exp 1

PB20111647 鲍润晖

exp1.1

启发式函数选择

不包含局部信息的启发式函数

- 一个显然的启发式函数是$h_1 = \lceil \frac{1的总数}{3} \rceil$
  它实际上把本题松弛到了 *每次可以翻转锁盘上的任意1、2或3个数* ，比原题的要求低了很多，所以是**可采纳的**
  下面考虑**证明其一致性**:
  设当前局面为n，代价 $h_1(n) = \lceil \frac{n中1的总数}{3} \rceil$，下一步实际能做的是翻转一个L形，到达局面$n'$。它可能把3个1翻转到0，因此 $h_1(n') = \lceil \frac{n中1的总数-3}{3} \rceil = \lceil \frac{n中1的总数}{3} \rceil - 1$； 也有可能把几个1翻转到0并把几个0翻转到1，那么此时 $h_1(n') \geq \lceil \frac{n中1的总数}{3} \rceil - 1$
  综上有 $h_1(n') + 1 \geq h_1(n)$，满足三角不等式，故有一致性
- 同理$h_2 = \frac{1的总数}{3}$也是一个启发式
  显然$h_2 \leq h_1$，h_1已然是可采纳的，那么h_2也是**可采纳的**。同理也可以证明h_2的**一致性**

比较h_1与h_2的优劣:
h_1更大，所以理论上会更占优势。但是h_1的取整导致它可能对细微的变化不够敏感：如果某一节点将要展开的子节点中1的总数为28、29、30，那么h_1将看不出它们之间有什么区别，从而随便选择一个子节点展开。而h_2可以区分它们的好坏。

包含局部信息的启发式函数

以上两个启发式都是翻转锁盘上的任意位置，这和原问题的L形约束相差较远

下面这种分散的情况期望用一次操作解决就太过乐观了。所以需要观察1的排布是否有局部性

```
1 0 0 0
0 0 0 0
0 0 0 1
1 0 0 0
```

下面这种虽然集中在一起，但是由于不是L形，也不能期望用一次操作解决。所以要合理定义局部，不能太大或太小

```
0 0 0 0
0 1 1 1
0 0 0 0
0 0 0 0
```

只有这种局部性才是能够一次操作解决的:

```
0 0 0 0
0 1 1 0
0 1 0 0
0 0 0 0
```

- 考虑用**尽可能少**的2×2的方块来覆盖锁盘上所有的1，方块之间可以重叠，此时方块的数量定义为h_3
  上面的例子中h_3的值分别为3、2、1
  而譬如

  ```
  0 1 0
  1 1 1
  1 1 0
  ```

  h_3值为2，两个方块重叠了一部分，并且这也是实际的值
  下面**证明它是可采纳的**:
  它把原问题松弛成了 *每一步可以翻转一个2×2方块中的1到4个数字*，那么到最优解的代价就是用尽可能少的方块数量。而之所以称其为松弛，是因为只有 *每一步翻转一个2×2方块中的3个数字* 和是原问题的约束一样，而 *每一步翻转一个2×2方块中的1、2、4个数字* 是在原问题下需要很多步才能办到的事情。所以是可采纳的。
  下面**证明它是一致的**:
  设当前局面为n，下一步实际能做的是翻转一个L形，到达局面n'

反证其不一致，就是要求 $h_3(n')+1<h_3(n)$，即 $h_3(n')\leq h_3(n)-2$，即翻转一个L形之后，n'比n少了2个覆盖方块（或者更多）
考虑这个L形在n中的位置：
  - 如果它刚好在某个方块上，那么翻转它只会影响该方块，不会少两个覆盖方块
  - 如果它和两或三个方块都有重叠，那么翻转它时会影响多个方块，但是通过枚举简单情况，可以发现这些被影响的方块不会被同时清零，也即n'比n最多就少了1个覆盖方块
    综上得证一致性
- 但是寻找这个最少方块的个数并不是容易的事。可以用简单的贪心算法得出一个较少的解，设该解为h_4

```
输入：初始锁盘M
输出：方块个数num

num = 0
repeat:
    尝试2*2方块在M中的所有摆放情况，得到覆盖最大1的个数max_n和位置loc
    if max_n == 0
        return num
    else
        num = num + 1
        将loc内所有元素变为0，更新M
    重置max_n和loc
```

