

# 状态压缩

## Abstract

信息学发展势头迅猛，信息学奥赛的题目来源遍及各行各业，经常有一些在实际应用中很有价值的问题被引入信息学并得到有效解决。然而有一些问题却被认为很可能不存在有效的(多项式级的)算法，本文以对几个例题的剖析，简述状态压缩思想及其应用。

## Keywords

状态压缩、集合、Hash、NPC

## Content

### Introduction

作为 OIers，我们不同程度地知道各式各样的算法。这些算法有的以  $O(\log n)$  的复杂度运行，如二分查找、欧几里德 GCD 算法(连续两次迭代后的余数至多为原数的一半)、平衡树，有的以  $O(\sqrt{n})$  运行，例如二级索引、块状链表，再往上有  $O(n)$ 、 $O(n^p \log^q n)$ ……大部分问题的算法都有一个多项式级别的时间复杂度上界<sup>1</sup>，我们一般称这类问题<sup>2</sup>为 **P** 类(deterministic Polynomial-time)问题，例如在有向图中求最短路径。然而存在几类问题，至今仍未被很好地解决，人们怀疑他们根本没有多项式时间复杂度的算法，它们是 **NPC(NP-Complete)** 和 **NPH(NP-Hard)** 类，例如问一个图是否存在哈密顿圈(NPC)、问一个图是否不存在哈密顿圈(NPH)、求一个完全图中最短的哈密顿圈(即经典的 Traveling Salesman Problem 货郎担问题, NPH)、在有向图中求最长(简单)路径(NPH)，对这些问题尚不知有多项式时间的算法存在。P 和 NPC 都是 **NP(Non-deterministic Polynomial-time)** 的子集，NPC 则代表了 NP 类中最难的一类问题，所有的 NP 类问题都可以在多项式时间内归约到 NPC 问题中去。NPH 包含了 NPC 和其他一些不属于 NP(也更难)的问题(即 NPC 是 NP 与 NPH 的交集)，NPC 问题的最优化版本一般是 NPH 的，例如问一个图是否存在哈密顿圈是 NPC 的，但求最短的哈密顿圈则是 NPH 的，原因在于我们可以在多项式时间内验证一个回路是否真的是哈密顿回路，却无法在多项式时间内验证其是否是最短的，NP 类要求能在多项式时间内验证问题的一个解是否真的是一个解，所以最优化 TSP 问题不是 NP 的，而是 NPH 的。存在判定性 TSP 问题，它要求判定给定的完全图是否存在权和小于某常数  $v$  的哈密顿圈，这个问题的解显然可以在多项式时间内验证，

<sup>1</sup> 请注意，大 O 符号表示上界，即  $O(n)$  的算法可以被认为是  $O(n^2)$  的， $O(n^p \log^q n)$  可以被认为是  $O(n^{p+1})$  的。

<sup>2</sup> 在更正式的定义中，下面提到的概念都只对判定性问题或问题的判定版本才存在。Levin 给出了一个适用于非判定问题的更一般的概念，但他的论文比 Cook 的晚发表 2 年。

因此它是 NP 的，更精确地说是 NPC 的。<sup>1</sup>

如上所述，对于 NPC 和 NPH 问题，至今尚未找到多项式时间复杂度的算法。然而它们的应用又是如此的广泛，我们不得不努力寻找好的解决方案。毫无疑问，对于这些问题，使用暴力的搜索是可以得到正确的答案的，但在信息学竞赛那有限的时间内，很难写出速度可以忍受的暴力搜索。例如对于 TSP 问题，暴力搜索的复杂度是  $O(n!)$ ，如此高的复杂度使得它对于高于 10 的数据规模就无能为力了。那么，有没有一种算法，它可以在很短的时间内实现，而其最坏情况下的表现比搜索好呢？答案是肯定的——**状态压缩(States Compression, SC)**。

作为对下文的准备，这里先为使用 Pascal 的 OIers 简要介绍一下 C/C++ 样式的位运算(bitwise operation)。

### 一、基本运算符

名称	C/C++ 样式	Pascal 样式	简记法则
按位与 (bitwise AND)	<b>&amp;</b>	and	全一则一 否则为零
按位或 (bitwise OR)		or	有一则一 否则为零
按位取反 (bitwise NOT)	~	not	是零则一 是一则零
按位异或 (bitwise XOR)	^	xor	不同则一 相同则零

以上各运算符的优先级从高到低依次为： $\sim, \&, ^, |$

### 二、特殊应用

#### a) and:

- i. 用以取出一个数的某些二进制位
- ii. 取出一个数的最后一个 1(lowbit)<sup>2</sup>:  $x \& -x$

#### b) or : 用以将一个数的某些位设为 1

#### c) not: 用以间接构造一些数: $\sim 0 = 4294967295 = 2^{32} - 1$

#### d) xor:

- i. 不使用中间变量交换两个数:  $a = a \wedge b; b = a \wedge b; a = a \wedge b;$
- ii. 将一个数的某些位取反

有了这些基础，就可以开始了。

<sup>1</sup> 不应该混淆 P、NP、NPC、NPH 的概念。这里只是粗略介绍，详见关于算法分析的书籍，这会使新手读者的理论水平上一个台阶。弄不明白也没关系，基本不影响对本文其他部分的理解^\_^

<sup>2</sup> 具有同样作用的还有  $(x-1) \& x \wedge x$ ，这个的道理更容易理解。lowbit 在树状数组(某种数据结构)中可以用到，这里不再单独介绍，感兴趣的可以参阅各牛的论文，或者加我 QQ。建议掌握，否则可能会看不懂我的部分代码。

