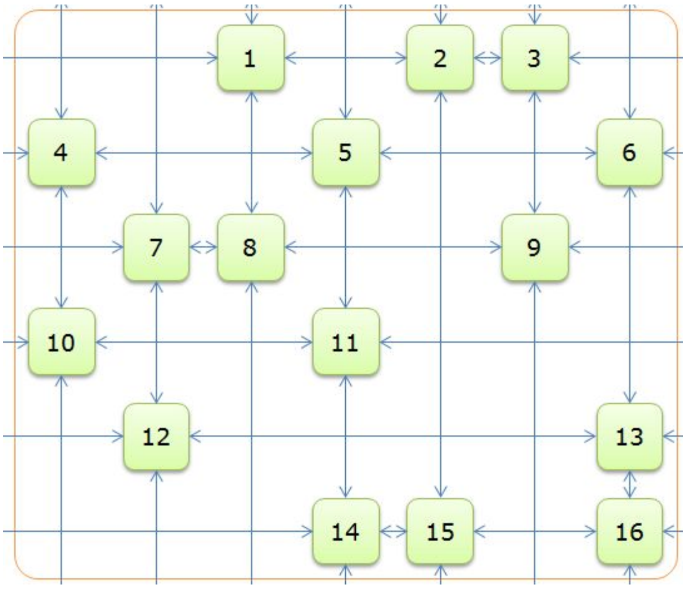
十字链表与 DLX 算法

by immortalCO

1 概述

十字链表，即所谓的 Dancing Links，是一种**双向链表**。和普通链表不同，每一个节点都有 4 个指针，分别指向它（在一个矩阵中）上、下、左、右的位置。



十字链表实现：4 个指针（最后一个元素指向第一个，达到循环）

它完美的特性，可以节省大量的时间，甚至会超过很多常数优化的非常好的程序。

Dancing Links X 算法是十字链表一个绝妙的运用，它一般是用于求解这样一类问题：

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

精确覆盖问题：选出 01 矩阵中的几行，使每列有且只有一个 1

比如，1、4、5 行可达到目的

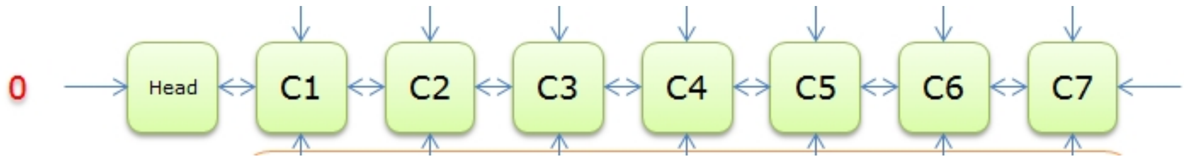
解决精确覆盖问题的算法被称作 X **算法**，而使用了 Dancing Links 的算法就被称作 DLX 算法。精确覆盖问

题的模型适用于很多问题，特别是 NP 类问题。

2 美在哪里？

2.1 绝妙的储蓄结构

双向链表的储蓄结构往往是由一个空的节点表示**头指针**，每个节点有一个**”前“和”后“指针**，分别指向它的”上一个“和“下一个”节点。而十字链表的每个节点则有 4 个指针，相比普通双向链表，多了**”上“和”下“指针**，其意义想必也不难理解。除了总的头指针，十字链表还有**”列“的头指针**，指向每个列的开头第一个元素。**”行“的头指针**似乎没那么必要，因为只要有”列“头指针就能定位节点，但在特殊的问题中，”行“的头指针还是有必要的。更多情况下，对于每个节点，我们还需要**记录它所在的列和行**，方便我们定位其头指针（列）以及是哪个 01 序列的一部分（行）;对于每个”列“头指针，我们还需要记录该列剩余节点数，方便确认搜索顺序，加快速度。



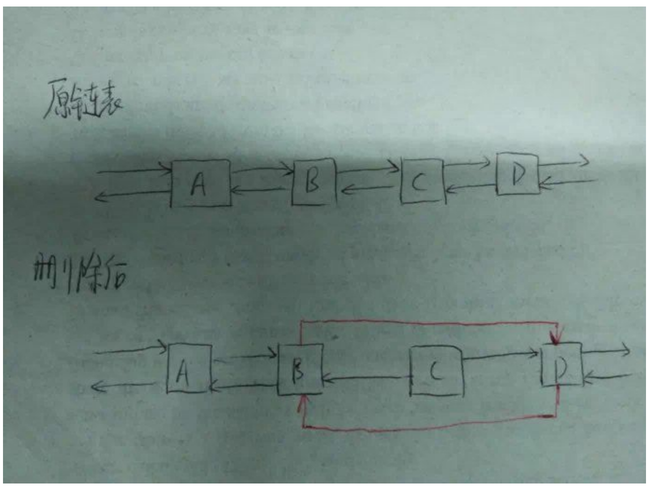
十字链表基本储蓄结构

2.2 优美的操作

众所周知，在单向链表中删除一个节点是低效而且麻烦的，而且恢复节点更是如此。但是我们的双向链表有着优美的结构，使删除和恢复变成了一种自然的动作。删除有如下操作：

L[R[x]]=L[x]，R[L[x]]=R[x]

