r!=c，
    设置 O_k←r；
    对于每个 j←R[r]，R[R[r]]，……，当 j!=r，
        覆盖列 j（见下）；
    search(k+1)；
    设置 r←O_k 且 c←C[r]；
    对于每个 j←L[r]，L[L[r]]，……，当 j!=r，
        取消列 j 的覆盖（见下）。
取消列 c 的覆盖（见下）并且返回。

输出当前解的操作很简单：我们连续输出包含 O_0、O_1、……、O_{k-1} 的行,这里包含数据对象 O 的行可以通过输出 N[C[O]]、N[C[R[O]]]、N[C[R[R[O]]]]……来输出。

为了选择一个列对象 c,我们可以简单地设置 c<-R[h]；这是最左边没有覆盖的列。或者如果我们希望使分支因数达到最小，我们可以设置 s<-无穷大，那么接下来：

```
对于每个 j←R[h]，R[R[h]]，……，当 j!=h，
    如果 S[j]<s 设置 c←j 且 s←S[h]。
```

那么 c 就是包含 1 的序数最小的列(如果不用这种方法减少分支的话，S 域就没什么用了)。
覆盖列 c 的操作则更加有趣：把 c 从表头删除并且从其他列链表中去除 c 链表的所有行。

```
设置 L[R[c]]←L[c] 且 R[L[c]]←R[c]。
对于每个 i←D[c]，D[D[c]]，……，当 i!=c，
    对于每个 j←R[i]，R[R{i]]，……，当 j!=i，
        设置 U[D[j]]←U[j]，D[U[j]]←D[j]，
        并且设置 S[C[j]]←S[C[j]]-1。
```

操作（1），就是我在本文一开始提到的，在这里他被用来除去水平、竖直方向上的数据对象。
最后，我们到达了整个算法的尖端，即还原给定的列 c 的操作。这里就是链表舞蹈的过程：

```
对于每个 i←U[c]，U[U[c]]，……，当 j!=i，
    对于每个 j←L[i]，L[L[i]]，……，当 j!=i，
        设置 S[C[j]]←S[C[j]]+1，
        并且设置 U[D[j]]←j，D[U[j]]←j。
设置 L[R[c]]←c 且 R[L[c]]←c。
```

注意到还原操作正好与覆盖操作执行的顺序相反，我们利用操作（2）来取消操作（1）。（其实没必要严格限制“后执行的先取消”，由于 j 可以以任何顺序穿过第 i 行；但是从下往上取消对行的移除操作是非常重要的，因为我们是从上往下把这些行移除的。相似的，对于第 r 行从右往左取消列的移除操作也是十分重要的，因为我们是从左往右覆盖的。）

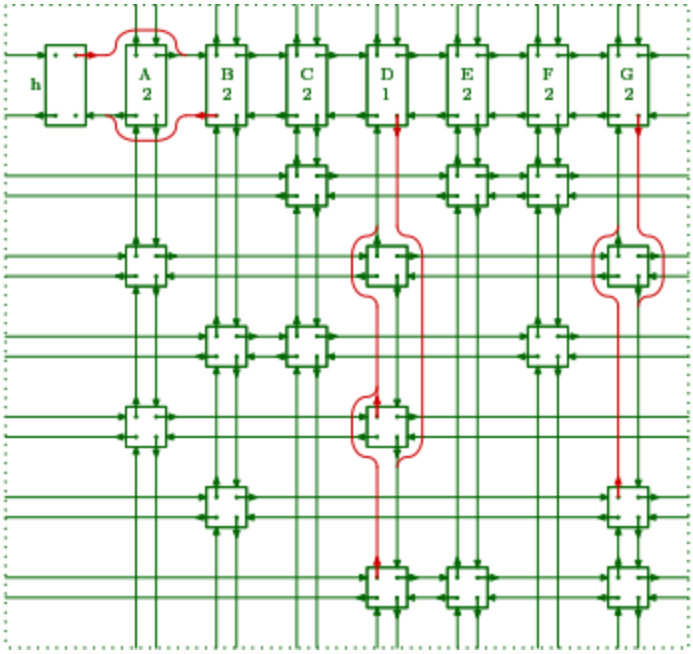


图 3 图 2 中第 A 列后面的链表被覆盖

考虑一下，例如，对图 2 表示的数据（3）执行 search(0)会发生什么。通过从其他列移除 A 的行来将其覆盖；那么现在整个结构就成了图 3 的样子。注意现在 D 列出现了不对称的链接：上面的元素首先被删除，所以它仍然指向初始的邻居，但是另一个被删除的元素指向了列头。

继续 search(0)，当 r 指向（A，D，G）这一行的 A 元素时，我们也覆盖 D 列和 G 列。图 4 展示了我们进入 search(1)时的状态，这个数据结构代表削减后的矩阵

$$\begin{matrix} \mathrm{B} & \mathrm{C} & \mathrm{E} & \mathrm{F} \end{matrix}$$
$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \qquad (4)$$

现在 search(1)将覆盖 B 列，而且 C 列将没有“1”。因此 search(2)将什么也找不到。接着 search(1)会找不到解并返回，图 4 的状态会恢复。外部的过程，search(0)，将把图 4 变回图 3，而且它会让 r 前进到（A，D）行的 A 元素处。

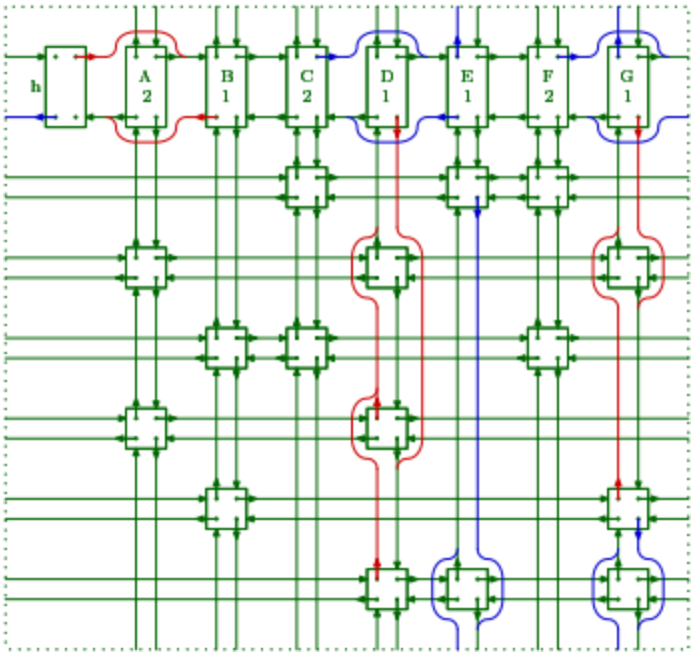


图 4 图 3 中 D 列和 G 列后的链被覆盖

很快就能找到解，并输出

A D
B G
C E F

如果在选择 c 的时候无视 S 域，会输出

A D
E F C
B G

如果每步选择最短的列。（每行输出的第一项是已经完成分支的列的名字）在一些例子上试验过这个算法的读者应该会明白我为什么给这篇论文选这个标题。

效率分析。当算法 X 用 Dancing Links 实现时，让我们称之为 DLX 算法。DLX 算法的运行时间本质上和它执行操作（1）来移除表中对象的次数是成比例的；这同时也是它执行操作(2)来还原对象的次数。我们把这个数量称作*更新*(*updates*)的次数。如果每步选择最短的列，则在对（3）求解的的过程中共做了 28 次更新：第 0 层更新 10 次，第 1 层更新 14 次，第 2 层更新 4 次。如果我们忽略启发条件 S，这个算法就在第 1 层更新 16 次，在第 2 层更新 7 次，总计 33 次。但是在后者的更新明显快些，因为 S[C[j]←S[C[j]]±1 这样的语句可以忽略；因此全部的运行时间会少些。当然，我们在给启发条件 S 的期望效果下一般结论前还需要对一些大规模的实例进行分析。

