状态压缩DP

主讲老师：党东

【状态压缩DP】

动态规划作为一个高效的算法已经在多个领域得到了广泛应用，而基于状态压缩的动态规划则是一种特殊的动态规划。近年来在信息学竞赛中也涌现出越来越多这种类型的题目。本文将以三道典型例题向你简要介绍基于状态压缩的动态规划的基本方法和适用范围。

【状态压缩DP】

动态规划作为一种常见的算法，已在信息学竞赛中屡见不鲜，而各种状态的表示作为动态规划的核心内容也早已为各位选手所熟知。但现在经常有这么一种题，它利用我们常见的各种状态表示就会具有后效性，而用搜索又无法满足时间的要求。这时就需要我们**对状态的表示进行改进**，基于状态压缩的动态规划就出场了！

普通的动态规划一般是以一个个元素作为状态，或是以一整块作为状态，这样虽然直观清楚，但记录的信息太少，因此就会导致后效性的存在。而**状态压缩模型则是把许多个元素的状态（比如一行）作为一个整体来表示状态**，这样就为下面的计算提供了更多的信息，从而保证了正确性。但是由于是多个元素的集合体，所以要在计算机上储存就要把它压缩，具体的方法将在后面详细描述。

【状态压缩DP】

状态压缩必备知识点：位操作

位操作是一种速度非常快的基本运算，有左移、右移、与、或、非等运算。

- **左移**：左移一位，相当于某数乘以2，比如110左移1位，变为1100，由6变为12,表示为(110<<1)=1100。因此左移x位,相当于该数乘以2^x。
- **右移**：右移一位，相当于某数除以2，比如110右移1位，变为11，由6变为3,表示为(110>>1)=11。因此右移x位,相当于该数除以2^x。
- **与运算**：按位进行“与“运算，都为1时结果为1，否则为0。例如：101&110=100。
- **或运算**：按位进行“或”运算，都为0时结果为0，否则为1。例如：101|110=111。
- **非运算**：每位取反。例如~101=010。

【状态压缩DP】

若当前状态为s，对s有下列操作：

1.判断第i位是否为0：(S&1<<i)==0，意思是将1左移i位与S进行与运算后，看结果是否为0。

2.将第i位置1：S|1<<i，意思是将1左移i位与S进行或运算。

3.将第i位置0：S&~(1<<i)，意思是S与第i位为0，其余位为1的数进行与运算。

例如：S=1010101，i=5

S&1<<i：1010101&0100000=0

S |1<<i： 1010101|0100000=1110101

S&~(1<<i)：1010101&1011111=1010101

【例1】骑士(Sgu223) --2772

【问题描述】

在n*n(1<=n<=10)的棋盘上放k(0<=k<=n*n)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总数。

【输入格式】

输入有多组方案，每组数据只有一行为两个整数n和k。

【输出格式】

每组数据一行为方案总数，若不能够放置则输出0。

【输入样例】	【输出样例】
3 2	16
4 4	79

【例1】骑士(Sgu223) --2772

【问题分析】

由问题很容易联想起经典的“八皇后”问题，似乎就是“皇后”变成了“国王”，而且格子范围似乎也差不多，所求问题也一样。那么这个问题也能用搜索解决吗？

可稍加分析可知搜索是很难胜任的，因为国王的数目可以是很大，加上它与“八皇后”问题的一个本质上的不同便是每个国王只影响周围的一个格子，所以剪枝条件也很少，指数级别的搜索是无法在时限内出解的。

那么一般的动态规划能解决吗？典型的二维DP，F[I,J]似乎无法很好地把状态表示出来，因此我们只能考虑状态压缩的动态规划。

首先我们要注意到这题的关键——每个国王只影响周围八个方向的一个格子，它虽然否定了搜索，却给状态压缩带来了无限生机！

我们改变之前动态规划的思维方式，一行一行地摆放国王，当我们摆放第I行时，这一行只会和前后一行的互相影响，而这一行的状态是可以由我们确定的。那是否可以把一行当作一个整体，然后像传统的动态规划那样进行处理呢？让我们试一下。

【例1】骑士(Sgu223) --2772

【问题分析】

每一行它对下一行的影响就体现在这一行的摆放方式以及之前总共放了多少个国王。所以我们可以把摆放方式作为状态，设f[i,j,s]表示第i行状态为a[j]且前i行已放s个国王的方案总数。这样很容易便得到了一个粗略的方程：