## Getting Started

我们暂时避开状态压缩的定义，先来看一个小小的例题。

### 【引例】<sup>1</sup>

在  $n*n(n \leq 20)$  的方格棋盘上放置  $n$  个车(可以攻击所在行、列)，求使它们不能互相攻击的方案总数。

### 【分析】

这个题目之所以是作为引例而不是例题，是因为它实在是个非常简单的组合学问题：我们一行一行放置，则第一行有  $n$  种选择，第二行  $n-1$ ，……，最后一行只有 1 种选择，根据乘法原理，答案就是  $n!$ 。这里既然以它作为状态压缩的引例，当然不会是为了介绍组合数学。我们下面来看另外一种解法：**状态压缩递推 (States Compressing Recursion, SCR)**。

我们仍然一行一行放置。取棋子的放置情况作为状态，某一行如果已经放置棋子则为 1，否则为 0。这样，一个状态就可以用一个最多 20 位的二进制数<sup>2</sup>表示。例如  $n=5$ , 第 1、3、4 列已经放置，则这个状态可以表示为 01101(从右到左)。设  $f[s]$  为达到状态  $s$  的方案数，则可以尝试建立  $f$  的递推关系。

考虑  $n=5, s=01101$ 。这个状态是怎么得到的呢？因为我们是一行一行放置的，所以当达到  $s$  时已经放到了第三行。又因为一行能且仅能放置一个车，所以我们知道状态  $s$  一定来自：

- ①前两行在第 3、4 列放置了棋子(不考虑顺序，下同)，第三行在第 1 列放置；
- ②前两行在第 1、4 列放置了棋子，第三行在第 3 列放置；
- ③前两行在第 1、3 列放置了棋子，第三行在第 4 列放置。

这三种情况互不相交，且只可能有这三种情况。根据加法原理， $f[s]$  应该等于这三种情况的和。写成递推式就是：

$$f[01101] = f[01100] + f[01001] + f[00101]$$

根据上面的讨论思路推广之，得到引例的解决办法：

$$f[0] = 1$$

$$f[s] = \sum f[s \wedge 2^i]$$

其中  $s \in [0 \dots 01, 1 \dots 11]$ ， $s$  的右起第  $i+1$  位为 1。<sup>3</sup>

反思这个算法，其正确性毋庸置疑(可以和  $n!$  对比验证)。但是算法的时间复杂度为  $O(n2^n)$ ，空间复杂度  $O(2^n)$ ，是个指数级的算法，比循环计算  $n!$  差了好多，它有什么优势？较大的推广空间。<sup>4</sup>

## Sample Problems

### 【例 1】<sup>5</sup>

在  $n*n(n \leq 20)$  的方格棋盘上放置  $n$  个车，某些格子不能放，求使它们不能互

<sup>1</sup> 本文所有例题的 C++ 代码均可以在附件中找到。

<sup>2</sup> 方便、整齐起见，本文中不在数的后面标明进制。

<sup>3</sup> 考虑上节介绍的位运算的特殊应用，可以更精巧地实现。

<sup>4</sup> 还有一个很...的用处，即对新手说：“来看看我这个计算  $n!$  的程序，连这都看不懂就别 OI 了~”

<sup>5</sup> 题目来源：经典组合学问题。

相攻击的方案总数。

### 【分析】

对于这个题目，如果组合数学学得不够扎实，你是否还能一眼看出解法？应该很难。对于这个题目，确实存在数学方法(容斥原理)，但因为和引例同样的理由，这里不再赘述。

联系引例的思路，发现我们并不需要对算法进行太大的改变。引例的算法是在枚举当前行(即  $s$  中 1 的个数，设为  $r$ )的放置位置(即枚举每个 1)，而对于例 1，第  $r$  行可能存在无法放置的格子，怎么解决这个问题呢？枚举 1 的时候判断一下嘛！事实的确是这样，枚举 1 的时候判断一下是否是不允许放置的格子即可。

但是对于  $n=20$ ， $O(n2^n)$  的复杂度已经不允许我们再进行多余的判断。所以实现这个算法时应该应用一些技巧。对于第  $r$  行，我们用  $a[r]$  表示不允许放置的情况，如果某一位不允许放置则为 1，否则为 0，这可以在读入数据阶段完成。运算时，对于状态  $s$ ，用  $tmps=s \wedge a[r]$  来代替  $s$  进行枚举，即不枚举  $s$  中的 1 转而枚举  $tmps$  中的 1。因为  $tmps$  保证了无法放置的位为 0，这样就可以不用多余的判断来实现算法，代码中只增加了计算  $a$  数组和  $r$  的部分，而时间复杂度没有太大变化。

这样，我们直接套用引例的算法就使得看上去更难的例 1 得到了解决。你可能会说，这题用容斥原理更快。没错，的确是这样。但是，容斥原理在这题上只有当棋盘为正方形、放入的棋子个数为  $n$ 、且棋盘上禁止放置的格子较少时才有简单的形式和较快的速度。如果再对例 1 进行推广，要在  $m*n$  的棋盘上放置  $k$  个车，那么容斥原理是无能为力的，而 SCR 算法只要进行很少的改变就可以解决问题<sup>1</sup>。这也体现出了引例中给出的算法具有很大的扩展潜力。

棋盘模型是状态压缩最好的展示舞台之一。下面再看几个和棋盘有关的题目。

### 【例 2】<sup>2</sup>

给出一个  $n*m$  的棋盘( $n, m \leq 80, n*m \leq 80$ )，要在棋盘上放  $k(k \leq 20)$  个棋子，使得任意两个棋子不相邻。每次试验随机分配一种方案，求第一次出现合法方案时试验的期望次数，答案用既约分数表示。