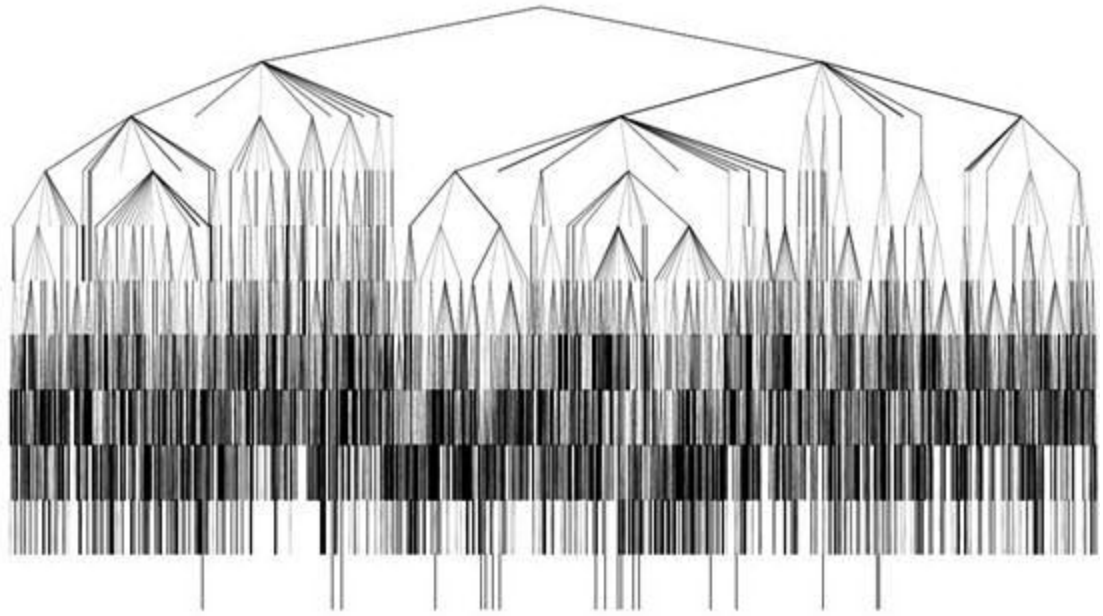


图 5 Scott 的 12 片 5 格骨牌拼图问题的搜索树

一个回溯程序通常把大部分时间用于搜索树的寥寥数层当中（参见 [24]）。例如，图 5 展示了对于 X=23 的 Dana Scott 的 12 片 5 格骨牌拼图问题使用启发条件 S 的搜索树。其轮廓如下：

层数	结点数	更新次数	每个结点的更新次数
0	1 (0%)	2,031 (0%)	2031.0
1	2 (0%)	1,676 (0%)	838.0
2	22 (0%)	28,492 (1%)	1295.1
3	77 (1%)	77,687 (2%)	1008.9
4	219 (2%)	152,957 (4%)	698.4
5	518 (5%)	367,939 (10%)	710.3
6	1,395 (13%)	853,788 (24%)	612.0
7	2,483 (24%)	941,265 (26%)	379.1
8	2,574 (25%)	740,523 (20%)	287.7
9	2,475 (24%)	418,334 (12%)	169.0
10	636 (6%)	32,205 (1%)	50.6
11	19 (0%)	826 (0%)	43.5
合计	10,421(100%)	3,617,723(100%)	347.2

（第 k 层的“更新次数”表示在第 k-1 层到第 k 层之间的计算过程中，元素从双向链表中被移除的次数。第 0 层的 2031 次更新对应着从表头中移除 X 列，并从

其他列中移除 2030/5=406 行；这些行和 X 在 23 处的放置相交迭。把数据列成表时做了一个小小的优化：列 c 在没有行这样的琐屑情形中既不被覆盖也不被还原。）注意超过半数的节点层数>=8，但是超过半数的更新发生在第 7 层之前。在前几层上的额外工作减少了后几层上困难工作的需求。
相应的，同一个问题在不利用 S 域的启发顺序时有着类似的统计量：

层数	结点数	更新次数	每个结点的更新次数
0	1 (0%)	2,031 (0%)	2031.0
1	6 (0%)	5,606 (0%)	934.3
2	24 (0%)	30,111 (0%)	1254.6
3	256 (0%)	249,904 (1%)	976.2
4	581 (1%)	432,471 (2%)	744.4
5	1,533 (1%)	1,256,556 (7%)	819.7
6	3,422 (3%)	2,290,338 (13%)	669.3
7	10,381 (10%)	4,442,572 (25%)	428.0
8	26,238 (25%)	5,804,161 (33%)	221.2
9	46,609 (45%)	3,006,418 (17%)	64.5
10	13,935 (14%)	284,459 (2%)	20.4
11	19 (0%)	14,125 (0%)	743.4
合计	103,005(100%)	17,818,752(100%)	173.0

当使用启发条件 S 时每次更新会用到 14 个存储空间，而忽略 S 时则是 8 个。因此在本例中启发条件 S 给总存储量乘上了一个数，大约是
(14*3,617,723)/(8*17,818,752)=36%。这个启发条件在大规模实例中效果更佳显著，因为它给总结点数减少了一个随层数呈指数增长的因数，同时执行它的代价仅仅是线性增长的。
假定启发条件 S 在大规模树中能发挥奇效，但对于小规模树却效果不佳，我尝试了一个混合方案即在低层使用 S 而在高层不用。然而，这个试验没有成功。如果，例如，S 在第 7 层被忽略，则 8 至 11 层的统计量如下：

层数	结点数	更新次数
8	18,300	5,672,258
9	28,624	2,654,310
10	9,989	213,944
11	19	10,179

接着如果我们在第 8 层执行改变，统计量就是：

层数	结点数	更新次数

9	11,562	1,495,054
10	6,113	148,162
11	19	6,303

因此我决定在 DLX 算法的所有层中都实施启发条件 S。

我那值得信赖的老 SPARCstation2 计算机，产于 1992 年，在处理大规模问题和维护 S 域时，每秒大约能执行 39 万次更新。斯坦福大学计算机系 1996 年购入的 120MHz 奔腾 I 计算机每秒能执行 121 万次更新，而我那新的 500MHz 奔腾 III 计算机每秒能执行 594 万次更新。因此运行时间随着技术的进步而减少；但是它仍然本质上和更新次数成比例，即链表舞蹈的次数。因此我更喜欢通过计算更新次数来衡量 DLX 算法的性能，而不是计算它用了多少秒。

Scott [34]很高兴地发现他那 MANIAC 上的程序在 3.5 小时内解决了骨牌问题。MANIC 每秒大约执行 4000 条指令，所以这代表粗略上有 5 千万条指令。他和 T.F.Trotter 发现一种使用 MANIC 的“位-与”指令的好办法，MANIC 上有 40 位的寄存器。他们的代码，对搜索树的每个节点执行大约 50,000,000 /(103,005+ 106,2 32+154,921)=140 条指令，相当有效，尽管事实上他们不得不处理因启发顺序而产生的将近 10 倍的节点。更进一步，DLX 算法的链表方法合计执行了 3,617,723+4,547,186+5,526,988=13,691,897 次更新，占用了 1 亿 9200 万的存储空间；而且它永远不可能符合 MANIC 那 5120 比特的存储空间！从这个立场来看，Dancing Links 技术对于 Scott 那 40 周岁的方法确实是个退步，尽管这个方法只适用于非常特殊的精确覆盖问题，即能利用简单几何结构的问题。