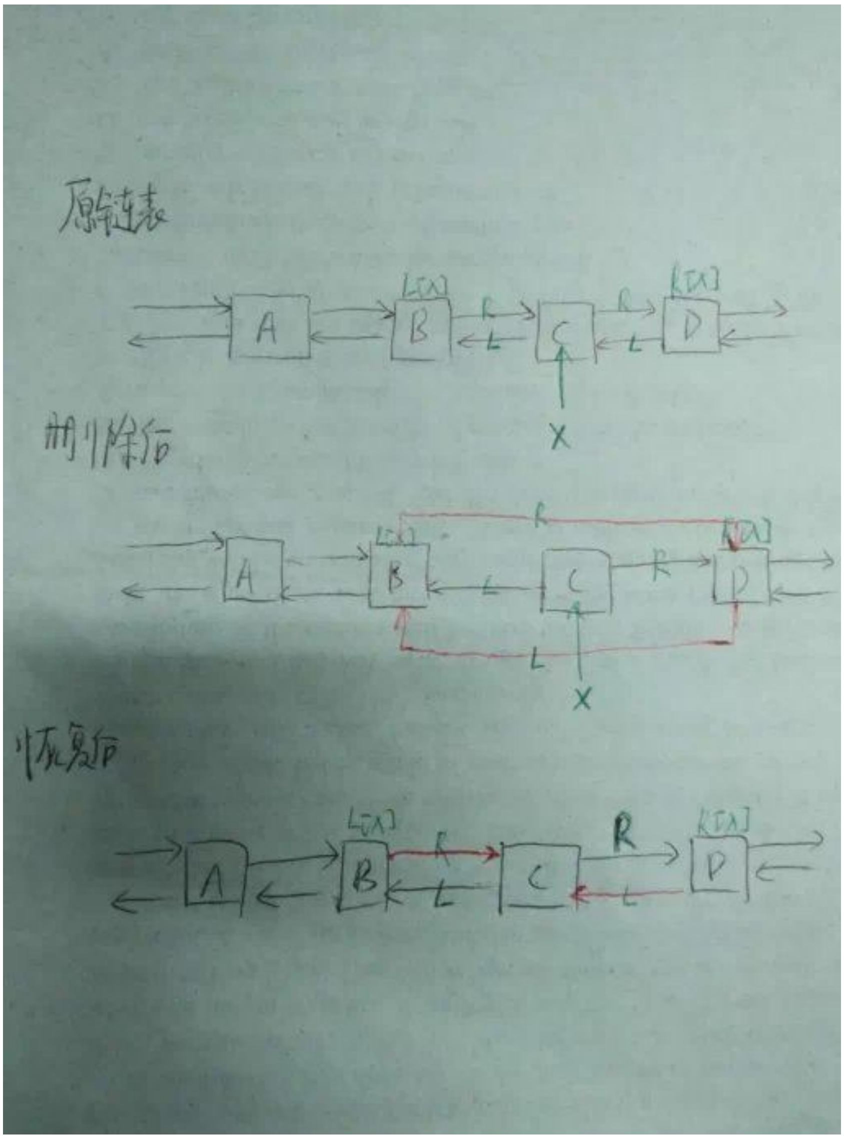
在这种绝妙的语句下，链表变成了这个样子：



有人可能会说：既然 C 节点删除了，就应该把 C 的一切删的一干二净，但在这里我们根本不必（事实上是不行）将 C 删除，因为在之后，我们可能还要将它还原。用了这样的语句，还原也变得容易了许多。有如下语句：

L[R[x]]=x，R[L[x]]=x

你发现了什么？没错！由于我们使用的这种**不完全删除**，虽然 C 并不可被访问到，但 C 的左、右指针仍然有效！只需要简单的恢复，就能将 C 点恢复成原状。下面的图说明了删除和还原操作的具体细节。如果还是不能理解，可以叫你的朋友和你一起玩指人游戏。虽然有些幼稚，但对你理解算法有很大帮助。



返回来想：如果我完全删除了 C，如何恢复 C 呢？自然少不了新建节点、判断前后关系等麻烦的操作，而现在我们的操作只需要两个简单的赋值，何乐而不为呢？

在这里，你可能明白了这样做的妙处，但还是会有疑问：这样做有什么用呢？接下来我们会进入另一个内容，很快你就能领略到十字链表的美妙之处。

2.3 X 算法简介——精确之美

精确何尝不是一种美丽呢？

上面那句话可能是为了切题而加的牵强之语，接下来的算法你可能无从体会，但很快你就能同时领略到所有一切的极美极妙之处。如果没耐心看完这一章，还是请求你耐着性子看完，否则连更美的东西都体会不到了。

回到最初的问题：

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

精确覆盖问题

显然这个问题是 NP 问题，是不可在多项式时间内求解的。对于这个问题，在搜索过程中，我们可以用**转化**的思想，把它转化成一个较小的精确覆盖问题。

如果我们手动来做，会怎么做呢？

假定我们选择第一行。第一行有 1 的位置 3、5、6 已经有了 1，所以在 3、5、6 位置也有 1 的 01 序列 3、6 就肯定不会选到，可以将它们**从行中删除掉**（紫色）。而由于 3、5、6 行的部分已经完成，也可以将它们**从列中删除掉**（蓝色）。问题转化成了这样：