### 【分析】

显然，本题中的期望次数应该为出现合法方案的概率的倒数，则问题转化为求出现合法方案的概率。而概率 =  $\frac{\text{合法方案数}}{\text{方案总数}}$ ，方案总数显然为  $C(n*m, k)$ ，则问题转化为求合法方案数。整理一下，现在的问题是：在  $n*m$  的棋盘上放  $k$  个棋子，求使得任意两个棋子不相邻的放置方案数。

这个题目的状态压缩模型是比较隐蔽的。观察题目给出的规模， $n, m \leq 80$ ，这个规模要想用 SC 是困难的，若同样用上例的状态表示方法(放则为 1，不放为 0)， $2^{80}$  无论在时间还是在空间上都无法承受。然而我们还看到  $n*m \leq 80$ ，这种给出数据规模的方法是不多见的，有什么玄机呢？能把状态数控制在可以承受的

<sup>1</sup> 如果这样改编，则状态的维数要增加，如有疑问可以参考下面的几个例子，这里不再赘述。

<sup>2</sup> 题目来源：经典问题改编。

范围吗? 稍微一思考, 我们可以发现:  $9*9=81>80$ , 即如果  $n, m$  都大于等于 9, 将不再满足  $n*m \leq 80$  这一条件。所以, 我们有  $n$  或  $m$  小于等于 8, 而  $2^8$  是可以承受的。我们假设  $m \leq n$  (否则交换, 由对称性知结果不变)  $n$  是行数  $m$  是列数, 则每行的状态可以用  $m$  位的二进制数表示。但是本题和例 1 又有不同: 例 1 每行每列都只能放置一个棋子, 而本题却只限制每行每列的棋子不相邻。但是, 上例中枚举当前行的放置方案的做法依然可行。我们用数组  $s[1..num]$ <sup>1</sup> 保存一行中所有的  $num$  个放置方案, 则  $s$  数组可以在预处理过程中用 DFS 求出, 同时用  $c[i]$  保存第  $i$  个状态中 1 的个数以避免重复计算。开始设计状态。如注释一所说, 维数需要增加, 原因在于并不是每一行只放一个棋子, 也不是每一行都要求有棋子, 原先的表示方法已经无法完整表达一个状态。我们用  $f[i][j][k]$  表示第  $i$  行的状态为  $s[j]$  且前  $i$  行已经放置了  $k$  个棋子<sup>2</sup> 的方案数。沿用枚举当前行方案的做法, 只要当前行的方案和上一行的方案不冲突即可, “微观”地讲, 即  $s[snum[i]]$  和  $s[snum[i-1]]$  没有同为 1 的位, 其中  $snum[x]$  表示第  $x$  行的状态的编号。然而, 虽然我们枚举了第  $i$  行的放置方案, 但却不知道其上一行( $i-1$ )的方案。为了解决这个问题, 我们不得不连第  $i-1$  的状态一起枚举, 则可以写出递推式:

$$f[0][1][0]=1;$$

$$f[i][j][k]=\sum f[i-1][p][k-c[j]]$$

其中  $s[1]=0$ , 即在当前行不放置棋子;  $j$  和  $p$  是需要枚举的两个状态编号, 且要求  $s[j]$  与  $s[p]$  不冲突, 即  $s[j] \& s[p]=0$ 。<sup>3</sup>

当然, 实现上仍有少许优化空间, 例如第  $i$  行只和第  $i-1$  行有关, 可以用滚动数组节省空间。

有了合法方案数, 剩下的问题就不是很困难了, 需要注意的就只有  $C(n*m, k)$  可能超出 64 位整数范围的问题, 这可以通过边计算边用 GCD 约分来解决, 具体可以参考附件中的代码。这个算法时间复杂度  $O(n*pn*num^2)$ , 空间复杂度(滚动数组)  $O(pn*num)$ , 对于题目给定的规模是可以很快出解的。

通过上文的例题, 读者应该已经对状态压缩有了一些感性的认识。下面这个题目可以作为练习。

### 【例 3】<sup>4</sup>

在  $n*n(n \leq 10)$  的棋盘上放  $k$  个国王(可攻击相邻的 8 个格子), 求使它们无法互相攻击的方案数。

### 【分析】

其实有了前面几个例子的分析, 这个题目应该是可以独立解决的。不过既然确实有疑问, 那我们就来分析一下。

<sup>1</sup> 运用简单的组合数学知识可以求出: 在格数为  $m$  的一行上放棋子且相邻两个棋子中间的空格不能少于  $d$  的  $num=g[m+d+1]$ , 其中  $g[i]=1 (i=1..d+1); g[j]=g[j-d-1]+g[j-1] (j>d)$ 。对于本题,  $num=144$ 。此式在下文也有应用。

<sup>2</sup> 为了避免歧义, 下文中用  $pn$  代替原题中的  $k$ 。

<sup>3</sup> 请读者停下来仔细揣摩这个递推式, 否则可能影响对本文后面内容的理解!