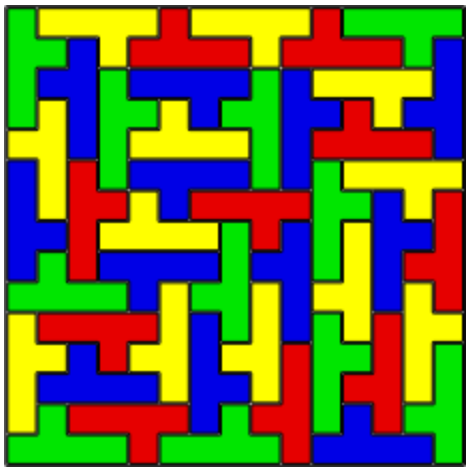
找出在 6*10 的矩形上放置骨牌所有方案的任务会比 Scott 那 8*8-2*2 的问题更加困难，因为 6*10 问题的回溯树更大，而且存在 2339 组本质不同的解 [21]。在这个情形下我们把 X 形骨牌限制在棋盘的左上角；我们的算法产生了 902,631 个节点和 309,134,131 次更新（或不使用启发条件 S 产生 28,320,810 个节点和 4,107,105,935 次更新）。它在奔腾 III 上可以用不足 1 分钟的时间解决这个问题；然而，我需要再次指出，骨牌的特殊角色允许有更快的算法。

John G.Fletcher 于 1965 年在 IBM7094 上仅用了 10 分钟就解决了这个问题，他利用了一个高度优化的程序，其内部循环有 765 条指令 [10]。7094 的时钟频率是 0.7MHz，并且它在一个时钟周期内能接受两个 36 位的字。Fletcher 的程序对于搜索树的每个节点仅需大约 600*700,000/28,320,810=15 个时钟周期；所以他的方法比 Scott 和 Trotter 的位运算方法更加高级，而且他是目前已知对于 12 骨牌放置问题最快的算法。(N.G. de Bruijn 好像已经将其独立探索出来了，参见 [7]。)

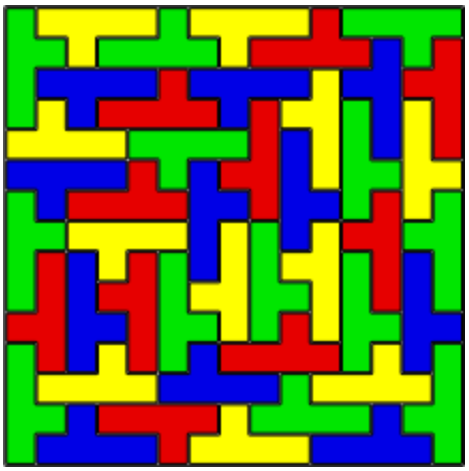
对 Dana Scott 的 0-1 矩阵问题稍作拓展，我们可以解决用 12 骨牌和一个四格板覆盖棋盘，且不限制四格板在棋盘中心的更一般的问题。这本质上是 Dudeney 的经典问题（Dudeney 在 1907 年发明了骨牌 [9]）。这样的棋盘划分总数显然没有在文献中出现；DLX 算法经过 1,526,279,783 次更新确定了它恰好是 16,146。许多人写了关于多联骨牌问题的文章，包括一些著名数学家，如 Golomb [15]，de Bruijn [7] [8]，Berlekamp，Conway 和 Guy [4]。他们对于放置骨牌的观点有时是基于枚举棋盘上填充单元格的方案数，有时是基于枚举可行的骨牌放置方案数。但是据我所知，没有人在以前指出这个问题是精确覆盖问题，在这个问题中，单元格和骨牌都具有优美的对称性。DLX 算法可以分支出难填格子或难放

骨牌的情况。这没什么区别，因为格子和骨牌都是给定输入矩阵中的列。

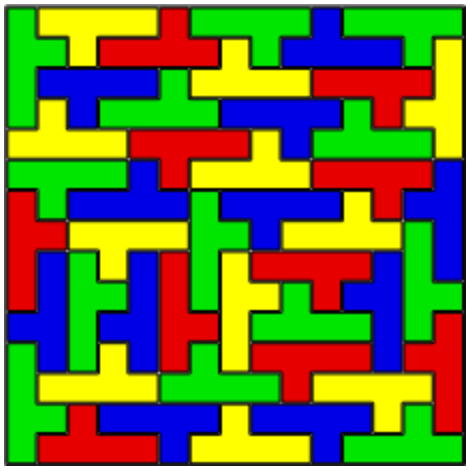
DLX 算法对于某些搜索树层数很多的问题，往往能比其他程序做得更好。例如，我们考虑在 15*15 的棋盘上放置 45 个 Y 型五格骨牌的问题。Jenifer Haselgrove 在被叫做 ICS Multum 的“快速迷你计算机”的帮助下于 1973 年研究了这个问题[20]。Multum 在一个多小时以后生成了一个答案，但她还不确定是否有其他的可行解。现在，利用上面介绍的 Dancing Links 方法，我们几乎可以立即得到一些解，而且打印出的方案多达 212 种。这些解可以根据四个角的状态归为四类；图 6 中展示了每类的一种方案：



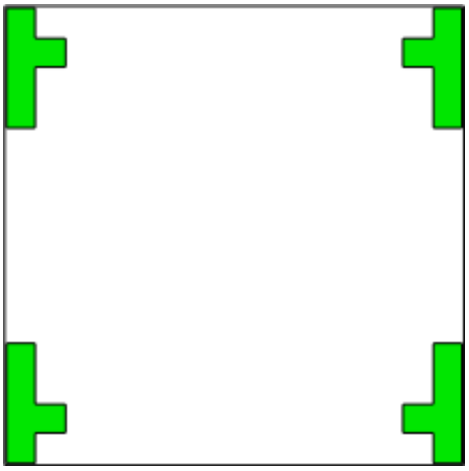
92 组解，14,352,556 个节点，1,764,631,769 次更新



100 组解，10,258,180 个节点，1,318,478,396 次更新



20 组解，6,375,335 个节点，806,699,079 次更新

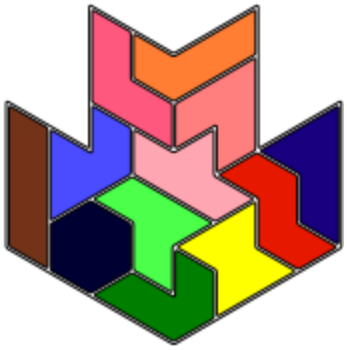


0 组解，1,234,485 个节点，162,017,125 次更新

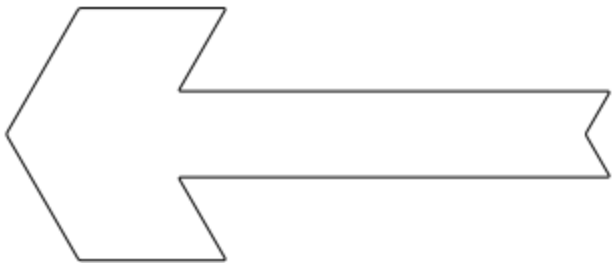
图 6 将 45 个 Y 型五格骨牌放入一个正方形

应用于六形组。在五十年代后期，T. H. O'Beirne 介绍了一种有趣的多联骨牌变种，即用三角形代替原骨牌中的方形。他将得出的形状称为多形组（polyiamond）：枢框形、宝石形、蛇形、蝶形、舟形等等。12 种六形组（hexiamonds）由 J. E. Reeve 和 J. A. Tyrell [32]独立发现，他们发现将六形组铺成 6*6 的菱形有超过 40 种方法。图 7 中展示了一种放置方案，以及几个