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

假定再选择第一行，则按照上述规则删除掉行和列后，情况发生了：

（0）

存在一列无法被删除，问题求解失败。将问题回溯到之前的状态

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

假定选择第二行，删除行和列，矩阵转化成了如下形式：

（1 1）

发现了什么？是的，只要这一步删除，就能将整个矩阵清空，即问题解决。

让我们总结上面的思路，可以得到这样的算法：

1、按照搜索顺序，选取当前状态中还没被删除的一行

2、删除并扫描这一行，将这一行中带 1 的列删除

3、扫描所有行，如果当前行和选择的这一行的同一列都拥有 1，则删除当前行

4、如果矩阵为空，则解找到，推出。

5、否则，递归处理这个删除了一部分的矩阵（DFS）

这便是 Knuth 提出的 X 算法。

2.3 发现美

让我们一起分析一下上面的算法的快慢：

由于我们只需要搜索出一个解，为了达到最小时间，我们应该以最快的方式找到它。这时我们就应该选择 **1 最少的列**进行 DFS。…………查找效率 O(m)

有经验的选手会用位运算取代数组，但除了常数之外并没有加速。每次删除都要判断是否为 1，并打上标记。…………O(n+m)

看起来挺快的，是不是？但你细心一想，问题就来了。让我们回顾一下上面的计算过程——

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

（1）选择行 1，删除行 1、3、6，删除列 3、5、6；

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

（2）选择行 1，删除行 1、2、3，删除列 1、3、4；

（0）

（3）选择行 1，无法删除，求解失败，回溯；

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

（4）选择行 2，删除行 1、删除列 1\3\4；

（1 1）

（5）选择行 1，删除行 1、删除列 1\2；

( )

（6）矩阵的列数为 0，求解完毕。

模拟这个做法。我们会把这样的矩阵存在一个布尔数组中（或者整数数组），n=6，m=7：

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

在执行操作（1）后，我们以为矩阵是这样的：

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

在转化成子问题时，我们让它变成了一个 n=3，m=4 的矩阵，但是并非如此！！！事实上，它还是一个 n=6，m=7 的矩阵！为什么会这样呢？让我们分析我们的处理方式：

<u>删除有关的行和列</u>

可能这并不能说明问题，让我们再具体一些：

<u>在有关的行和列打上删除标记</u>

发现问题了吗？打标记只是简单的记上几个布尔值，**并不是真正的删除**。我们的矩阵还是一个 n=6，m=7 的矩阵！在数组中，它是这样表示的：

[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
1	0	[redacted]	1	0	[redacted]	1
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
1	0	[redacted]	1	0	[redacted]	0
0	1	[redacted]	0	0	[redacted]	1
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]

当你判断打了标记并 continue;的时候，恰恰增大了复杂度，浪费了时间！这是很多初学者会犯的错误。continue 并不能节省时间。试想，你有一个一亿×一亿的矩阵，在其中删除了九千九百九十九行、九千九百九十九列，但你在判断的时候，仍然是要扫过整个一亿×一亿的矩阵，并在其中跳过九千九百九十九次，而非扫描一个 10×10 的迷你矩阵。这个效率简直是天壤之别。而要把它优化掉，就必须把它真正的删除，变成这样的一个矩阵：