这个算法在 尝试 步骤用时$O(n^2)$，总共循环次数$O(方块个数)$，总占用空间$O(n^2)$，可以说计算代价相当小了
可以找到例子说明h_4并**不可采纳**，比如下面的例子$h_4=3$，$h_3=2$，实际估计代价也是2

```
0 1 1 0
1 1 1 1
0 0 0 0
0 0 0 0
```

但是对于**大多数情况**来说，$h_4\geq h_3$并不意味着$h_4\geq$ 实际估计代价，因为h_3确实很小，足以满足实际估计代价 $\geq h_4\geq h_3$

比较这些启发式函数在5*5样例上的运行结果：

对input0（5*5）	h_1	h_2	h_4
总展开节点	789	2846	200
解的步数	5	5	5

对input3（5*5）	h_1	h_2	h_4
总展开节点	3791	14922	342
解的步数	7	7	7

对input4（5*5）	h_1	h_2	h_4
总展开节点	129612	--	13502
解的步数	9	--	9

对input5（5*5）	h_1	h_2	h_4
总展开节点	26751	27254	394
解的步数	7	7	7

可以发现h_4大约优于h_1 10倍，而h_2比h_1更差
这说明了对于可采纳且一致的启发式，越大的启发式意味着越占优势

但是这三个启发式在面对更大的图的时候并不管用，需要寻找新的启发式

不可采纳的启发式

如果对于比较大的图，追求很快求出一个比较好的解，那么调大启发式（以至于其不可采纳）是一个不错的选择

如果采用不可采纳的启发式 $2*h_4$ 那么有

$2*h_4$	**input6 (9*9)**	**input7 (9*9)**	**input8 (10*10)**	**input9 (12*12)**
总展开节点数	27	117	24	190

$2 * h_4$	input6 (9*9)	input7 (9*9)	input8 (10*10)	input9 (12*12)
解的步数	11	16	16	23
最优解步数	11	14	16	23

可以发现其中有的解的步数和总展开节点数都相差不大的，这说明此时搜索算法向深拓展了很多，而向宽没怎么拓展。这是因为 g+h 中 h 已经大于 g 不少，导致解的深度 g 的惩罚显得不是很厉害。而考虑到这题的性质：无论什么图都是有解的，所以算法可以*一条路走到黑*，而不用担心进入死胡同

显然 $2 * h_4$ 的2是随便取的，那么可以尝试调参 $\alpha * h_4$，$\alpha \geq 1$，以尝试每种input的解的最小步数

$1.5 * h_4$	input6 (9*9)	input7 (9*9)	input8 (10*10)	input9 (12*12)
总展开节点数	45	2105	40	1759
解的步数	11	16	16	23
最优解步数	11	14	16	23

可以发现input7和input9的展开节点数显著上升，这暗示了这个启发式的搜索空间明显更大了，也就是前一个启发式漏掉了更多的应该被检查的节点

但是这件事可以做得更细腻一点：**我们可以尝试出一个 $\alpha(M) * h_4$，使得它恰好是当前锁盘M的预期代价的一个下界**，其中$\alpha(M)$是锁盘M的一个函数，这样A*算法的启发式函数就可以**自始至终都满足可采纳性**了
关于$\alpha(M)$的选择，有几个观察：

- 对$\forall M$，$\alpha(M) = 1$时，$\alpha(M) * h_4$是M预期代价的下界，只不过太小了，展开节点数很多
- $\alpha(M) * h_4$越接近M的预期代价，则展开节点数越少，所以$\alpha(M)$要尽可能大，但是不能高估预期代价
- M越复杂（可以用1的数目做度量），h_4与预期代价之间的差额就越大，就可以尝试更大的$\alpha(M)$
- 上面这个观察是对平均而言，而对于具体的M和$\alpha(M)$，有可能会找到违背可采纳性的反例，所以这个可采纳性是对平均而言的，可以称之为**平均可采纳性**