F[i,j,S]=∑F[i-1,k,T]

a[j],a[k]分别表示一种摆放方式，F[i,j]表示第i行用a[j]的摆放方式，且a[j]与a[k]相兼容。并且S等于T加上a[j]这种方式在这一行放置的国王数。

很明显这个方程是没有后效性的，可关键就在于j,k怎么在计算机上表示出来，这就需要我们的主题：**状态压缩**。

【例1】骑士(Sgu223) --2772

【问题分析】

看图便知，每一个格子只有两种状态，放和不放，并且注意到格子宽度最大为9。由这便想到了熟悉的二进制表示法。每一个格子对应一个二进制位。这样每一行便对应一个N位的二进制数，如下图所示：

1 0 0 0 1 0 0 0

即十进制的128+8=136

这样我们就可以把一种摆放方式转化为一个数，这样上面方程中的J,K就可以用数字来代替。我们就把一行的摆放方式作为**状态**，并把它**压缩**成了一个数！具体的算法流程如下：

①对于每一行，我们通过搜索得出一个合法状态。

②然后再枚举上一行与这一行相容的状态再累加即可，状态用N位的二进制数表示，最大仅为512，所以一个512*9*81的数组就可以了，还可以用**滚动数组**的技巧。空间是肯定可以承受的。

而一个粗略的时间复杂度：O(K*N*2^N*2^N)，似乎大了点。不过注意到由于国王是不能相邻放置的。所以我们可以用一个f(n)来表示当列数为n时每一行可能的放置总数。则f(n)=f(n-1)+f(n-2)

初始值：f(1)=2，f(2)=3。 则f(9)=89。因此最大也才是9*89*89*81≈6000000。是可以承受的。

【例1】骑士(Sgu223) --2772

【问题分析】

设f[i][j][mm]表示第i行状态为s[j]且前j行已放mm个国王的方案数

设c[i]表示方案i中国王（即1）的数量。

则状态转移方程为：

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

1<=i<=n,1<=j<=tot,1<=k<=tot,0<=mm<=kk

初始条件：f[0,1,0]=1

Answer= $\sum_{i=1}^{tot} f[n][i][kk]$

【例1】骑士(Sgu223) --2772

【参考代码】

```
#include<iostream>
using namespace std;
const int MaxS=155,MaxK=101;
long long f[11][MaxS][MaxK]={0},Ans;
int num[MaxS],S[MaxS],N,K,s0;
void Prepare()
{  int i,j,k;
   s0=0; Ans=0;
   memset(f,0,sizeof(f));
   for(i=0;i<(1<<N);i++)   //枚举每行的状态：如N=8时，从00000000~11111111共计2^8种
   {  if(i&(i<<1))continue; //检查当前状态冲突不
      k=0; //统计这种状态放置国王的个数
      for(j=0;j<N;j++)   if(i&(1<<j))k++;
      S[++s0]=i;  //可用状态
      num[s0]=k; //放置国王数
   }
}
```

【例1】骑士(Sgu223) --2772


```
void State_Compress_Dp()
{  int i,j,kk,t,s1,s2;
   f[0][1][0]=1;
   for(i=1;i<=N;i++)//阶段：行
   for(t=1;t<=s0;t++)//前一行的状态
   {  s1=S[t];
      for(kk=0;kk<=K;kk++)//前i-1行放置的国王总数
      {  if(!f[i-1][t][kk])continue;//没有这种状态
         for(j=1;j<=s0;j++)//枚举当前行的状态
         {  s2=S[j];
            if((s1&s2)||(s1&(s2<<1))||(s1&(s2>>1)))continue;//有冲突
            if(kk+num[j]>K)continue;
            f[i][j][kk+num[j]]+=f[i-1][t][kk];
         }
      }
    }
   for(i=1;i<=s0;i++)Ans+=f[N][i][K];
   printf("%lld\n",Ans);
}
```

【例1】骑士(Sgu223) --2772

```
int main()
{  while(scanf("%d%d",&N,&K)!=EOF)    // while(scanf("%d%d",&N,&K)==2)
   {   Prepare();
       State_Compress_Dp();
   }
   return 0;
}
```