1	0	1	0	1
1	0	1	0	0
0	1	0	0	1

如何真正的删除呢？答案是——

链表

2.4 优化之美

再次分析上面的 X 算法：

1、按照搜索顺序，选取当前状态中还没被删除的一行

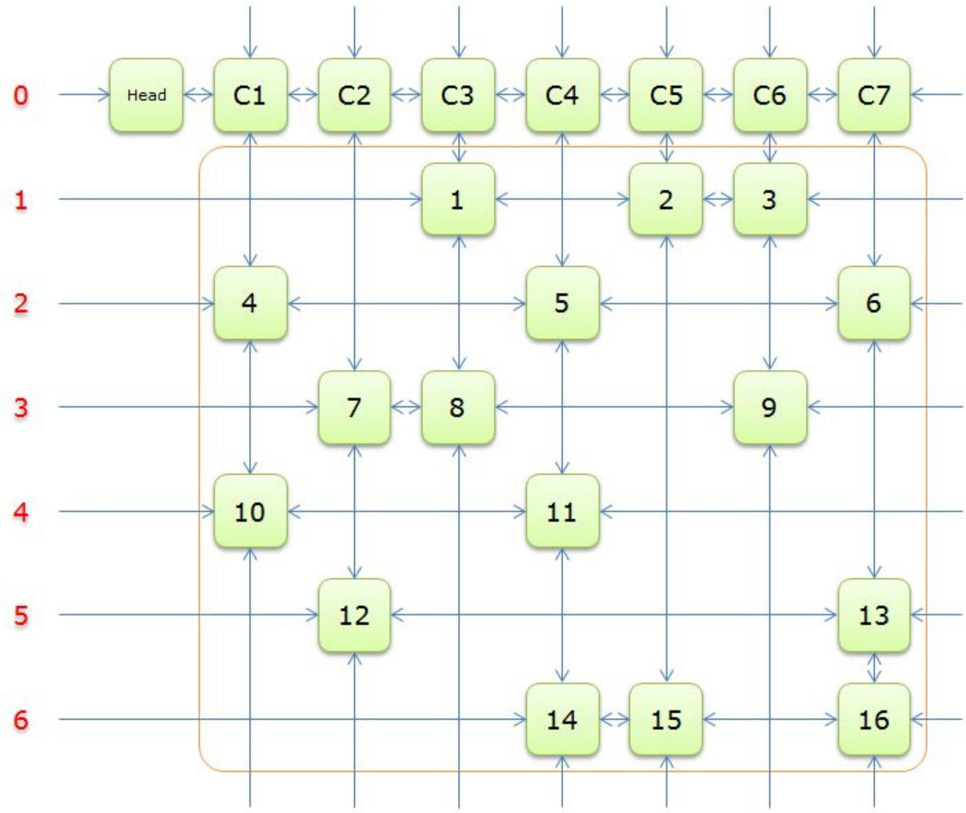
2、删除并扫描这一行，将这一行中带 1 的列删除

3、扫描所有行，如果当前行和选择的这一行的同一列都拥有 1，则删除当前行

4、如果矩阵为空，则解找到，推出。

5、否则，递归处理这个删除了一部分的矩阵（DFS）

我们看到上面只有一个数字：1。这就是我们优化的入手处。在矩阵中，只有 1 是有用的，而这个矩阵往往是稀疏的（否则多半无解）。这时候，本文章的主题：双向十字链表（Dancing Links）自然引出。既然只需要 1，我们就用双向十字链表存下这个矩阵，只记里面的 1：