<sup>4</sup> 题目来源: SGU.223 《Little Kings》

首先,你应该能想到将一行的状态 DFS 出来(如果不能,请返回重新阅读,谢谢),仍然设为  $s[1..num]$ ,同时仍然设有数组  $c[1..num]$ 记录状态对应的 1 的个数。和例 2 相同,仍然以  $f[i][j][k]$ 表示第  $i$  行状态为  $s[j]$ ,且前  $i$  行已经放置了  $k$  个棋子的方案数。递推式仍然可以写作:

$$f[0][1][0]=1;$$

$$f[i][j][k]=\sum f[i-1][p][k-c[j]]$$

其中仍要求  $s[j]$ 和  $s[p]$ 不冲突。

可是问题出来了:这题不但要求不能行、列相邻,甚至不能对角线相邻! $s[j]$ 、 $s[p]$ 不冲突怎么“微观地”表示呢?其实,稍微思考便可以得出方法:用  $s[p]$ 分别和  $s[j]$ 、 $s[j]*2$ 、 $s[j]/2$ 进行冲突判断即可,原理很显然。解决掉这唯一的问题,接下来的工作就没有什么难度了。算法复杂度同例 2。

下一个例题是状态压缩棋盘模型的经典题目,希望解决这个经典的题目能够增长你的自信。

#### 【例 4】<sup>1</sup>

给出一个  $n*m(n\leq 100,m\leq 10)$ 的棋盘,一些格子不能放置棋子。求最多能在棋盘上放置多少个棋子,使得每一行每一列的任两个棋子间至少有两个空格。

#### 【分析】<sup>2</sup>

显然,你应该已经有 DFS 搜出一行可能状态的意识了(否则请重新阅读之前的内容 3 遍,谢谢),依然设为  $s[1..num]$ ,依旧有  $c[1..num]$ 保存  $s$  中 1 的个数,依照例 1 的预处理搞定不能放置棋子的格子。

问题是,这个题目的状态怎么选?继续像例 2、3 那样似乎不行,原因在于棋子的攻击范围加大了。但是我们照葫芦画瓢:例 2、3 的攻击范围只有一格,所以我们的状态中只需要有当前行的状态即可进行递推,而本题攻击范围是两格,因此增加一维来表示上一行的状态。用  $f[i][j][k]$ 表示第  $i$  行状态为  $s[j]$ 、第  $i-1$  行状态为  $s[k]$ 时前  $i$  行至多能放置的棋子数,则状态转移方程很容易写出:

$$f[i][j][k]=\max\{f[i-1][k][l]\}+c[j]$$

其中要求  $s[j]$ 、 $s[k]$ 、 $s[l]$ 互不冲突。

因为棋子攻击范围为两格,可以直观地想象到  $num$  不会很大。的确,由例 2 中得到的  $num$  的计算式并代入  $d=2$ 、 $m=10$ ,得到  $num=60$ 。显然算法时间复杂度为  $O(n*num^3)$ ,空间复杂度(滚动数组) $O(num^2)$ 。此算法还有优化空间。我们分别枚举了三行的状态,还需要对这三个状态进行是否冲突的判断,这势必会重复枚举到一些冲突的状态组合。我们可以在计算出  $s[1..num]$ 后算出哪些状态可以分别作为两行的状态,这样在 DP 时就不需要进行盲目的枚举。这样修改后的算法理论上比上述算法更优,但因为  $num$  本身很小,所以这样修改没有显著地减少运行时间。值得一提的是,本题笔者的算法虽然在理论上并不是最优<sup>3</sup>,但由于位运算的使用,截至 2 月 9 日,笔者的程序在 PKU OJ 上长度最短,速度第二快。

这个题目是国内比赛中较早出现的状态压缩题。它告诉我们状态压缩不仅可以像前几个例题那样求方案数,而且可以求最优方案,即状态压缩思想既可以应用到递推上(SCR),又可以应用到 DP 上(SCDP),更说明其有广泛的应用空间。

<sup>1</sup> 题目来源: NOI2001《炮兵阵地》; PKU.1185

<sup>2</sup> 读者应该独立思考一个小时后再看分析,它值得这样。

<sup>3</sup> 有种应用三进制的方法理论上可能更优,但因为手工转换进制效率远不如系统的位运算高,且无法使用位运算判断可行性,程序效率并不比文中的高,故不再赘述那种方法。下文的一些题目将用到多进制。

看了这么多棋盘模型应用状态压缩的实例，你可能会疑问，难道状态压缩只在棋盘上放棋子的题目中 useful？不是的。我们暂时转移视线，来看看状态压缩在其他地方的应用——**覆盖模型**。

### 【例 5】<sup>1</sup>

给出  $n*m$  ( $1 \leq n, m \leq 11$ ) 的方格棋盘，用  $1*2$  的长方形骨牌不重叠地覆盖这个棋盘，求覆盖满的方案数。

### 【分析】

这也是个经典的组合数学问题：多米诺骨牌完美覆盖问题(或所谓二聚物问题)。有很多关于这个问题的结论，甚至还有个专门的公式：如果  $m, n$  中至少有一个是偶数，则结果 =  $\prod_{i=1}^{m/2} \prod_{j=1}^{n/2} \left[ 4\cos^2\left(\frac{i*\pi}{m+1}\right) + 4\cos^2\left(\frac{j*\pi}{n+1}\right) \right]$ 。这个公式形式比较

简单，且计算的复杂度是  $O\left(\frac{m*n}{4}\right)$  的，很高效。但是这个公式内还有三角函数，且中学生几乎不可能理解，所以对我们能力的提高没有任何帮助。用 SCR 算法能较好地解决这个问题。