比如采用$\frac{M\text{中1的总数}}{\text{锁盘格子总数}} + 1$作为$\alpha(M)$，它的值域在$[2, 1]$并且满足上述条件

A*算法

解的上界

下面证明任意一个2*2以上大小的锁盘都是有解的，先从一个点的情况考虑：

```
0 0     1 1     1 0     0 0
1 0 -> 0 0 -> 1 1 -> 0 0
```

可见在这三次操作里面，左下角的点翻转了3次，其它点翻转了2次。所以只有左下角的点被改变了。
那么对于一个任意的棋盘，我们对每个点都如此操作即可

所以可以得出$\text{解的上界} = 1\text{的总数} * 3$

剪枝

对于某个锁盘的局面进行某个翻转时，有可能是将三个0翻转成三个1，这种展开显然是多余的。启发式函数可能对这种情况给与比其它情况更大的惩罚，但是也有可能不会这样。所以需要直接限定不能将三个0翻转成三个1。

这种带来的剪枝是很可观的：

对input0（5*5） 启发式h_1	剪枝	不剪枝
总展开节点	789	4330
解的步数	5	5

算法结构

首先介绍算法中定义的结构体：
矩阵结构体：

```
struct Matrix{
      vector< vector<bool> > a;
      Matrix(){}
      //在p的基础上通过i,j,s变化生成新矩阵，此处i,j,s需要合法，如果此时将三个0翻成三个1，则设useless为true
      Matrix(Matrix p, int i, int j, int s, bool &useless)
      void print()
      bool operator<(Matrix x)
};
```

它包含成员变量 a 表示锁盘内容
定义了在p的基础上通过i,j,s变化生成新矩阵的构造函数，并且可以判断该变化需不需要剪枝
重载了 < 以使用c++的数据结构

矩阵状态结构体：

```
struct MatrixState{
        Matrix m;
        int pos; //记录状态编号
        int g; //到达此状态移动的步数
        float h; //预期代价

        MatrixState(){}
        //创建新矩阵状态，并按照h_func计算h
        MatrixState(Matrix x, int p, int t, float (*h_func)(Matrix M))
        bool operator<(MatrixState x)
}
```

它是A*算法中的每个节点的类型
包含矩阵结构体 m ，状态编号（回溯解时需要），以及该节点的g和h
定义了创建新矩阵状态的构造函数，并且可以指定新矩阵状态计算时的启发式函数

下面介绍算法流程：

```
//A*搜索，输入初始矩阵状态M，返回总步数step和最终结果在path中的下标cnt（后续以此回溯解路径）
void AstarSearch(MatrixState MS,int &step,int &cnt)
{
    set<Matrix> seen; //用于判断状态矩阵是否重复的集合
    priority_queue<MatrixState> OPEN; //待展开的队列，按g+h从小到大排序
    //设置初始
    OPEN.push(MS);
    seen.insert(MS.m);
    path[1]=MS;

    int count = 0;

    while(!OPEN.empty()){
        //弹出第一个矩阵状态F
                MatrixState F=OPEN.top();
                OPEN.pop();
                //判断终止状态
                if(!F.h){
                        step=F.g;
                        while(path[cnt].h) cnt--;
                        cout << "总遍历节点：" << count << endl;
                        return ;
                }

                //展开所有节点
        //要按s分类，因为每种s对应一些i,j非法
        for(int s=1; s<=4; s++)
        {
            int i_start, i_end, j_start, j_end;
            switch (s)
            {
            case 1:
                i_start = 1;
                i_end = Msize;
                j_start = 0;
                j_end = Msize-1;
                break;
            case 2:
                i_start = 1;
                i_end = Msize;
                j_start = 1;
                j_end = Msize;
                break;
            case 3:
                i_start = 0;
                i_end = Msize-1;
                j_start = 1;
                j_end = Msize;
                break;
            case 4:
                i_start = 0;
                i_end = Msize-1;
                j_start = 0;
                j_end = Msize-1;
                break;

            default:
                break;
            }
            for(int i = i_start; i < i_end; i++)
                for(int j = j_start; j < j_end; j++)
                {
                    //一种剪枝：如果当前把三个0翻成三个1，则useless为true
                    bool useless = false;
                    //展开当前节点
                    Matrix new_m(F.m, i, j, s, useless);
                    //如果seen集合中没有此矩阵，则更新seen，并插入OPEN队列
                    if(!seen.count(new_m))
                    {
                        seen.insert(new_m);
                        //此状态的g是父节点F.g+1
                        MatrixState new_MS(new_m, cnt+1, F.g+1, h1);
                        OPEN.push(new_MS);
                        //设置向前查找
                        path[++cnt]=new_MS;
                        pre[cnt]=F.pos;
                    }
                }
        }
        }
}
```