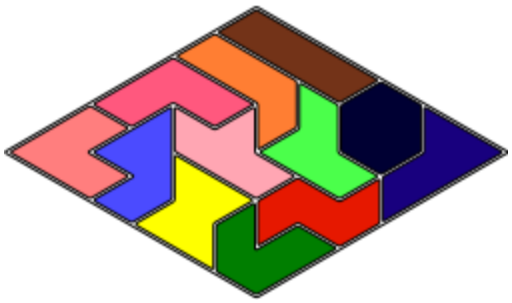
令我在刚开始接触六形组时忍不住动手尝试的箭头剖分。用这 12 种六形组铺成 6*6 的菱形恰有 156 种方案。（这个事实最初被 P. J. Torbijn [35]证明，他没有使用计算机；如果我们将“斯芬克斯形”的 12 种方向限制为 3 个，DLX 算法可以经过 37,313,405 次更新证实他的结论。）



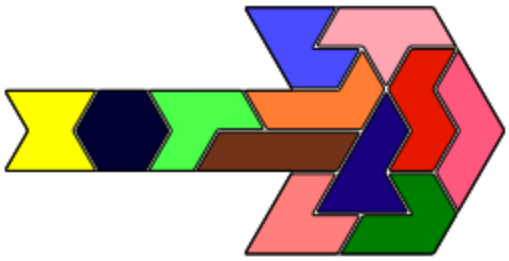
4 个解，6,677 个节点，4,687,159 次更新



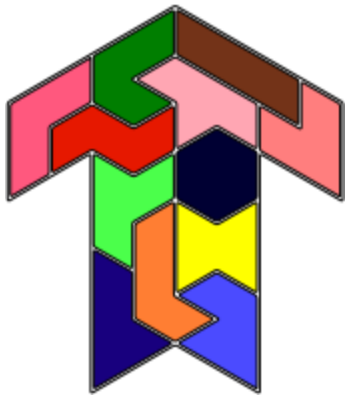
0 个解，7,603 个节点，3,115,387 次更新



156 个解，70,505 个节点，37,313,405 更新



41 个解，35,332 个节点，14,948,759 次更新



3 个解，5,546 个节点，3,604,817 次更新

图 7 将 12 种六形组塞入菱形和类箭头型

O’Beirne 对于 12 种六形组中有 7 种在反转时能够得到不同形状这一现象非常着迷，而且这 19 种单面六形组能够搭建一个六边形：即六形组之六边形（如图 8）。在 1959 年 11 月，经过了 3 个月的实验，他发现了一个解；两年之后他向《新科学家》的读者挑战来解决自己的问题 [28][29][30]。其间，他向 Richard Guy 和他的家庭展示了这个难题。Guy 的一家在新加坡，也就是 Richard 那时当教授的地方 [17]发表了几组解。Richard Guy 讲述了自己关于这个迷人游戏的故事

[18]。他说当 O'Beirne 第一次描述这个难题的时候，“每个人都想立即试一试，48 小时都没人睡觉。”

一个每层有许多可能性的19层搜索树为Dancing Links方法营造了极好的测试环境，因此我把 O'Beirne 问题扔给我的程序 ^_^。我根据六形组到中心的距离把一般情况打散为 7 个子情况；此外，当距离为 0 时我考虑“皇冠形”的两种子情况。图 8 展示了 7 种情况的例子以及搜索的统计量。更新的总次数有 134,425,768,494 次。

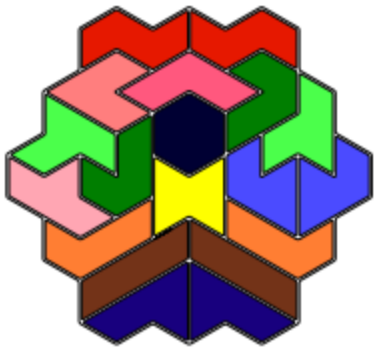
我的目标是不仅要计算出这些解，而且要寻找尽可能对称的安排，作为对 Berlekamp，Guy 和 Conway 的书《成功之路》 [4，第 788 页]中一个问题的答复。我们定义构型的水平对称性为左右翻转前后均在个块之间边的数目数。这个覆盖六边形内部有 156 条边，19 个单边六形组有 96 条内部虚边。因此如果一个方案完美对称——左右反转不发生改变——其水平对称性为 60。但是没有这样完美对称可行解存在。构型的垂直对称性定义类似，只是变为上下翻转。六形组问题的“最优对称解”是所有可行解中水平或垂直对称分值最大且小分值与大分值尽量接近的解。图 8 中展示了每一类中对称性最大的解。（而且对于图 1 中展示的 Dana Scott 问题的解也是如此：它的垂直对称性为 36，水平对称性为 30。）可能得到的最大垂直对称性是 50；它在图 8（c）中已经实现，而且其他的 7 个解通过分别对三个对称子部分重排列得到。这八个中的四个水平对称性为 32；其它的水平对称性为 24。John Conway 在 1964 年通过手算发现了这些解，并推测他们是“最佳对称覆盖”。但是这份荣誉只属于图 8（f）的解，至少根据我的定义，因为图 8（f）水平对称性为 52，垂直对称性为 27。其他水平对称性为 52 的几组解垂直对称性分别为 20，22 和 24。（其中的两种方法竟然有惊人的性质：19 块中的 13 块水平翻转前后没有发生变化；这是所有块的对称，而不仅仅是边。）

在我完成这个枚举之后，我首次阅读了[18]，并了解到 Marc M.Paulhus 在 1996 年 5 月已经枚举出了所有解 [31]。这很好，我的独立计算将验证这个结果。但是实际上他的程序并不正确——我的程序找到了 124,519 组解，他的程序找到了 124,518 组解！他在 1999 年重新运行了他的程序，现在我们达成了共识。

（a）

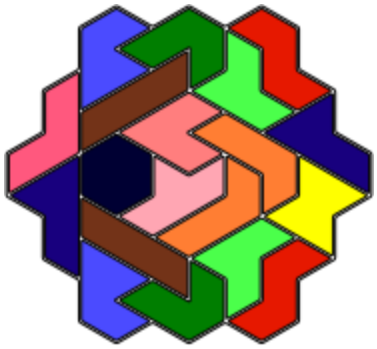
水平对称性=51，垂直对称性=24
1,914 个解，4,239,132 个节点，2,142,276,414 次更新

（b）



水平对称性=52，垂直对称性=24
5,727 个解，21,583,173 个节点，11,020,236,507 次更新

（c）　（d）　（e）



水平对称性=32，垂直对称性=50
11,447 个解，20,737,702 个节点，
10,315,775,812 次更新

（f）

水平对称性=51，垂直对称性=22
7,549 个解，24,597,239 个节点，
12,639,698,345 次更新

水平对称性=48，垂直对称性=30
6,675 个解，17,277,362 个节点，
8,976,245,858 次更新

（g）

水平对称性=52，垂直对称性=27
15,717 个解，43,265,607 个节点，21,607,912,011 次更新

水平对称性=48，垂直对称性=29
75,490 个解，137,594,347 个节点，67,723,623,547 次更新

图 8 0’Beirne 六形组六边形问题的解，小六边形到达六边形中心的距离各不相同。

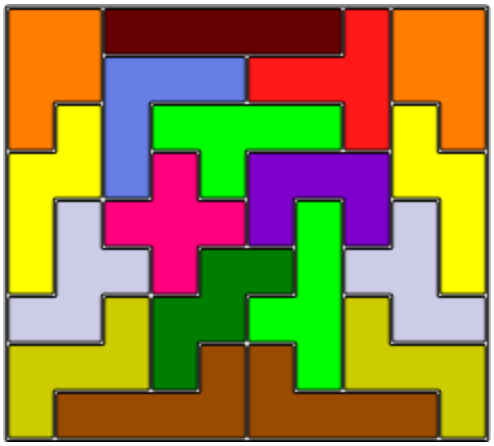
0’Beirne [29]还提出了一个类似的有 18 种单面骨牌的问题。他问能否把将这些骨牌塞入一个 9*10 的矩形中，而且 Golomb 在 [15，第 6 章]中提供了一个例子。Jenifer Leech 写程序验证了将单面骨牌放入 3*30 的矩形中恰有 46 种方案；详见 [26]。图 9 展示了一个“最佳对称”的例子（它事实上并不是十分对称）。