【例2】牧场的安排 --2576 //luogu：1879

【问题描述】Farmer John新买了一块长方形的牧场，这块牧场被划分成M列N行(1<=M<=12; 1<=N<=12)，每一格都是一块正方形的土地。FJ打算在牧场上的某几格土地里种上美味的草，供他的奶牛们享用。遗憾的是，有些土地相当的贫瘠，不能用来放牧。并且，奶牛们喜欢独占一块草地的感觉，于是FJ不会选择两块相邻的土地，也就是说，没有哪两块草地有公共边。当然，FJ还没有决定在哪些土地上种草。

作为一个好奇的农场主，FJ想知道，如果不考虑草地的总块数，那么，一共有多少种种植方案可供他选择。当然，把新的牧场荒废，不在任何土地上种草，也算一种方案。请你帮FJ算一下这个总方案数。

【输入格式】

第1行: 两个正整数M和N，用空格隔开；

第2..M+1行: 每行包含N个用空格隔开的整数，描述了每块土地的状态。

输入的第i+1行描述了第i行的土地。所有整数均为0或1，是1的话，表示这块土地足够肥沃，0则表示这块地上不适合种草

【输出格式】第1行: 输出一个整数，即牧场分配总方案数除以100,000,000的余数

【例2】牧场的安排 --2576 //luogu：1879

【样例输入】

2 3

1 1 1

0 1 0

【样例输出】9

【输出说明】

按下图把各块土地编号：

1 2 3

4

开辟一块草地的话，有4种方案：选1、2、3、4中的任一块。

开辟两块草地的话，有3种方案：13、14以及34。

开辟三块草地只有一种方案：134。再加把牧场荒废的那一种，总方案数为4+3+1+1=9种。

【例2】牧场的安排 --2576 //luogu：1879

【题目大意】N*M的棋盘，每个格子不是0就是1，1代表可以种草，否则不能。相邻的两个格子不能同时种草，问总方案数。

【思路点拨】状态压缩类动态规划。将每一排的N个看成一个N位2进制数，先预处理出每一行可以运用的状态，这样可以去掉很多无效的状态（如110..），然后dp处理，枚举当前有效状态和上一行有效状态的关系。

设f[i][j]表示前i行中第i行用j这个状态的方案数。

$$f[i][j]=\sum_{k=1}^{a[i-1].num} f[i-1][k]$$ ，2<=i<=m,1<=j<=a[i].num,1<=k<=a[i-1].num

条件：a[i].st[j]与a[i-1].st[k]要相容；

初始条件：f[1][i]=1　(1<=i<=a[1].num)

Answer= $$\sum_{i=1}^{a[m].num} f[m][i]$$

【例2】牧场的安排 --2576 //luogu：1879

```
#include<iostream>
using namespace std;
const int MAX=99999999,maxn=15;
struct state{int st[4196],num;}a[maxn];
int m,n,f[maxn][1000]={0};
void getstate(int i,int t)
{  int num=0;
   for(int j=0;j<(1<<n);j++)//对于2^n个状态进行分别检查符合要求不
      if((j&(j<<1))||(j&(j>>1))||(j&t))continue;
      //去掉有2个及以上的1的情况，去掉和原图不相符的情况
        else a[i].st[++num]=j; //保存这种状态
     a[i].num=num; //保存这一行可能的状态个数
}
```

【例2】牧场的安排 --2576 //luogu：1879


```
void init()
{  int t,i,j,x;
   cin>>m>>n; //读入行和列
   for(i=1;i<=m;i++)//对于每一行处理
   {  t=0;
      for(j=1;j<=n;j++){cin>>x;t=(t<<1)+1-x;}
       //这列用十进制数t存储，小技巧t中，0代表可以种草，方便判断。
      getstate(i,t); //计算这一行可以运用的状态
   }
}
int main()
{  init();
    dp();
}
```

【例2】牧场的安排 --2576 //luogu：1879

```
void dp()
{  int i,j,k,ans=0;
   for(i=1;i<=a[1].num;i++)f[1][i]=1; //初始化第一行
   for(i=2;i<=m;i++)//以行为阶段处理
     for(j=1;j<=a[i].num;j++)//枚举这行的状态为状态
     {  f[i][j]=0;
        for(k=1;k<=a[i-1].num;k++)//上一行的状态为决策
        {  if(a[i].st[j]&a[i-1].st[k])continue; //当前行与上一行不相容
           f[i][j]+=f[i-1][k];
        }
     }
   for(i=1;i<=a[m].num;i++)ans=(ans+f[m][i])%100000000;
   cout<<ans;
}
```