算法采用c++的set容器 `seen` 维护已经看过的矩阵，采用priority_queue实现待展开矩阵状态 `OPEN` 的优先队列
在用第一个矩阵状态初始设置后，只要 `OPEN` 队列没有清空，就弹出其第一个矩阵状态（也即g+h最小的）作为 `F` ，如果 `F` 是终点则结束，否则展开 `F` 的所有子节点
在展开时需要对边界处理，因为并非所有ijs都是合法的。如果展开的节点是见过的则不展开，反之将它插入 `seen` 和 `OPEN` 中

为了最后能输出解，还需要在此时维护移动过程中产生的状态结构体数组 `path` 和每个状态的前一个状态的结构体编号 `pre`

别人的技巧和经验

在尝试解决大于5*5的锁盘的时候，一开始碍于时间限制只能采用不可采纳的启发式。在这里总结了我身边同学的不可采纳的启发式：

- $\alpha * \lceil \frac{1的总数}{3} \rceil$，$\alpha > 1$
  - 据说调整α的大小可以得到比较少的展开节点数
- $\sum_{i,j} \frac{dis(a_i,a_j)}{2*(1的总数)}$ 即所有1的点之间的距离平均
  - 其中a_i, a_j是那些1的点的坐标，$dis(,)$取哈夫曼距离
  - 用这个作为启发式，它的值是相当大了。但是它会给1的点分散的图很大的惩罚，以此驱使A*算法先考察1的点集中一些的图。这个想法是好的，因为两个点的距离更近的话，更有可能通过少数几次变化将它们同时解决
  - 但是这个启发式的解释性很差：
    - 两个点之间的距离应该与*将它们同时解决的变化数*挂钩，而不应该与哈夫曼距离挂钩。当两个点的哈夫曼距离大到一定程度的时候，将它们解决的变化数达到6并且不会再变了
    - 多个点中的两两距离比较复杂，真正要考虑的距离可能只是它的一个部分
  - 所以这个启发式对输入很敏感，有的样例可以秒出答案，而有的却要展开很多很多节点
- $\sum_i w(a_i)$ 即对所有是1的点赋权重并相加
  - 其中a_i是那些1的点，$w()$是一个考虑该点附近情况的一个权重函数
  - 理论上所有启发式都可以等价于这个形式，但是这个权重函数很难考虑出一个简单并且合适的

后来又得知了一个剪枝的方法，对于每个状态，只需要考虑**包含锁盘上第一个1在内的12种翻转**，这瞬间将每个状态的展开节点数降到很少，对于很大的锁盘也可以用可采纳的启发式解决了
而这个剪枝的想法也来源于对题目的观察：**一个解的每个步骤都是可以任意调换顺序的**，所以总能将它调换成翻转包括当前第一个1的顺序。由此可见对题目的分析还是很重要的

以下运行结果全部采用这种剪枝为基础

运行结果总结

最优解步数：

input0(5*5)	input1(3*3)	input2(4*4)	input3(5*5)	input4(5*5)	input5(5*5)	input6(9*9)	input7(9*9)	input8(10*10)	input9(
5	4	5	7	7	7	11	14	16	23

总展开节点数对比：

- h_1 （1的总数/3取上整）

input0(5*5)	input1(3*3)	input2(4*4)	input3(5*5)	input4(5*5)	input5(5*5)	input6(9*9)	input7(9*9)	input8(10*10)	inp
292	38	13	84	348	59	4398	3123	太大了	太大