46 个解，605,440 个节点，190,311,749 次更新，水平对称性=27，垂直对称性=18

图 9 用单面骨牌填充 3*30 的矩形

我开始计算对于 9*10 矩形的解，每行用 6 个 1 来描述 18 阶段精确覆盖问题会比每行用 7 个 1 来描述 19 阶段精确覆盖问题容易。但是我很快发现这个任务没什么希望，除非我发现一个更好的算法。根据 [24]中的蒙特卡罗估算程序知道这

大概需要 19 后跟 15 个 0 这么多次更新，以及一个有 64 万亿的节点的巨型搜索树。如果这个估算是正确的，我可以在几个月内出解；但比起这我宁愿去找到一个新的麦森数。
然而，我设想了一个可能有最大水平对称值的解；如图 10：



水平对称性=72，垂直对称性=47
图 10 这是用单面骨牌填充矩形的最具对称性的方案吗？

一个失败的试验。基于“染色”的特殊观点往往在解决瓷片类问题时具有重要的洞察力。例如，众所周知，在 [5，第 142 和 394 页]中，我们移除两个对角上的格子，那么就无法用骨牌覆盖剩下的 62 个格子。原因是棋盘已经损毁，换句话说，就是有 32 个白色格子和 30 个黑色格子，但每块骨牌对于每种颜色只能覆盖一个单元格。如果我们对这个覆盖问题应用 DLX 算法，在得出无解的结论前，他会执行 4,780,846 次更新（而且找出 13,922 种方法来放置 31 块骨牌中的 30 块骨牌）。
六形组六边形问题可以用相似的方式黑白染色：所有指向左边的三角形都染成黑色，换句话说，就是所有指向右边的三角形都染成白色。那么单面六形组中的 15 种对于每种颜色都覆盖三个格子；而剩下的几种，即“斯芬克斯形”、“舟形”及其它们的对称形对于两种颜色分别占据 4 个格子和 2 个格子。因此这个问题的每组解必须恰好放置 2 个上述的四种六形组，并多占据黑格子。
我认为我应该对于每种放置这两块占黑格子多的六形组的方案都把问题划分成 6 个子问题来加速，每个子问题的解数大约是原问题的 1/6，而且每个子问题都更为简单，因为六形组中的四种都只能执行原来操作的一半。因此我期望子问题的运行速度能是原问题的 16 倍左右，而且我希望得到一些额外的关于六形组之间的约束信息，从而让 DLX 算法作出更明智的决策。
但我最终在数字上得到了很糟糕的结果。原问题对于图 8(c)用了 8,976,245,858 次更新得到了 6675 组解。粗略估计，六个子问题分别有 955，1208，1164，1106，1272 和 970 组解；但它们需要 17 亿到 22 亿次更新，而且完成所有子问题需要 11,519,571,784 次更新。这个“聪明”点子没带来什么好处。

应用于四形条。取代了方形和三角形，Brian Barwell [3]考虑用 4 条线段或木棒来构成图形。他将这个图形称为多形条，并记载说有 2 种散形条，5 种三形条，和 16 种四形条。四形条从娱乐的角度来看是十分有趣的；我在 1993 年收到了一

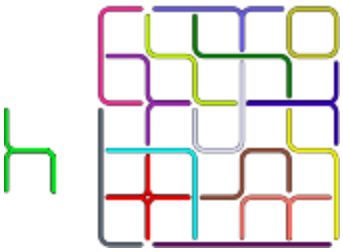
个吸引人的难题，这个问题等价于在 4*4 的方格中放置 10 个四形条 [1]，我花了好几个小时来攻破这个难题。

Barwell 证明了 16 种四形条不能凑成任何对称图形。但是如果不用五种水平和竖直线段数不等的四形条中的某一种，我们可以找到填充 5*5 方形的方案（见图 11）。这样的难题动手实现是相当困难的，而且他在写论文时也只找到了 5 组解；他估计可能有不到 100 组解。（全部解是由 Wiezorke 和 Haubrich [37]发现的，他们在看过 [1]后独立探索了这个谜题。）

多形条问题还具有一个骨牌问题和多形组问题没有的特点：所有拼块不能相交。例如，图 12 展示了考虑图 11（c）时发现的一个不可行解。虽然每条线段都覆盖了，但是“V”和“Z”相交了。

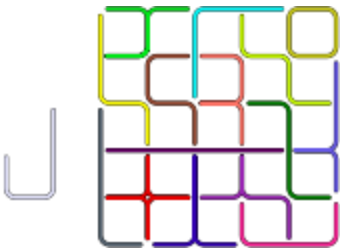
我们通过将其泛化为精确覆盖问题来处理这个复杂因素。取代了前面对 0-1 矩阵中列只能被不相交的行覆盖这个规则，我们将列区别成两类：初类和复类。这个泛化的问题要求每个初类列恰被覆盖一次，而每个复类列至多被覆盖一次。

图 11（c）中的四形条问题可以用很自然的方法转化为一般的覆盖问题。首先我们介绍初类列：F、H、I、J、N、O、P、R、S、U、V、W、X、Y、Z，他们分别代表 15 种四形条（包括 L），以及列 Hxy 和 Vxy，他们分别代表水平线段（x，y）——（x+1，y）和垂直线段（x，y）——（x，y+1），0<=x，y<5。我们还需要复类列 Ixy 来表示内部的连接点（x，y），0<x，y<5。就像骨牌问题和六形组问题一样，每行代表一块拼板放置方案，当然如果一个拼板在非边界处有连续的水平线段或者垂直线段，它还得算上内部连接点。



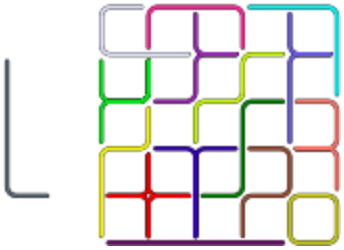
（a）

72 个解，1,132,070 个结点，283,814,227 次更新



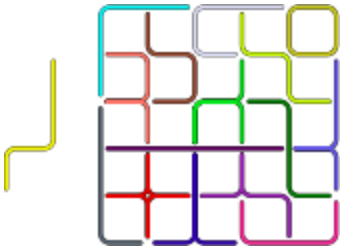
（b）

382 个解，3,422,455 个结点，783,928,340 次更新



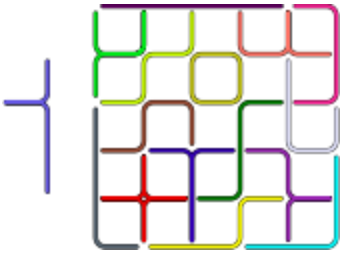
（c）

607 个解，2,681,188 个结点，611,043,121 次更新



（d）

530 个解，3,304,039 个结点，760,578,623 次更新



（e）

204 个解，1,779,356 个结点，425,625,417 次更新

图 11 用 16 种四形条中的 15 种填充 5*5 的方形，我们必须不用 H、J、L、N 或者 Y

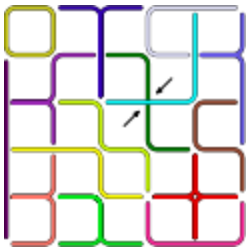


图 12 四形条不能在这里相交

举个例子，下面的两行就代表了图 12 中 V 和 Z 的放置方案：

V H23 I33 H33 V43 I44 V44
Z H24 V33 I33 V32 H32

其公共的元素 I33 意味着这两行发生了相交。另一方面，I33 不是初列，因为我们没有必要去覆盖它。图 11（c）中的解就只覆盖了内部点 I14、I21、I32 和 I41。幸运的是，我们几乎可以直接套用之前的算法来解决这个泛化覆盖问题。唯一的不同之处就是我们初始化时至给初列的列头做循环链表。每个复列的列头只需要将 L 和 R 域简单地指向自己就可以了。剩下的步骤和前面完全相同，因此我们仍将称之为 DLX 算法。