即使我不继续介绍，你也能明显的发现：整个空间占用，从原来的 7×6 变成了现在的 16（1 的个数）！

有人可能会问：删除简单，恢复容易吗？那么就请看上面的删除和恢复代码。这种不完全删除的性质使得两个操作都易如反掌。

接下来的下一块，我就要讲解具体的算法，如何用 Dancing Links 实现 X 算法。

2.4 舞蹈之美（这是本文的核心内容）

终于讲到重点了，想想就有些小激动（= =）。

一个比较好的实现方式是用数组来模拟链表，这样也可方便的建立矩阵，也可以加快运行速度。对每一个对象，记录如下几个信息：

L[x]：指向 x 位置左边第一个 1 的地址

R[x]：指向 x 位置右边第一个 1 的地址

U[x]：指向 x 位置上面第一个 1 的地址

D[x]：指向 x 位置下面第一个 1 的地址

C[x]：指向 x 位置所在列头的地址

Information[x]：和 x 所在行有关，为该行对应的编号（或信息）

对于每个列头，我们还需要记录这些：

S[x]：当前列存在节点数

对于整个 Dancing Links，我们还需要：

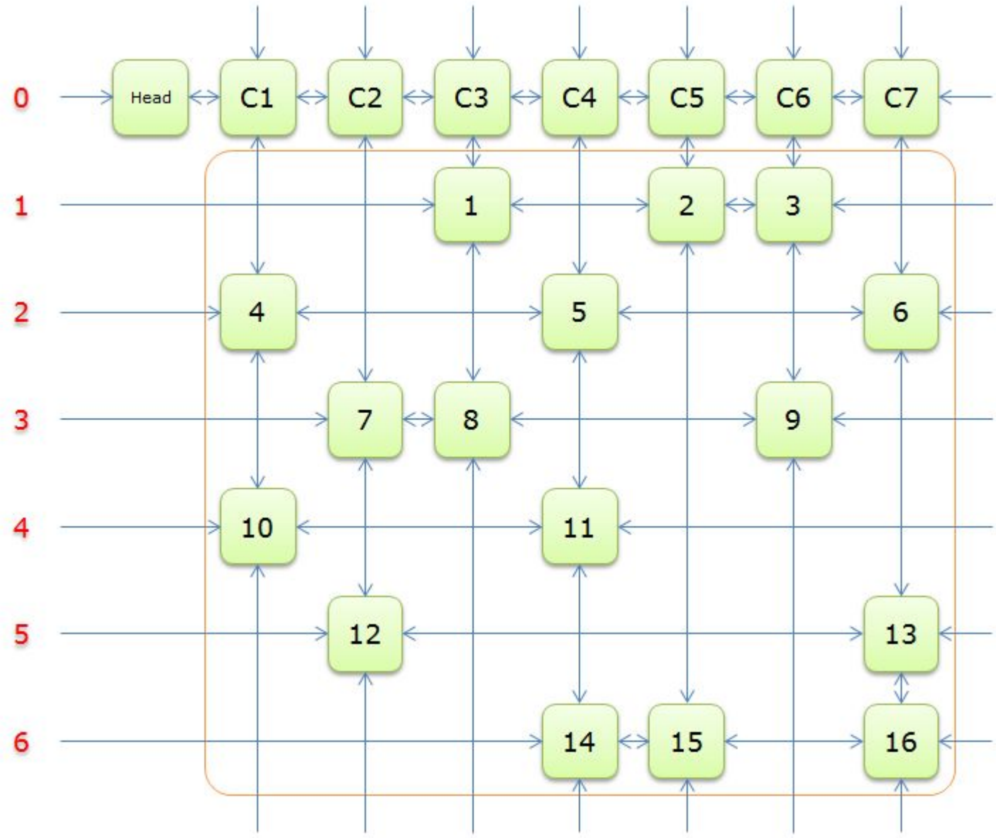
head：虚拟的头指针，一般为 0

Col[]：列头指针，一般为 1~m，无需记录

Ans[]：记录搜索结果，是一个栈，压入的是 Information 中的内容

top：整个内存空间的大小，方便新建节点

这样，我们便建好了相应的 Dancing Links。下面就是我们建好的 Dancing Links 的真实结构图。

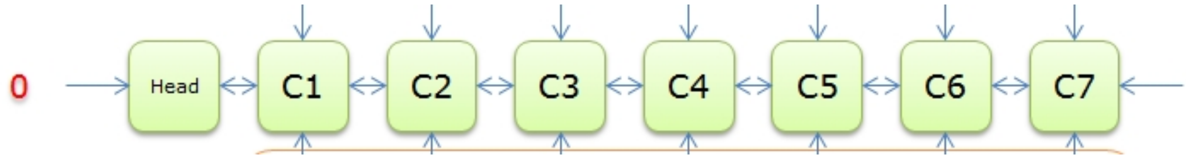


就这个图说明一下，head 一整行用于辅助访问所有的列头。和 head 这一行一样，每个列头都是一个循环的十字链表，这就像我们的**钥匙圈**，或者管道。而我们就要运用循环的性质，来完成我们最重要的步骤，也

就是“跳舞”。

在解决 X 算法的问题前，我们还要介绍一些细节，包括如何初始化和插入节点，以及如何迭代。在常数要求很高的题目（尤其是数独，本身就是一个大常数），这些显得极为重要。

初始化，由于我们并不知道矩阵的具体内容，只知道有 m 列，所以我们事先建好 head 和列头节点。最后我们的要求是这样的：



根据分配规则和易用性原则，head 指针是 0,其他列头 Col[i]=i。对于列头指针 i，由于列是空的，所以 L[i]=R[i]=i（本身）且 i 列大小 S[i]=0。每一列的前一和后一节点分别是前一和后一的列头指针，由于易用性原则，所以 L[i]=i-1，R[i]=i+1。但这并不通用，第 m 列的 R[m]=head，为了达到循环。最后，R[head]=1，L[head]=m。下面是 C++ 代码：

```
void init()
{
        for(int i=0;i<=n;i++)
        {
                U[i]=D[i]=i;L[i]=i-1;R[i]=i+1;
        }
        L[0]=n;R[n]=0;
        topRow=n+1;
}
```

由于矩阵多半是稀疏的（否则多半无解），而只有 1 有用，所以我们再用一个数组 newrow[]和 sizenew 来记录新的这一行的 1 的位置，比如对于一行（1,1,0,0,1），我们就记录（1,2,5）。添加代码也不难实现，只需在列的表中添加节点，并处理好本行节点内的关系即可，最后不要忘了++S[newrow[i]]，并在 Information 中记录下本行的信息。代码如下（只记录了所在行（row），没有具体记录信息）：

```
void addrow(int r)
{
        int first=top,c;
        for(int i=0;i<sizenew;i++)
        {
                c=newrow[i];
                L[top]=top-1;           R[top]=top+1;
                D[top]=c;               U[top]=U[c];
                D[U[c]]=top;            U[c]=top;
                row[top]=r;             C[top]=c;
                ++S[c];
                ++top;
        }
        L[first]=top-1;R[top-1]=first;
}
```

在有向链表中，假设我们要从左到右迭代依次访问，可以用这样的语句：

for(int i=R[x];i!=x;i=R[i])

必须要解释的是，x 节点必须是头或者提前处理，因为这样的语句不会访问到 x。这样，通过把 R 换成 U、D、L 等方向，就能实现各个方向的迭代。为了让代码简洁易懂，更为了避免“笔误”，我们利用了 C++的宏定义：

#define For(i,A,s) for(int i=A[s];i!=s;i=A[i])

这样就能用 For(i,R,x)代替前一句语句，而且代码可读性大大增加，因为三个要素：**循环变量名、方向和起始点**统统齐全了。现在我们已经画好了妆，穿上了舞鞋和裙子，准备开始美丽的舞蹈。回看 X 算法：

1、按照搜索顺序，选取当前状态中还没被删除的一行

2、删除并扫描这一行，将这一行中带 1 的列删除

3、扫描所有行，如果当前行和选择的这一行的同一列都拥有 1，则删除当前行

4、如果矩阵为空，则解找到，推出。

5、否则，递归处理这个删除了一部分的矩阵（DFS）

我们可以一步一步的用 Dancing Links 解决。我们一步步分析算法：（在分析中，我们假设（事实上满足）无论何时此 Dancing Links 符合规则。

1、按照搜索顺序，选取当前状态中还没被删除的一行

首先是搜索顺序。我们采用**由列找行**的方式，即确定一个列，深搜它为 1 的行。我们有两种选择方式。如果要求字典序最小（或最大），则从 R[head]（或 L[head]）开始搜索。如果只需寻找有无解，则之前的 S[]数组能帮助我们，从 1 最少的列开始搜，这样就能让搜索树上层叶子减少。