- h_4 （1的2*2覆盖总数）

input0(5*5)	input1(3*3)	input2(4*4)	input3(5*5)	input4(5*5)	input5(5*5)	input6(9*9)	input7(9*9)	input8(10*10)	inp
73	19	34	32	16	11	909	243	959	271

- dijkstra

input0(5*5)	input1(3*3)	input2(4*4)	input3(5*5)	input4(5*5)	input5(5*5)	input6(9*9)	input7(9*9)	input8(10*10)	inp
2419	37	236	1318	1828	1548	太大了	太大了	太大了	太大

显然就算最差的A*算法的性能也比dijkstra好，因为它毕竟是带有启发信息的

exp1.2

转化为约束满足问题

本题变量集合为每天的每个班次，值域集合为$\{0,1,...,N\}$

将每一天的每一班次视作一个可取$\{0,1,...,N\}$的变量，其中i表示它被指派了阿姨i，0表示它未被指派

定义几个约束：

- 每天分为轮班次数个值班班次
- 每个班次都分给一个宿管阿姨
- 同一个宿管阿姨不能工作连续两个班次
- 每个宿管阿姨在整个排班周期中，应至少被分配到$\lfloor \frac{DS}{N} \rfloor$次

那么第一个约束说明了有D*S个变量，第二个约束说明该约束满足问题的结束标志是每个变量都被指派了非0取值，这两者可以不纳入约束满足问题中约束的考虑
剩余两个约束定义为g_1和g_2，其中g_1是**二元约束**，g_2是**全局约束**

下面考虑怎么处理值班请求：
考虑到每一天的每一班次只能有一个赋值，所以**可以满足的请求数上界**是$D*S$

将值班请求作为约束优化

值班请求也可以视为约束：
对每一天的每一个班次，每个阿姨都有一个对应的请求，对应一个约束

- 如果阿姨的请求为1，而该班次的指派是她时，此约束不违背，此请求满足
- 如果阿姨的请求为1，而该班次的指派不是她时，此约束违背，此请求不满足
- 如果阿姨的请求为0，那么该班次的指派无论是不是她，都不视为违背，此请求不满足

（考虑到请求为0的约束无论班次指派，所以可以不考虑它们）

而总共$N*D*S$条值班请求中是1的部分，它们构成总请求约束集合$Request$，每个约束都是**一元约束**
我们要最大化满足的请求，就是找到$Request$集合的一个**最大的子集**R_1，使得存在一个排班表，它满足R_1中的所有约束以及上文的g_1和g_2约束，此时R_1的大小即满足的请求数

将值班请求作为值域优化

考虑对某一天的某一班次，有阿姨$\{n_{i_1},...n_{i_l}\}$都在请求，那么将它分配给$N-\{n_{i_1},...n_{i_l}\}$中的阿姨显然是浪费；将它分配给$\{n_{i_1},...n_{i_l}\}$中的某一个阿姨n_k，则$\{n_{i_1},...n_{i_l}\}-n_k$的请求不会满足，$n_k$的请求得到满足，这对于该班次来说，已经是最大的满足请求数了

所以可以用值班请求作为该班次的值域：
对于第d天的第s班次，有阿姨$\{n_{i_1}^{d,s},n_{i_2}^{d,s},...\}$都在请求，则设置该班次可被指派的值域为$\{0,n_{i_1}^{d,s},n_{i_2}^{d,s},...\}$（0表示尚未指派阿姨），在上面运行约束是$g_1$和$g_2$的CSP算法即可：

- 如果有解，那么此时可以满足的请求数达到了上界$D*S$
- 如果没解，那么为了安排值班表，得扩充某些班次的值域，（由于我们关注的是优化值班请求，而值班请求依赖于班次，所以扩充时直接扩充到N即可）
  - 假设扩充了k个班次的所有情形CSP都无解，而扩充了$k+1$个班次的某个情形CSP有解，那么可满足的最大申请数$D*S-k-1$，该解就是最终的排班表