我们只需简单地给每个复列添加一个在该列包含一个 1 的行即可把泛化覆盖问题转化为等价的精确覆盖问题。但是我们最好针对泛化问题进行处理，因为泛化的算法会更加简洁、快速。

我决定在焊接类四形条的子集上试验，即那些因为包含分支点而没有形成简单路径的：F、H、R、T、X、Y。如果我们向对待骨牌和多形组一样加入那些不对称多形条的镜像图形则有 10 个单面焊接类四形条。而且——啊哈——这 10 种四条形可以塞入 4*4 的格子（见图 13）。只有三种可行解，包括下面展示的两种完美对称方案。我决定不展示出第三种解，X 在这个解的中央，因为我希望读者能够自己找到它。

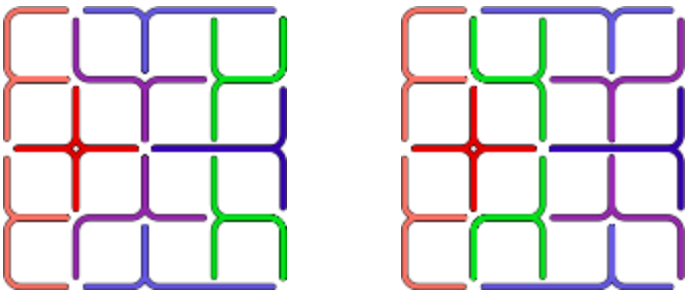


图 13 三种焊接类四条形填充方案中的两种

单面*非焊接类四形条*有 15 种，我想他们应该能用类似的方式填充 5*5 的格子；但最终我发现这是不可行的。因为如果 I 型垂直放置，J、J’、L、L’、N、N’中的四种就必须尽量水平放置，而这严格地限制了其可行性。事实上，我无法用这些拼板凑出任何简单的对称图形，我至今最大的成果就是凑出了图 14 所展示的“双簧管”。

图 14 15 种单面非焊接四形条

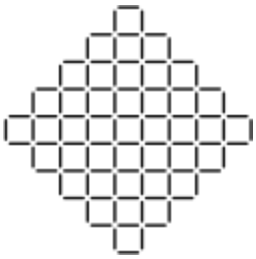


图 15 能用 25 种四形条构造这个图形吗？

我还做过一个不成功的尝试，就是将所有的 25 种单面四形条塞进图 15 中的这个阿芝台克宝石图；但我没能找到方法来证明这个解是不存在的。目前看来，详尽的搜索也是不可能的。

应用于皇后问题。现在我们回到那个促使 Hitotumatu 和 Noshita 介绍 Dancing Links 的问题，即 N 皇后问题，因为这个问题确实是我们应用于多形条的泛化覆盖问题的一个特例。例如，四皇后问题就是要求覆盖和行列对应的 8 个初列（R0、R1、R2、R3、F0、F1、F2、F3），以及与对角线对应的复列（A0、A1、A2、A3、A4、A5、A6、B0、B1、B2、B3、B4、B5、B6），只使用下面 16 行：

R0	F0	A0	B3
R0	F1	A1	B4
R0	F2	A2	B5
R0	F3	A3	B6
R1	F0	A1	B2
R1	F1	A2	B3
R1	F2	A3	B4
R1	F3	A4	B5
R2	F0	A2	B1
R2	F1	A3	B2
R2	F2	A4	B3
R2	F3	A5	B4
R3	F0	A3	B0
R3	F1	A4	B1

R3 F2 A5 B2
R3 F3 A6 B3

一般来说，N 皇后的 0-1 矩阵的每行应该是：

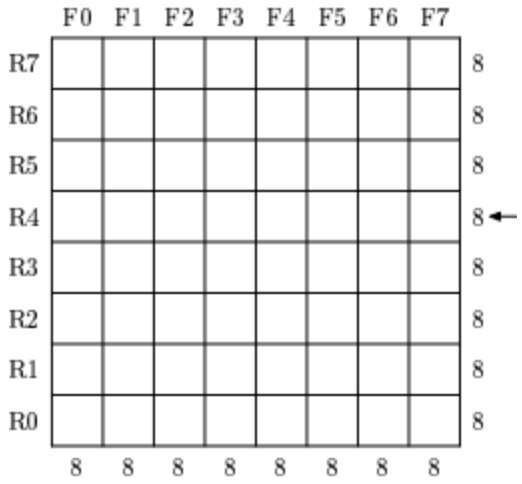
Ri Fj A(i+j) B(N-1-i+j)

对于 0<=i，j<=N。（这里 Ri 和 Fj 代表棋盘中的行和列，而 Ak 和 Bl 则代表对角线和逆对角线。复列 A(0)、A(2N-2)、B(0)和 B(2N-2)在矩阵中都只能出现一个行，因此可以忽略它们。）
当我们在对这个泛化覆盖问题使用 DLX 算法时，它的操作和 N 皇后的传统算法十分不同，因为它有时会分支出不同的方式来占据行和列。进一步说，我们可以考虑通过 S 值（分支因数）来改变分支顺序来提高效率：先放棋盘中间的，因为这约束了更多之后放置的可能性。
考虑例如八皇后问题。图 16（a）展示了一个空棋盘，有八种方式来占据行和列。假设我们决定在 R4 和 F7 的位置放置一个皇后，如图 16（b）。接着就有 5 种方式覆盖 F4；选择了 R5 和 F4 的位置之后，如图 16（c），有 4 种方法覆盖 R3，依此类推。在每个阶段我们选择约束性最强的行和列，利用“organ-pipe ordering”

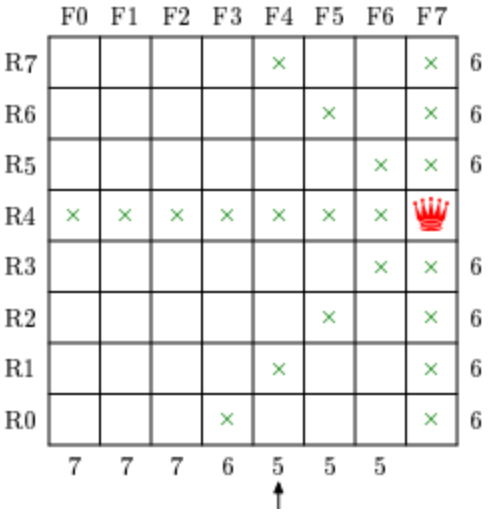
R4 F4 R3 F3 R5 F5 R2 F2 R6 F6 R1 F1 R7 F7 R0 F0