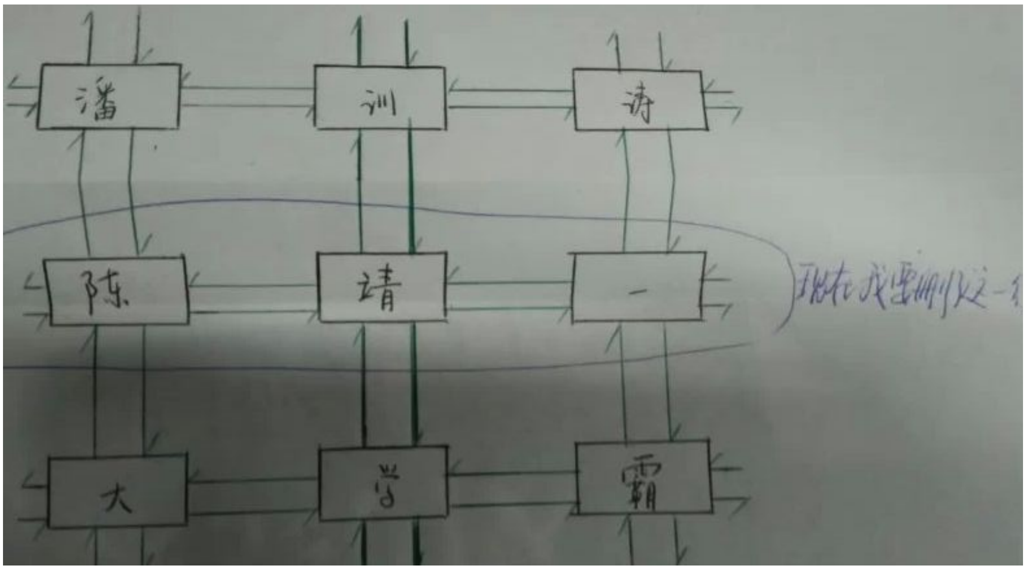
2、删除并扫描这一行，将这一行中带 1 的列删除

这时候我们应该确认一个删除整行（列）的方式。有人说可以这么定义一个删除（或恢复）点的函数：

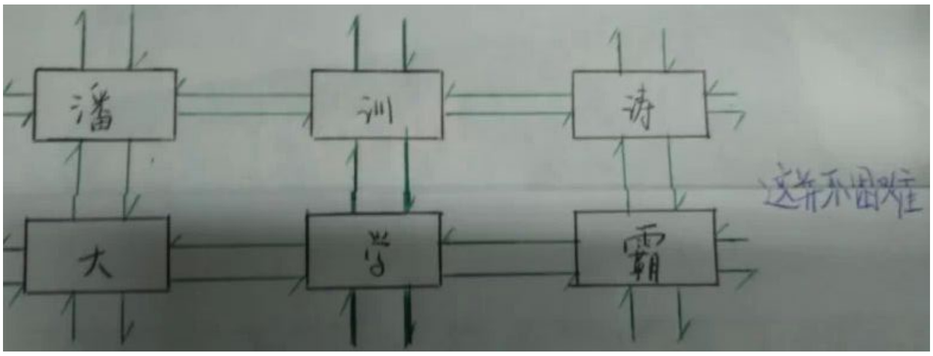
```
void removePoint(int x)
{
        R[L[x]]=R[x];
        L[R[x]]=L[x];
        U[D[x]]=U[x];
        D[U[x]]=D[x];
        --S[x];
}

void recoverPoint(int x)
{
        R[L[x]]= L[R[x]]= U[D[x]]=D[U[x]]=x;
        ++S[x];
}
```

然后再迭代。虽然这样看起来很好，但其实暗藏 bug（当一整行都被删掉，会导致恢复的时候有一些难以预料的麻烦），而且浪费了很多时间。为什么呢？试想，如果一整条链（行或列），**没有任何一个节点和外界有连通**，那么即使链的内部是连通的，也**不会被迭代访问到**。就像这样，当删除一个行的时候：



原本 pxt 和 cjy 都是学霸，后来 cjy 成了学神，如何把 cjy 删掉呢？虽然可以用涂改带分别把三个字的每一条边和字一起涂掉，但由于这是纸张，我们还有一种更聪明的方法：



看到了什么？我们只需要把 cjy 上下的节点连起来即可，根本不需要把 c、j、y 三个字拆开。这不仅保存了 cjy 的完整性，体现了对他的尊敬，而且更在恢复（虽然在这个例子中 cjy 不会退化成学霸）中，也只需要连接一些节点，而不需要再处理 cjy 三个字之间的关系（事实上这并不好处理）。概括这个好笑且真实的示例，我们就能得到下面的道理：