【例3】炮兵阵地（NOI2001）--1479//luogu：2704

【题目描述】司令部的将军们打算在N*M的网格地图上部署他们的炮兵部队。一个N*M的地图由N行M列组成，地图的每一格可能是山地（用“H”表示），也可能是平原（用“P”表示），如下图。在每一格平原地形上最多可以布置一支炮兵部队（山地上不能够部署炮兵部队）；一支炮兵部队在地图上的攻击范围如图中黑色区域所示：

P	P	H	P	H	H	P	P
P	H	P	H	P	H	P	P
P	P	P	H	H	H	P	H
H	P	H	P	P	P	P	H
H	P	P	P	P	H	P	H
H	P	P	H	P	H	H	P
H	H	H	P	P	P	P	H

如果在地图中的灰色所标识的平原上部署一支炮兵部队，则图中的黑色的网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格。图上其它白色网格均攻击不到。从图上可见炮兵的攻击范围不受地形的影响。

现在，将军们规划如何部署炮兵部队，在防止误伤的前提下（保证任何两支炮兵部队之间不能互相攻击，即任何一支炮兵部队都不在其他支炮兵部队的攻击范围内），在整个地图区域内最多能够摆放多少我军的炮兵部队。

【例3】炮兵阵地（NOI2001）--1479//luogu：2704

【输入格式】文件的第一行包含两个由空格分割开的正整数，分别表示N和M；接下来的N行，每一行含有连续的M个字符（ ‘P’或者 ‘H’），中间没有空格。按顺序表示地图中每一行的数据。N≤100；M≤10。

【输出格式】文件仅在第一行包含一个整数K，表示最多能摆放的炮兵部队的数量。

【输入样例】

5 4

PHPP

PPHH

PPPP

PHPP

PHHP

【输出样例】

6

【例3】炮兵阵地（NOI2001）--1479//luogu：2704

【题目大意】给你个棋盘，有障碍，可以在非障碍点放置士兵，但是其上下左右4个方向的两个格子范围内不能再放士兵，求最多可以放多少士兵。

【题目分析】

由于M<=10，所以我们先枚举出2^M种可能不互相的冲突，存进b数组。然后以行为阶段进行DP。

设f[i][j][k]表示第i行用第j个放置方案，第i-1行用第k个放置方案，所得到的最大放置数。则：

状态转移方程为：$f[i][j][k]=\max\{f[i-1][k][y]+num[a[i][j]]\}$

其中a[i][j]表示第i行的第j种状态，num[a[i][j]]表示第i行第j个状态下的士兵数。

判断条件：

行数	状态
i-2	y
i-1	k
i	j

三者要兼容

初始条件：f[1][i][1]=num[a[1][i]],(1<=i<=p[1])

Answer=max{f[i][j][k]}

总复杂度为O（N*60^3）。

【例3】炮兵阵地（NOI2001）--1479//luogu：2704


```
#include<iostream>
using namespace std;
int n,m,a[110][1100]={0},p[110]={0},num[1100]={0};
long long f[110][500][500]={0};
bool ok[1100][1100]={0};
void do_state(int t,int tem) //计算每行的状态数目
{  int i;
   for(i=0;i<(1<<m);i++)//枚举状态，记下第t行在行上满足不冲突，
   //且与原图中可放置相符合的情况
   {  if(((i<<1)&i)||((i>>1)&i)||((i<<2)&i)||((i>>2)&i)) continue;
      //处理左右相邻的两格
      if(tem&i) continue; //这种状态不符合地形图
      a[t][++p[t]]=i;
      //记录状态,a[t][i]表示第t行的第i种情况,p[t]表示第t行的状态数
   }
}
```

【例3】炮兵阵地（NOI2001）--1479//luogu：2704


```
void DP()
{  int i,j,k,y;long long ans=0;
   p[0]=1;
   for(i=1;i<=p[1];i++)   {f[1][i][1]=num[a[1][i]];ans=max(ans,f[1][i][1]);} //初始化
   for(i=2;i<=n;i++)//枚举行为阶段
      for(j=1;j<=p[i];j++)//以这行的状态为状态
        for(k=1;k<=p[i-1];k++)//前一行的状态
          if(!(a[i][j]&a[i-1][k]))//兼容
             for(y=1;y<=p[i-2];y++)//前前一行的状态
               if(!(a[i][j]&a[i-2][y])&&!(a[i-1][k]&a[i-2][y]))//兼容
              {  f[i][j][k]=max(f[i][j][k],f[i-1][k][y]+num[a[i][j]]);
                 ans=max(ans,f[i][j][k]);
              }
   cout<<ans;
}
```