来打破约束。在图 16（d）的 R2 和 F3 位置放置皇后后导致无法覆盖 F2，所以只试放了 4 个皇后就要回溯。

（a）



（b）



（c）

（d）

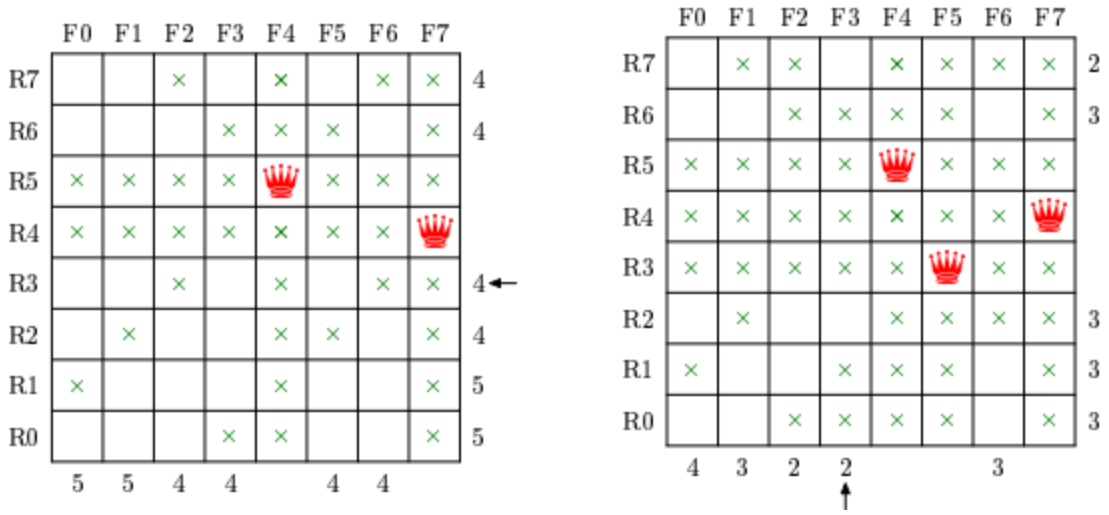


图 16 利用行列对称性解决八皇后问题

在 DLX 算法的开始时用这种顺序把首节点连接起来可以显著减少运行时间。例如，对于 16 皇后问题，如果使用 R0 R1 …… R15 F0 F1 …… F15 的顺序，则搜索树有 312,512,659 个节点并需要 5,801,583,789 次更新，但如果使用 organ-pipe ordering R8 F8 R7 F7 R9 F9 …… R0 F0，则仅需要大约原先 54%的更新次数。另一方面，将行或列连起来的顺序对于算法的总运行时间是没有改善的。

以下提供 DLX 使用 organ-pipe ordering 解决 N 皇后问题一些小规模情况的统计数据，这里没有使用对称性减少解的数目：

N	解数	结点数	结点更新次数	R 结点数	R 结点更新次数
1	1	2	3	2	3
2	0	3	19	3	19
3	0	4	56	6	70
4	2	13	183	15	207
5	10	46	572	50	626
6	4	93	1,497	115	1,765
7	40	334	5,066	376	5,516
8	92	1,049	16,680	1,223	18,849
9	352	3,440	54,818	4,640	71,746
10	724	11,578	198,264	16,471	269,605
11	2,680	45,393	783,140	67,706	1,123,572
12	14,200	211,716	3,594,752	312,729	5,173,071
13	73,712	1,046,319	17,463,157	1,589,968	26,071,148
14	365,596	5,474,542	91,497,926	8,497,727	139,174,307
15	2,279,184	31,214,675	513,013,152	49,404,260	800,756,888
16	14,772,512	193,032,021	3,134,588,055	308,130,093	4,952,973,201

17 95,815,104 1,242,589,512 20,010,116,070 2,015,702,907 32,248,234,866
18 666,090,624 8,567,992,237 141,356,060,389 13,955,353,609 221,993,811,321

这里“R 结点数”和“R 结点更新次数”指代我们只考虑将 R0、R1、……、R(N-1)作为需要覆盖初列时的结果；列 Fj 是复列。这样一来减少了一些只在棋盘的列上进行分支的操作。随着 N 的增长，将行和列进行混合的优势越来越明显，但是我不确定 R 结点更新次数和 RF 结点更新次数的比率在 N 趋近于无穷大时是极大还是收敛于一个常数。

我应当指出，其实也有不需要生成确切的放置皇后的方案就可以得到方案数的特殊方法存在 [33]。

总结备注。使用 Dancing Links 来对精确覆盖问题执行“自然”算法的 DLX 算法，是枚举这类问题的所有解的一种有效方法。对于小规模情况，这个算法与专门解决这类具有几何性质的问题（如 N 皇后问题，骨牌放置问题）的特殊算法速度差不多。而在大规模情况下，它甚至能比那些特殊算法更快，因为它有启发式的搜索顺序。而且随着计算机的速度越来越快，我们总能够计算越来越大规模的数据。

在这篇论文中我使用精确覆盖问题阐明了 Dancing Links 功能的多样性，但其实我还可找到更多能够渗透了这种思想的回溯应用。例如，对 Waltz 过滤算法的精确逼近 [36]；或许正是它潜意识地引导我选择了这个名字。我最近对英文词典中大约 600 个三字母单词使用 Dancing Links 来寻找下面这样的方阵：

ATE	BED	OHM	PEA	TWO
WIN	OAR	RUE	URN	ION
LED	WRY	BET	BAY	TEE

这些方阵的每行、每列和每个对角线都是一个单词；大约 6 千万次更新就得出了所有的解。正如 Haralick 和 Elliott 在关于约束满足问题的早期论文中考虑的一样 [19]，我相信舞蹈技术从长远来看会比在每层复制当前状态要好得多。这个方法确实更加简单、实用、有趣。

“What a dance / do they do / Lordy, I am tellin’ you!” [2]

致谢。感谢 Sol Golomb、Richard Guy 和 Gene Freuder 在我准备这篇论文时给我的帮助。感谢 Maggie McLoughin 将我凌乱的手稿制作成 TeX 文档时的出色工作。而且我由衷地感谢 Tomas Rokicki，他为我的试验提供了新型电脑，我希望能在这台电脑上愉快地跳几年链表舞。

历史注记。（1）虽然 IAS 计算机在普林斯顿广泛被认为是”MANIAC”，但它其实属于类似却又不同的产于 Los Alamos 的计算机系列(参见 [27])。（2）George Jelliss [23]发现著名谜题制作者 H. D. Benjamin 和 T. R. Dawason 在 1946-1948 进行了多形条概念的实验。然而他们显然没有公布任何工作进展。（3）我对四形条的命名和 Barwell 最初的命名有点出入 [3]：我喜欢用 J、R 和 U 来称呼被他称作 U、J 和 C 的几个拼板。

程序。文件 dance.w 是我在准备这篇论文时使用的 DLX 算法的实现，在网页 http://www-cs-faculty.stanford.edu/~knuth/program.html 上可以找到。相关的文件还有 polyominoes.w、polyamonds.w、polysticks.w 和 queens.w。

参考资料

[1] *845 Combinations Puzzles: 845 Interestingly Combinations* (Taiwan: R.O.C. Patent 66009). [There is no indication of the author or manufacturer. This puzzle, which is available from www.puzzletts.com, actually has only 83 solutions. It carries a Chinese title, "Dr. Dragon's Intelligence Profit System."]

[2] Harry Barris, *Mississippi Mud* (New York: Shapiro, Bernstein & Co., 1927).

[3] Brian R. Barwell, "Polysticks," Journal of Recreational Mathematics **22** (1990), 165–175.

[4] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy, *Winning Ways for Your Mathematical Plays* **2** (London: Academic Press, 1982).

[5] Max Black, *Critical Thinking* (Englewood Cliffs, New Jersey: Prentice-Hall, 1946). [Does anybody know of an earlier reference for the problem of the "mutilated chessboard"?]

[6] Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, *Structured Programming* (London: Academic Press, 1972).

[7] N. G. de Bruijn, personal communication (9 September 1999): "... it was almost my first activity in programming that I got all 2339 solutions of the 6*10 pentomino on an IBM1620 in March 1963 in 18 hours. It had to cope with the limited memory of that machine, and there was not the slightest possibility to store the full matrix... But I could speed the matter up by having a very long program, and that one was generated by means of another program."

[8] N. G. de Bruijn, "Programmeren van de pentomino puzzle," *Euclides* **47** (1971/72), 90–104.

[9] Henry Ernest Dudeney, "74.--The broken chessboard," in *The Canterbury Puzzles*, (London: William Heinemann, 1907), 90–92, 174–175.

[10] John G. Fletcher, "A program to solve the pentomino problem by the recursive use of macros," *Communications of the ACM* **8** (1965), 621–623.