显然，如果  $n, m$  都是奇数则无解(由棋盘面积的奇偶性知)，否则必然有至少一个解(很容易构造出)，所以假设  $n, m$  至少有一个偶数，且  $m \leq n$  (否则交换)。我们依然像前面的例题一样把每行的放置方案 DFS 出来，逐行计算。用  $f[i][s]$  表示把前  $i-1$  行覆盖满、第  $i$  行覆盖状态为  $s$  的覆盖方案数。因为在第  $i$  行上放置的骨牌最多也只能影响到第  $i-1$  行，则容易得递推式：

$$f[0][1 \cdots 11] = 1$$

$$f[i][s_1] = \sum f[i-1][s_2]$$

其中  $(s_1, s_2)$  整体作为一个放置方案，可以把所有方案 DFS 预处理出来。下面讨论一下本题的一些细节。

首先讨论 DFS 的一些细节。对于当前行每一个位置，我们有 3 种放置方法：① 竖直覆盖，占据当前格和上一行同一列的格；② 水平覆盖，占据当前格和该行下一格；③ 不放置骨牌，直接空格。如何根据这些枚举出每个  $(s_1, s_2)$  呢？下面介绍两种方法：

第一种：

DFS 共 5 个参数，分别为： $p$ (当前列号)， $s_1, s_2$ (当前行和上一行的覆盖情况)， $b_1, b_2$ (上一列的放置对当前列上下两行的影响，影响为 1 否则为 0)。初始时  $s_1=s_2=b_1=b_2=0$ 。①  $p=p+1, s_1=s_1*2+1, s_2=s_2*2^2, b_1=b_2=0$ ；②  $p=p+1, s_1=s_1*2+1, s_2=s_2*2+1, b_1=1, b_2=0$ ；③  $p=p+1, s_1=s_1*2, s_2=s_2*2+1, b_1=b_2=0$ 。当  $p$  移出边界且  $b_1=b_2=0$  时记录此方案。

第二种：

观察第一种方法，发现  $b_2$  始终为 0，知这种方法有一定的冗余。换个更自然的方法，去掉参数  $b_1, b_2$ 。①  $p=p+1, s_1=s_1*2+1, s_2=s_2*2$ ；②  $p=p+2,$

<sup>1</sup> 题目来源：经典问题；PKU.2411 《Mondriaan's Dream》

<sup>2</sup> 注意！第  $i$  行的放置方案用到第  $i-1$  行的某格时， $s_2$  中该格应为 0！

$s1=s1*4+3$ ,  $s2=s2*4+3$ ; ③ $p=p+1$ ,  $s1=s1*2$ ,  $s2=s2*2+1$ 。当  $p$  移出边界时记录此方案。这样, 我们通过改变  $p$  的移动距离成功简化了 DFS 过程, 而且这种方法更加自然。

DFS 过程有了, 实现方法却还有值得讨论的地方。前面的例题中, 我们为什么总是把放置方案 DFS 预处理保存起来? 是因为不合法的状态太多, 每次都重新 DFS 太浪费时间。然而回到这个题目, 特别是当采用第二种时, 我们的 DFS 过程中甚至只有一个判断(递归边界), 说明根本没有多少不合法的方案, 也就没有必要把所有方案保存下来, 对于每行都重新 DFS 即可, 这不会增加运行时间却可以节省一些内存。

这个算法时间复杂度为多少呢? 因为 DFS 时以两行为对象, 每行  $2^m$ , 共进行  $n$  次 DFS, 所以是  $O(n*4^m)$ ? 根据“O”的上界意义来看并没有错, 但这个界并不十分精确, 也可能会使人误以为本算法无法通过  $1 \leq n, m \leq 11$  的测试数据, 而实际上本算法可以瞬间给出  $m=10, n=11$  时的解。为了计算精确的复杂度, 必须先算出 DFS 得到的方案数。

考虑当前行的放置情况。如果每格只有①③两个选择, 则应该有  $2^m$  种放置方案; 如果每格有①②③这 3 个选择, 且②中  $p$  只移动一格, 则应该有  $3^m$  种放置方案。然而现在的事实是: 每格有①②③这 3 个选择, 但②中  $p$  移动 2 格, 所以可以知道方案数应该在  $2^m$  和  $3^m$  之间。考虑第  $i$  列, 则其必然是: 第  $i-1$  列采用①③达到; 第  $i-2$  列采用②达到。设  $h[i]$  表示前  $i$  列的方案数, 则得到  $h[i]$  的递推式:

$$\begin{aligned} h[0] &= 1, h[1] = 2 \\ h[i] &= 2 * h[i-1] + h[i-2] \end{aligned}$$

应用组合数学方法<sup>1</sup>求得其通项公式  $h[m] = \frac{2+\sqrt{2}}{4}(1+\sqrt{2})^m + \frac{2-\sqrt{2}}{4}(1-\sqrt{2})^m$ 。注意到式子的第二项是多个绝对值小于 1 的数的乘积, 其对整个  $h[m]$  的影响甚小, 故略去, 得到方案数  $h[m] \approx 0.85 * 2.414^m$ , 符合  $2^m < h[m] < 3^m$  的预想。

因为总共进行了  $n$  次 DFS, 每次复杂度为  $O(h[m])$ , 所以算法总时间复杂度为  $O(n * h[m]) = O(n * 0.85 * 2.414^m)$ , 对  $m=10, n=11$  不超时也就不足为奇了。应用滚动数组, 空间复杂度为  $O(2^m)$ 。

对于本题, 我们已经有了公式和 SCR 两种算法。公式对于  $m * n$  不是很大的情况有效, SCR 算法在竞赛中记不住公式时对小的  $m, n$  有效。如果棋盘规模为  $n * m (m \leq 10, n \leq 2^{31}-1)$ , 则公式和 SCR 都会严重超时。有没有一个算法能在 1 分钟内解决问题呢<sup>2</sup>? 答案是肯定的, 它仍然用到 SC 思想。