如果要完全删除一行（列），只需要在竖直（水平）方向删掉每一个节点。

就这样我们可以做好一切。由于不完全删除，被删除的点自己的指针还指着原来的点，所以我们可以先删除掉一整行，然后再处理为 1 的列。

3、扫描所有行，如果当前行和选择的这一行的同一列都拥有 1，则删除当前行

这步大可以在处理这一行的时候做完。我们以当前点为起点，竖直方向遍历并在水平方向删除一整列上的所有节点。由于前面美妙的性质，我们有如下好处：

（1）当前点无需（且禁止）被删除第二次，而迭代遍历方式恰好符合这一点

（2）由于是循环链表，这样会把列头指针也一并删掉

在删除的时候记得更新列的大小，但如果删掉一整列则根本不用进行更改。恢复的时候，为了符合原则，我们用和删除相反的方向进行恢复操作，先恢复行，再恢复每个列。

删除和恢复一整列的正确代码如下：

```
void remove(int c)
{
	L[R[c]]=L[c];
	R[L[c]]=R[c];
	For(i,D,c)
		For(j,R,i)
		{
			U[D[j]]=U[j];
			D[U[j]]=D[j];
			--S[C[j]];
		}
}
void restore(int c)
{
	For(i,U,c)
		For(j,L,i)
		{
			U[D[j]]=j;
			D[U[j]]=j;
			++S[C[j]];
		}
	L[R[c]]=c;
	R[L[c]]=c;
}
```

在搜索过程中，我们是先选择一列，所以我们先把这一列删掉。注意虽然这时列头节点还在，但后面一定有机会把它删掉。接下来就是枚举行时对于每个节点调用一次 remove(i)即可，调用代码如下：

```
remove(c);
For(i,D,c)
{
	For(j,R,i)remove(C[j]);
	/**
	Continue Searching;
	if completed exit;
	*/
	For(j,L,i)restore(C[j]);
}
restore(c);
```

调用时，由于循环的优美性质，一整行不会被完全删除，而是按照上方的规则，符合了时间效率。注意调用的过程。Restore 是在回溯的时候调用的。

4、如果矩阵为空，则解找到，推出。

现在这个判定就简单了：如果 head 这一行只剩 head 一个节点，操作就完成辣！

5、否则，递归处理这个删除了一部分的矩阵（DFS）

如何递归调用是一个小问题，这和设计整个 DFS 函数有关。如果只是验证有没有解，它可以没有参数；如果要输出解，就要开一个栈记录解，同样可以没有参数；但如果要有更多用途，比如《靶形数独》中要输出最大值，参数就要传递当前的总和（Information 也要做修改，之后将作为例题讲解）。至此，舞蹈的谱子已经完成了，下面是 DFS 函数 dance()的完整代码：

```
bool dance()
{
	if(R[0]==0)return 1;
	int c=R[0];
	For(i,R,0)if(S[i]<S[c])c=i;
	remove(c);
	For(i,D,c)
	{
		For(j,R,i)remove(C[j]);
		if(dance())return 1;
		For(j,L,i)restore(C[j]);
	}
	restore(c);
	return 0;
}
```

再次提醒：这只是最基本的函数，仅仅只能验证可解性，接下来的例题，我将详细讲解如何运用这美妙的模型。我还会提到一些分析过程。一旦可以使用 DLX 模型，就最好使用，因为它已经具有了最好最完整的优化，没有什么优化能比链表更优美、更动人。

3 表演：靶形数独

题目：FZYZOJ[1205]

数独和 n 皇后都是 DLX 的经典运用。N 皇后较简单，数独较繁，但仍可以做。

我们想，01 矩阵的哲学意义，就是每一包给你几个东西，最后你每个东西必须有且只能有 1 个。这种“有且只有”的关系如何推倒而出呢？行代表包，列代表物品。我们这么想：

（1）每一行代表第 i 行 j 列填的数为 z，所以包的价值 information 即为 score(i,j)×z，也可以哈希下 i、j、z

（2）物品 1：如果第 i 行 j 列有一个数字，则得到这个物品

（3）物品 2：如果 i 行有数字 z，则得到这个物品

（4）物品 3：如果 j 列有数字 z，则得到这个物品

（5）物品 4：如果 i 行 j 列所在的方块有数字 z，则得到这个物品

这样我们就有了这么一种构造行的方法。物品 1～4 每种都有 81 种不同的物品（共有 81 个格子，每行、列、块有 9 个位置，每个位置有 9 种填法）共 324 列，而总共有 81（格子数）×9（每格填法）=729 行，这么大的矩阵，也只有 DLX 能实现了。而且 DLX 比加了大量优化的搜索还要快上 3 倍以上。

首先读入数据。如果一个格子已经填了数，那么直接插入这个行，不要插入其他选项。由于物品 1 的限制，这一行必定会选到，而且根据搜索序，这一行会第一个被决策。如果没有填数，填入 9 个行，分别表示这