[11] Robert W. Floyd, "Nodeterministic algorithms", *Journal of the ACM* **14** (1967), 636-644.

[12] Martin Gardner, "Mathematical games: More about complex dominoes, plus the answers to last month's puzzles," *Scientific American* **197**, 6 (December 1957), 126-140.

[13] Michael R. Gareyand David S. Johnson, *Computers and Intractability* (San Francisco: Freeman, 1979).

[14] Solomon W. Golomb, "Checkerboards and polyominoes," *American Mathematical Monthly* **61** (1954), 675-682.

[15] Solomon W. Golomb, *Polyominoes*, second edition (Princeton, New Jersey: Princeton University Press, 1994).

[16] Solomon W. Golomb and Leonard D. Baumart, "Backtrack programming," *Journal of the ACM* **12** (1965), 516-524.

[17] Richard K. Guy, "Some mathematical recreations," *Nabla* (Bulletin of the Malayan Mathematical Society) **7** (1960), 97-106, 144-153.

[18] Richard K. Guy, "O'Beirne's Hexiamond," in *The Mathemagician and Pied Puzzler*, edited by Elwyn Berlekamp and Tom Rodgers (Natick, Massachusetts: A. K. Peters, 1999), 85-96.

[19] Robert M. Haralick and Gordon L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," *Artificial Intelligence* **14** (1980), 263-313.

[20] Jenifer Haselgrove, "Packing a square with Y-pentominoes," *Journal of Recreational Mathematics* **7** (1974), 229.

[21] C. B. and Jenifer Haselgrove, "A computer program for pentominoes," *Eureka* **23**, 2 (Cambrige, England: The Archimedeans, October 1960), 16-18.

[22] Hirosi Hitotumatu and Kohei Noshita, "A technique for implementing backtrack algorithms and its application," *Information Processing Letters* **8** (1979), 174-175.

[23] George P. Jelliss, "Unwelded polysticks," *Journal of Recreational Mathematics* **29** (1998), 140-142.

[24] Donald E. Knuth, "Estimating the efficiency of backtrack programs," *Mathematics of Computation* **29** (1975), 121-136.

[25] Donald E. Knuth, TeX: The Program (Reading, Massachusetts: Addison-Wesley, 1986).

[26] Jean Meeus, "Some polyominio and polyamond problems," *Journal of Recreational Mathematics* **6** (1973), 215-220.

[27] N. Metropolis and J. Worlton, "A trilogy of errors in the history of computing," *Annals of the History of Computing* **2** (1980), 49-59.

[28] T. H. O'Beirne, "Puzzles and Paradoxes 43: Pell's equation in two popular problems," *New Scientist* **12** (1961), 260-261.

[29] T. H. O'Beirne, "Puzzles and Paradoxes 44: Pentominoes and hexiamonds," *New Scientist* **12** (1961), 316-317. ["So far as I know, hexiamond has not yet been put through the mill on a computer; but this could doubtless be done."]

[30] T. H. O'Beirne, "Puzzles and Paradoxes 45: Some hexiamond solutions: and an introduction to a set of 25 remarkable points," *New Scientist* **12** (1961), 379-380.

[31] Marc Paulhus, "Hexiamond Homepage," http://www.math.ucalgary.ca/~paulhusm/hexiamond1.

[32] J. E. Reeve and J. A. Tyrell, "Maestro puzzles," *The Mathematical Gazette* **45** (1961), 97-99.

[33] Igor Rivin, Ilan Vardi, and Paul Zimmermann, "The *n*-queens problem," *American Mathematical Monthly* **101** (1994), 629-639.

[34] Dana S. Scott, "Programming a combinatorial puzzle," Technical Report No.1 (Princeton, New Jersey: Princeton University Department of Electrical Engineering, 10 June 1958), ii+14+5 pages. [From page 10: "... the main problem is the program was to handle several lists of indices that were continually being modified."]

[35] P. J. Torbijn, "Polyiamonds," *Journal of Recreational Mathematics* **2** (1969), 216-227.

[36] David Waltz, "Understanding line drawings of scenes with shadows," in *The Psychology of computer Vision*, edited by P. Winston (New York: McGraw-Hill, 1975), 19-91.

[37] Bernhard Wiezorke and Jacques Haubrich, "Dr. Dragon's polycons," *Cubism For Fun* **33** (Febuary 1994), 6-7.

补注 在 1999 年 11 月，拜罗伊特大学（Universität Bayreuth）的 Alfred Wassermann 成功用单面四形条覆盖了阿芝台克宝石图，他利用计算机集群运行了 DLX 算法。这个解答相当漂亮，它已经被发布在
http://did.mat.uni-bayreuth.de/wassermann/allsolutions.ps.gz

译者的话

正如 Knuth 本人所说，作者写作本文的目的正是希望传播这个看似很简单的程序技巧。而译者翻译本文，也是希望达到同样的目的。
本文基本忠于原作，部分图片的位置根据版式以及内容进行了微调。如果您认为翻译有什么不当的地方，请及时联系我们。

关于 Dancing Links 这个词语的翻译，在第一次出现的部分我们将它称之为“舞蹈链”。但是之后出现的地方，我们全部用“Dancing Links”替代。因为我们觉得，“舞蹈链”这个名字，不能很好的体现出这个算法的美妙之处，于是我们沿用了 Knuth 的命名。如果你有什么更好的命名建议，也请联系我们。

译者隋清宇的话 当初看到这篇文章的时候，就被 Dancing Links 的美妙性质深深迷住。我把这篇文章交给吴豪的时候，他也有同样的想法。但是，这篇并不短的英文论文，给阅读带来了不小的障碍。
于是，我和吴豪就开始着手文章的翻译工作。刚开始部分的翻译很顺利，但是当翻译了几乎一半的时候，我们的工作不得不暂停下来——主要原因是我即将参加 NOI（全国信息学奥林匹克竞赛，http://www.noi.cn/）。接下来的一段时间，这件事情几乎被我忘记了。
但就在前段时间，一个关于动态规划的英文文章（我们有在之后的某个时间开始翻译这篇文章的想法）突然让我和吴豪想起了这篇还没有完成的 Dancing Links 论文的翻译。于是找出以前的进度，然后在几天的时间内完成了最后的翻译工作。实际上，这篇文章的大部分翻译工作都是吴豪一手完成的，在这里我代表个人对他表示谢意。
以前曾经写过一些小东西，比较成功的比如 Splay 的论文，而比较弱智的就很多了，比如 KMP 那篇不知所云的东西。不过对文章进行翻译，这是第一次，而且是 Knuth 大师的这么长的论文。如果没有吴豪的帮助，这个任务肯定不能完成。现在翻译即将结束，我感觉愈发的激动。
希望我这篇文章能够对普及 Dancing Links 算法起到一些微不足道的推动作用，这样我就可以感到很满足了。
（现在你看到的是文章的半成品——已经发出的部分是我已经完成校对和排版的部分。而剩余部分的翻译已经完成，正在紧张的校对和排版中。我会经常不定期更新，这句话将在排版全部完成后消失，那时也将放出压缩包的下载）

译者吴豪的话
链表，充满奇幻色彩的数据结构；
　回溯，解决难题的万能钥匙。
Knuth 教授在这篇文章中将两者有机地结合到了一起，
　从此，算法领域诞生了一颗新星———Dancing Links。

我们将目睹，链表跳着那纤美、卓绝的舞步，优雅地在搜索树的世界穿梭；
　　我们将领略，舞蹈技术的准确与迅速。
从此，难题不再那么高不可攀；
　　面对困难，我们也不再畏葸不前。
NP 不再是梦想；
　　AC 也成为了可能。
来吧，让我们和 Knuth 教授一起，进入 Dancing Links 的世界；
　　微笑着，带着轻盈的舞步，迎接新的挑战！