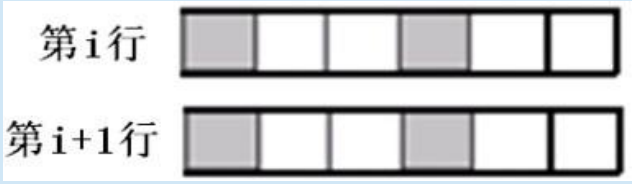
【例3】炮兵阵地（NOI2001）--1479//luogu：2704


```
void init() //读入数据
{  int i,j,k,tem; char ch;
   cin>>n>>m; //n行m列
   for(i=1;i<(1<<m);i++)
      for(j=0;j<m;j++)
   if((i>>j)&1) num[i]++; //统计每种情况中1的个数
   for(i=1;i<=n;i++)
   {  tem=0;
      for(j=1;j<=m;j++)
      {  cin>>ch;
         tem=(tem<<1)+1-(ch=='P');    //用tem记录每行的情况，1表示山地，0表示平原
      }
      do_state(i,tem); //计算每行的状态
    }
}
int main()  {  init();  DP();  }
```

【状态压缩DP】

【状压DP小结】

以每一行作为1种状态，研究上下两行之间的关系，利用上下两行之间的约束条件进行决策转移。如下图。一般采用一个二进制数表示状态，有时也用三进制或四进制数等。用二进制数表示状态最大的好处就是在决策转移时可以采用位运算，这样能极大提高算法效率。



状态压缩的动态规划实际上也是按常规处理的一种动态规划的做法，只不过由于每一个阶段状态数可能比较多，状态通常采用压缩存储方式，以利于运算和转移。试题的特点一般是数据整体规模较小，或者某一维规模较小。

【状态压缩DP】

可以去luogu搜“状态压缩”关键词，做带提高标签的。

【思考】骑士(kings) --2601

【问题描述】

用字符矩阵来表示一个8x8的棋盘，'.'表示是空格，'P'表示人质，'K'表示骑士。每一步，骑士可以移动到他周围的8个方格中的任意一格。如果你移动到的格子中有人质（即'P'），你将俘获他。但不能移到出棋盘或当前是'K'的格子中。

请问最少要移动多少步骑士才能俘获所有的人质。

【输入格式】

第一行一个整数N(<=5)，表示有多少个棋盘。即多组测试数据。

每一组有8行，每行8个字符。字符只有'.',大写'P',大写'K'三种字符。'P'和'K'的个数范围都在[1,10]。

【输出格式】

有N行，每行只一个整数，相应棋盘俘获全部人质所需要的最少步数。

【思考】骑士(kings) --2601

【输入样例1】

1

.PPPPKP.

.....

.....

.....

.....

.....

.....

.....

【输出样例1】

6

【思考】骑士(kings) --2601

【输入样例2】

```
2
P.....P
........
........
........
...KK...
........
........
P.....P
.....P.P
..K....P
....K...
..PP...P
...K..KK
........
K.......
KP.K....
```

【输出样例2】

```
20
9
```

【思考】骑士(kings) --2601

【题目分析】本题突破口在于假设只有一个骑士的情况进行处理：

（1）f1[v1][p]表示从人质v1出发，走遍集合p中所有人质的最小距离。

f1[v1][p]=min{dis[v1][v2]+f1[v2][p-(1<<v1)]}

（2）g[i][p]：表示从第i个骑士出发，抓获集合p中所有人质的最小距离。

g[i][p]=min{dis[i+num_p][k]+f1[k][p]}

（3）f[i][p]：表示前i个骑士抓获集合p中所有人质的最小距离和

f[i][p]=min{f[i-1][p-k]+g[i][k]}

ans=f[num_k][1<<num_p-1]

【计算步骤】

①求出每两点之间的距离

②求出从某个人质V1出发走遍集合P中人质的最小距离

③求出每个国王抓获集合P人质的最小距离

④求出前K个国王抓获集合P中人质的最小距离,即合并DP