此算法中应用到一个结论: 给出一个图的邻接矩阵  $G$ (允许有自环, 两点间允许有多条路径, 此时  $G[i][j]$  表示  $i$  到  $j$  的边的条数), 则从某点  $a$  走  $k$  步到某点

<sup>1</sup> 特征方程等方法。这里不再赘述。

<sup>2</sup> 对于如此大的数据, 高精度将成为瓶颈, 故这里不考虑高精度的复杂度; 我没有找到能 1s 出解的算法, 故延长时限到 1min。

**b** 的路径数为  $G^k[a][b]$ <sup>1</sup>。本结论实际上是通过递推得到的，简单证明如下：从 *i* 走 *k* 步到 *j*，必然是从 *i* 走 *k-1* 步到 *t*，然后从 *t* 走 1 步到 *j*，根据加法原理，即  $G[k][i][j]=\sum G[k-1][i][t]*G[t][j]$ <sup>2</sup>。是否感到这个式子很眼熟？没错，它和矩阵乘法一模一样，即： $G[k]=G[k-1]*G$ 。因为矩阵乘法满足结合律，又由  $G[1]=G$ ，所以我们得到结果： $G[k]=G^k$ 。

下面介绍这个算法。考虑一个有  $2^m$  个顶点的图，每个顶点表示一行的覆盖状态，即 SCR 算法中的 *s1* 或 *s2*。如果(*s1,s2*)为一个放置方案，则在 *s2* 和 *s1* 之间连一条(有向)边，则我们通过 DFS 一次可以得到一个邻接矩阵 *G*。仍然按照逐行放置的思想来考虑，则要求我们每行选择一个覆盖状态，且相邻两行的覆盖状态(*s1,s2*)应为一个放置方案，一共有 *n* 行，则要求选择 *n* 个状态，在图中考虑，则要求我们从初始(第 0 行)顶点(1...111)<sub>*n*</sub> 步走到(1...111)<sub>1</sub>，因为图的邻接矩阵是 DFS 出来的，每条边都对应一个放置方案，所以可以保证走的每条边都合法。因此，我们要求的就是顶点(1...111)<sub>1</sub> 走 *n* 步到达(1...111)<sub>*n*</sub> 的路径条数。由上面的结论知，本题的答案就是  $G^n[1\dots 111][1\dots 111]$ 。

现在的问题是，如何计算 *G* 的 *n* 次幂？连续  $O(n)$  次矩阵乘法吗？不可取。矩阵的规模是  $2^m*2^m$ ，一次普通矩阵乘法要  $O((2^m)^3)=O(8^m)$ ， $O(n)$  次就是  $O(n*8^m)$ ，比 SCR 算法还要差得多。其实我们可以借用二分思想。如果要计算  $3^8$  的值，你会怎么算呢？直接累乘将需要进行 7 次乘法。一种较简单的方法是： $3*3=3^2$ ， $3^2*3^2=3^4$ ， $3^4*3^4=3^8$ ，只进行了 3 次乘法，效率高了许多<sup>3</sup>。因为矩阵乘法满足结合律，所以可以用同样的思路进行优化。这种思路用递归来实现是非常自然的，然而，本题的矩阵中可能有  $2^{10}*2^{10}=2^{20}=1048576$  个元素，如果用(未经优化的)递归来实现，将可能出现堆栈溢出。不过庆幸的是我们可以非递归实现。用 *bin*[] 保存 *n* 的二进制的每一位，从最高位、矩阵 *G* 开始，如果 *bin*[当前位]为 0，则把上一位得到的矩阵平方；如果为 1，则平方后再乘以 *G*。这种方法的时间复杂度容易算出： $O(\log n)$  次矩阵乘法，每次  $O(8^m)$ ，共  $O(8^m*\log n)$ 。

这样对于  $m \leq 7$  就可以很快出解了。但对于  $m=n=8$ ，上述算法都需要 1s 才能出解，无法令人满意。此算法还有优化空间。

我们的矩阵规模高达  $2^m*2^m=4^m$ ，但是其中有用的(非 0 的)有多少个呢？根据介绍 SCR 算法时得到的 *h*[*m*] 计算式，*G* 中有  $4^m-h[m]=4^m-0.85*2.414^m$  个 0，对于  $m=8$ ，可以算出 *G* 中 98.5% 的元素都是 0，这是一个非常非常稀疏的矩阵，使用三次方的矩阵乘法有点大材小用。我们改变矩阵的存储结构，即第 *p* 行第 *q* 列的值为 *value* 的元素可以用一个三元组(*p,q,value*)来表示，采用一个线性表依行列顺序来存储这些非 0 元素。怎样对这样的矩阵进行乘法呢？观察矩阵乘法的计算式  $c[i][j]=\sum a[i][k]*b[k][j]$ ，当 *a*[*i*][*k*]或者 *b*[*k*][*j*]为 0 时，结果为 0，对结果没有影响，完全可以略去这种没有意义的运算。则得到计算稀疏矩阵乘法的算法：**枚举 *a* 中的非 0 元素，设为(*p,q,v1*)，在 *b* 中寻找所有行号为 *q* 的非 0 元素(*q,r,v2*)，并把 *v1\*v2* 的值累加到 *c*[*p*][*r*]<sup>4</sup>中。**这个算法多次用到一个操作：找出所有行号为 *q* 的元素，则可以给矩阵附加一个数组 *hp*[*q*]，表示线性表中第一个行号为 *q* 的元素的位置，若不存在则 *hp*[*q*]=0。算出二维数组 *c* 之后再对其进行压缩存储