个格子填 1 ~ 9 的结果。

搜索时定义 dance(sum)，表示当前总和为 sum。每次 DFS 的时候，把加上这一步的得分，然后继续操作。如果搜索完成，不要 return，更新答案，然后回溯。完整代码如下：

```
#include<cstdio>
#include<algorithm>
#define maxn 4000
#define For(i,A,s) for(int i=A[s];i!=s;i=A[i])
using namespace std;
template<class T>void read(T&aa)
{
    int bb;
       char ch;
       while(ch=getchar(),(ch<'0'||ch>'9')&&ch!='-');
    ch=='-'?(bb=1,aa=0):(aa=ch-'0',bb=0);
    while(ch=getchar(),ch>='0'&&ch<='9')aa=aa*10+ch-'0';if(bb)aa=-aa;
}
int n,top,ans=-1;
int U[maxn],D[maxn],R[maxn],L[maxn],C[maxn],row[maxn],newrow[maxn],S[maxn],sizenew;
int score[9][9]=
{
    {6,6,6,6,6,6,6,6,6},
    {6,7,7,7,7,7,7,7,6},
    {6,7,8,8,8,8,8,7,6},
    {6,7,8,9,9,9,8,7,6},
    {6,7,8,9,10,9,8,7,6},
    {6,7,8,9,9,9,8,7,6},
    {6,7,8,8,8,8,8,7,6},
    {6,7,7,7,7,7,7,7,6},
    {6,6,6,6,6,6,6,6,6}
};
int encode(int a,int b,int c)
{
    return a*81+b*9+c+1;
}
void decode(int code,int& a,int& b,int& c)
{
    code--;
    c=code%9;code/=9;
    b=code%9;code/=9;
    a=code;
}
void init()
{
    for(int i=0;i<=n;i++)
    {
```

```
        U[i]=D[i]=i;L[i]=i-1;R[i]=i+1;
    }
    L[0]=n;R[n]=0;
    top=n+1;
}
void addrow(int r)
{
    int first=top,c;
    for(int i=0;i<sizenew;i++)
    {
        c=newrow[i];
        L[top]=top-1;           R[top]=top+1;
        D[top]=c;           U[top]=U[c];
        D[U[c]]=top;            U[c]=top;
        row[top]=r;             C[top]=c;
        ++S[c];
        ++top;
    }
    L[first]=top-1;R[top-1]=first;
}
void remove(int c)
{
    L[R[c]]=L[c];
    R[L[c]]=R[c];
    For(i,D,c)
        For(j,R,i)
        {
            U[D[j]]=U[j];
            D[U[j]]=D[j];
            --S[C[j]];
        }
}
void restore(int c)
{
    For(i,U,c)
        For(j,L,i)
        {
            U[D[j]]=j;
            D[U[j]]=j;
            ++S[C[j]];
        }
    L[R[c]]=c;
    R[L[c]]=c;
}
void dance(int sum)
{
```

```
	if(R[0]==0)
	{
		ans=max(sum,ans);
//printf("%d\n",ans);
		return;
	}
	int c=R[0];
	int aa,bb,cc;
	For(i,R,0)if(S[i]<S[c])c=i;
	remove(c);
	For(i,D,c)
	{
		For(j,R,i)remove(C[j]);
		dance(sum+row[i]);
		For(j,L,i)restore(C[j]);
	}
	restore(c);
}
int main()
{
	int val;
	n=324;
	init();
	for(int i=0;i<9;++i)
		for(int j=0;j<9;++j)
		{
			read(val);
			if(val)
			{
				int v=val-1;
				sizenew=0;
				newrow[sizenew++]=encode(0,i,j);
				newrow[sizenew++]=encode(1,i,v);
				newrow[sizenew++]=encode(2,j,v);
				newrow[sizenew++]=encode(3,(i/3)*3+j/3,v);
				addrow(score[i][j]*(v+1));
			}
			else for(int v=0;v<=8;++v)
			{
				sizenew=0;
				newrow[sizenew++]=encode(0,i,j);
				newrow[sizenew++]=encode(1,i,v);
				newrow[sizenew++]=encode(2,j,v);
				newrow[sizenew++]=encode(3,(i/3)*3+j/3,v);
				addrow(score[i][j]*(v+1));
			}
```

```
        }
    dance(0);
    printf("%d\n",ans);
    return 0;
}
```

评测记录 R79850 - FZYZ Online Judge

用户：f　题目：P1205　总耗时：1.931s　编译器：G++　状态：Accepted

代码及编译信息

测试点	满分	得分	状态	耗时	内存占用
1	5	5	Accepted	0.000000s	276K
2	5	5	Accepted	0.004999s	272K
3	5	5	Accepted	0.001999s	272K
4	5	5	Accepted	0.001998s	272K
5	5	5	Accepted	0.007997s	276K
6	5	5	Accepted	0.000999s	272K
7	5	5	Accepted	0.001999s	272K
8	5	5	Accepted	0.001999s	276K
9	5	5	Accepted	0.017997s	276K
10	5	5	Accepted	0.007998s	272K
11	5	5	Accepted	0.053991s	272K
12	5	5	Accepted	0.002999s	280K
13	5	5	Accepted	0.108982s	276K
14	5	5	Accepted	0.143978s	276K
15	5	5	Accepted	0.122981s	276K
16	5	5	Accepted	0.208967s	276K
17	5	5	Accepted	0.416936s	280K
18	5	5	Accepted	0.246961s	272K
19	5	5	Accepted	0.234964s	280K
20	5	5	Accepted	0.341948s	280K

返回

时间和内存都十分理想。

至此，美丽的 DLX 算法完全讲解完毕。这便是整个十字链表（DL）的精髓所在。至于 n 皇后问题，可自行搜索，难度不大，比本题更易想到。

4 NOI 加油