<sup>1</sup> 上标表示乘幂。

<sup>2</sup> 因为 0 乘任何数都为 0，所以  $G[k-1][i][t]$ 或  $G[t][j]$ 是否为 0 对最后的和没有影响。

<sup>3</sup> 这里假设大整数间的乘法和小整数间的乘法耗费同样的时间，实际上并非这样。但对于大数据，我们只关心结果 mod 某个常数的值，所以可以进行这样的假设。

<sup>4</sup> 此时 *c* 是个  $2^m*2^m$  的二维数组

即可。此矩阵乘法的时间复杂度为  $O(\frac{a.\text{not}0 * b.\text{not}0}{2^m} + 4^m)$ ，在最坏情况下， $a.\text{not}0=b.\text{not}0=4^m$ ，算法的复杂度为  $O(8^m)$ ，和经典算法相同。因为矩阵非常稀疏，算法复杂度近似为  $O(4^m)$ <sup>1</sup>。考虑整个算法的时间复杂度： $O(\log n)$ 次矩阵乘法，每次  $O(4^m)$ ，则总时间复杂度  $O(\log n * 4^m)$ ，对于  $m \leq 9$  也可以很快出解了，对于  $m=10$ ， $n=2147483647$ ，此算法在笔者机器上(Pm 1.6G, 512M)运行时间少于 20s。虽然仍然不够理想，但已经不再超时数小时。此算法空间复杂度为  $O(\max\_not0 + 4^m)$ ，对于  $m=10$ ， $\max\_not0$  小于 190000。

以上给出了公式、SCR、矩阵乘方这 3 个算法，分别适用于不同的情况，本题基本解决。

读者应该已经注意到，覆盖模型和棋盘模型有很多共同点，譬如都是在矩形某些位置放入棋子(或某种形状的骨牌)来求方案数(如上例)或最优解(下面将会给出几个例题)。但不难看出，覆盖模型和棋盘模型又有着很大的不同：棋盘模型中，只要棋子所在的位置不被别的棋子攻击到即可，而覆盖模型中，棋子的攻击范围也不可以重叠。所以简单来说，**覆盖模型就是攻击范围也不能重叠的棋盘模型**。下面再给出一个与上例类似的覆盖模型的例题以加深印象。

### 【例 6】<sup>2</sup>

给出  $n * m$  ( $1 \leq n, m \leq 9$ ) 的方格棋盘，用  $1 * 2$  的矩形的骨牌和 L 形的( $2 * 2$  的去掉一个角)骨牌不重叠地覆盖，求覆盖满的方案数。

### 【分析】

观察题目条件，只不过是比例 5 多了一种 L 形的骨牌，因此很自然地顺着例 5 的思路走。本题中两种骨牌的最大长度和例 5 一样，所以仍然用  $f[i][s]$  表示把前  $i-1$  行覆盖满、第  $i$  行覆盖状态为  $s$  的覆盖方案数，得到的递推式和例 5 完全一样：

$$f[0][1 \cdots 11] = 1$$

$$f[i][s1] = \sum f[i-1][s2]$$

其中  $(s1, s2)$  整体作为一个放置方案。例 5 中有两种 DFS 方案，其中第二种实现起来较第一种简单。但在本题中，新增的 L 形骨牌让第二种 DFS 难以实现，在例 5 中看起来有些笨拙的第一种 DFS 方案在本题却可以派上用场。回顾第一种 DFS，我们有 5 个参数，分别为： $p$ (当前列号)， $s1$ 、 $s2$ (当前行和对应的上一行的覆盖情况)， $b1$ 、 $b2$ (上一列的放置对当前列两行的影响，影响为 1 否则为 0)。本题中，可选择的方案增多，故列表给出：

<sup>1</sup> 其实，虽然  $G$  为很稀疏的矩阵，但乘方后非 0 元素增多，不一定还是稀疏矩阵。但对于  $m=n=8$ ，计算结束后，矩阵中仍然有 80% 的 0，上述算法仍然高效。此处的复杂度是理想情况。

<sup>2</sup> 题目来源：SGU.131 《Hardwood Floor》

	覆盖情况	条件	参数 s 变化	参数 b 变化
1	0 0	无	$s1=s1*2+b1$	$b1=0$
	0 0		$s2=s2*2+1-b2$	$b2=0$
2	0 0	$b1=0$	$s1=s1*2+1$	$b1=1$
	1 1		$s2=s2*2+1-b2$	$b2=0$
3	1 0	$b1=0$	$s1=s1*2+1$	$b1=0$
	1 0	$b2=0$	$s2=s2*2$	$b2=0$
4	1 0	$b1=0$	$s1=s1*2+1$	$b1=1$
	1 1	$b2=0$	$s2=s2*2$	$b2=0$
5	0 1	$b1=0$	$s1=s1*2+1$	$b1=1$
	1 1		$s2=s2*2+1-b2$	$b2=1$
6	1 1	$b2=0$	$s1=s1*2+b1$	$b1=1$
	0 1		$s2=s2*2$	$b2=1$
7	1 1	$b1=0$	$s1=s1*2+1$	$b1=0$
	1 0	$b2=0$	$s2=s2*2$	$b2=1$

容易看出，在本题中此种 DFS 方式实现很简单。考虑其复杂度，因为 L 形骨牌不太规则，笔者没能找到一维的方案数的递推公式，因此无法给出复杂度的解析式。但当  $m=9$  时，算法共生成放置方案 79248 个，则对于  $n=m=9$ ，算法的复杂度为  $O(9*79248)$ ，可以瞬间出解。和上例一样，本题也没有必要保存所有放置方案，也避免 MLE。

那么，对于本题是否可以应用上题的矩阵算法呢？答案是肯定的，方法也类似，复杂度为  $O(8^m \cdot \log n)$ 。然而，对于本题却不能通过上题的稀疏矩阵算法加速，原因在于刚开始时矩阵中只有  $1-79248/4^9=70\%$  的 0，而运算结束后整个矩阵中只有 2 个 0，根本无法达到加速效果。

由于有上题的铺垫，基本相同的本题也很快得到了解决。

### 【例 7】<sup>1</sup>

给出  $n*m(n, m \leq 10)$  的方格棋盘，用  $1*r$  的长方形骨牌不重叠地覆盖这个棋盘，求覆盖满的方案数。

#### 【分析】

本题是例 5 的直接扩展。如果说例 5 中公式比 SCR 好，本题可以指出当公式未知时 SCR 依然是可行的算法。直接思考不容易发现方法，我们先考虑  $r=3$  时的情况。首先，此问题有解当且仅当  $m$  或  $n$  能被 3 整除。更一般的结论是：用  $1*r$  的骨牌覆盖满  $m*n$  的棋盘，则问题有解当且仅当  $m$  或  $n$  能被  $r$  整除。当  $r=2$  时，则对应于例 5 中  $m, n$  至少有一个是偶数的条件。此结论的组合学证明从略。

不同于例 5， $1*3$  骨牌的“攻击范围”已经达到了 3 行，可以想象例 5 中的表示方法已经无法正确表示所有状态，但其思路依然可以沿用。例 5 中用  $f[i][s]$  表示把前  $i-1$  行覆盖满、第  $i$  行覆盖状态为  $s$  的覆盖方案数，是因为当前行的放

<sup>1</sup> 题目来源：经典问题改编

置方案至多能影响到上一行，状态中只要包含一行的覆盖状态即可消除后效性。本题中当前行的放置方案可以影响到上两行，故可以想到应保存两行的覆盖状态以消除后效性，即增加一维，用  $f[i][s1][s2]$  表示把前  $i-2$  行覆盖满、第  $i-1$  行覆盖状态为  $s1$ 、第  $i$  行覆盖状态为  $s2$  的覆盖方案数。先不论上述表示方法是否可行(答案是肯定的)， $r=2$  时状态有 2 维， $r=3$  时有 3 维，推广后状态变量居然有  $r$  维，这样的方法不具有推广价值，而且空间复杂度也太高。

仔细分析上述方案，可以发现其失败之处。 $s1$  的第  $p$  位  $s1_p$  为 1(覆盖)时， $s2_p$  是不可能为 0 的(要求覆盖满)，则这两位( $s1_p, s2_p$ )的(0,0),(0,1),(1,0),(1,1)四种组合中有一种不合法，而上述状态表示方法却冗余地保存了这个组合，造成空间复杂度过高，也进行了多余的计算<sup>1</sup>。通过上面的讨论可以知道，每一位只有 3 种状态，引导我们使用三进制。我们用  $f[i][s]$  表示把前  $i-2$  行覆盖满、第  $i-1$  和第  $i$  行覆盖状态为  $s$  的覆盖方案数，但这里状态  $s$  不再是二进制，而是三进制： $s_p=0$  表示  $s1_p=s2_p=0$ ； $s_p=1$  表示  $s1_p=0, s2_p=1$ ； $s_p=2$  表示  $s1_p=s2_p=1$ 。这样，我们就只保留了必要的状态，空间和时间上都有了改进。当  $r=4$  时，可以类推，用四进制表示三行的状态， $r=5$  时用五进制……分别写出  $r=2,3,4,5$  的程序，进行归纳，统一 DFS 的形式，可以把  $DFS(p,s1,s2)$  分为两部分：① for  $i=0$  to  $r-1$  do  $DFS(p+1, s1*r+i, s2*r+(i+1)\text{mod } r)$ ；②  $DFS(p+r, s1*r^r+r^r-1, s2*r^r+r^r-1)$  问题解决。但 DFS 的这种分部方法是我们归纳猜想得到的，并没有什么道理，其正确性无法保证，我们能否通过某种途径证明它的正确性呢？仍以  $r=3$  为例。根据上面的讨论， $s_p$  取值 0 到 2，表示两行第  $p$  位的状态，但  $s_p$  并没有明确的定义。我们定义  $s_p$  为这两行的第  $p$  位从上面一行开始向下连续的 1 的个数，这样的定义可以很容易地递推，递推式同上两例没有任何改变，却使得上述 DFS 方法变得很自然<sup>2</sup>。

分析算法的时间复杂度，同例 5 一样需要用到 DFS 出的方案个数  $h[m]$ ，并且仿照例 5 中  $h[m]$  的递推式，我们可以得到<sup>3</sup>：

$$h[i]=r^i \quad (i=0\sim r-1)$$

$$h[j]=r*h[j-1]+h[j-r] \quad (j=r\sim m)$$

理论上我们可以根据递推式得到例 5 中那样的精确的通项公式，但需要解高于三次的方程，且根多数为复数，无法得到例 5 那样简单优美的表达式，这里仅给出  $r=2..5, m=10$  时  $h[m]$  的值依次为：5741, 77772, 1077334, 2609585, 9784376。对于推广后的问题，例 5 的矩阵算法依然可行，但此时空间将是一个瓶颈。

<sup>1</sup> 同样的，例 4 中的状态表示方法也有冗余，但因为合法方案很少，又因为位运算的使用，算法仍然高效。

<sup>2</sup> 请尝试写出  $r=3$  时用这种定义的递推式和相应的程序

<sup>3</sup> 其实从 DFS 过程可以很直观地看出