所以可以按照扩充的班次数运行**广度优先搜索**

比较两种方法：

- 它们都是将CSP算法作为一个验证手段，通过尝试不同的约束或者值域的CSP，找到一个使得可满足的申请数最大的CSP
- 将值班请求作为约束优化
  - 改变约束实现起来比较麻烦
  - 对可满足的申请数*从小到大*搜索，不能保证搜到的第一个解是最优解
- 将值班请求作为值域优化
  - 改变值域实现起来比较简单，可以使用同一套CSP算法模板
  - 对可满足的申请数*从大到小*搜索，可以保证搜到的第一个解就是最优解

而且对于本次实验样例，因为1的分布比较密集，所有可以满足的请求数都接近上界$D*S$，综上选择**将值班请求作为值域优化**

CSP算法流程和实现

维护一个状态结构体，里面包含在回溯搜索算法中当前状态每个班次的可用赋值和指派情况

```
struct Status
{
    //可用赋值表：共D*S*(N+1)，available[d][s][n]为真表示d,s的n赋值可用，n=0不表示东西
    vector< vector< vector<bool> > > available;
    //指派表：共D行S列取值0~N，d行s列的值表示第d天第s班次的指派，0表示未指派
    vector< vector<short> > assign;
}


回溯搜索 back_track(state)
输入：约束条件、初始状态
输出：完全有效赋值 assign 或失败

if state.assign 是完全有效赋值 then
    return state.assign
通过MRV选择一个未赋值变量 X
选择一个 X 的所有可能变量的随机排序 D
foreach var in D do
    将指派 X 取 var 加入 assign，构造新状态 new_state
    在 new_state 上运行前向检查，更新 new_state
    if new_state.assign 是有效的 then
        result = back_track(new_state)
        if result != failure then
            return result
return failure
```

何时使用全局约束

回顾全局约束g_2：每个宿管阿姨在整个排班周期中，应至少被分配到$\lfloor \frac{DS}{N} \rfloor$次

但是在求解CSP问题时，对每个阿姨，她的排班是逐渐被确定的，即排班数是*由少到多增长*，如果所有阿姨都还没有分配很多排班，那么该怎么确定这个排班达到了$\lfloor \frac{DS}{N} \rfloor$次的下界呢?

一个简单的办法是把所有赋值都确定下来之后，再检查是否满足全局约束
这个函数被实现为 `bool check_cons2(Status state)`

```
//递归终点
if(check_finish(state) )
{
    if(check_cons2(state))
    {
        cout << "finish!"<<endl;
        state.print();
        return;
    }
    else
    {
        cout << "bad end!"<<endl;
        return;
    }
}
```

但是这个的性能非常差。我们放弃了**早死早超生原则**，而允许一些在赋值中途就能发现问题的赋值表走到最后

可以选择一个能够在赋值中途发现问题的方法：考察工作最多的几个阿姨是否让其它阿姨不能达到最低工作标准
这个函数被实现为 `bool check_cons2_mid(Status state)`，实现起来更加复杂了，但是效率提升显著

还有一点：**节点的赋值的展开顺序**对性能也有很大影响：
譬如每个节点的可用赋值都是 1~N ，并且每个节点的赋值展开都是按照 1~N 的顺序尝试，那么递归运行 `back_track` 时，前几个节点会被依次赋值为 `1，2，1，2，1，...`（因为它不违背约束g_1并且目前还没有违背g_2），但是这个赋值再往后有很大可能违背g_2，因为它用1，2太多了
那么有什么办法避免这种情况吗？答案是**随机选择一个节点的赋值的展开顺序**，即上面伪代码的 `选择一个 X 的所有可能变量的随机排序 D`，这也是一种提前避免无效探索的剪枝

前向检查

约束g_1是二元约束，非常适合约束传播，而g_2则不适合，故不约束传播

```
//将新状态的d，s，n赋值进行弧相容检查
//只传播cons1
//递归进行，直到不能约束传播为止
void forward_check(short d, short s, short n)
{
    //cons1中弧相容只涉及前后两个节点，注意跨天情况和边界情况（这里考虑s>=3）
    short d1, s1, d2, s2;
    if(d == 0 && s == 0)
    {
        available[0][1][n] = false;
        short only_n1 = -1;
        if(count_available(0, 1, only_n1) == 1 && assign[0][1]==0)
        {
            assign[0][1] = only_n1;
            forward_check(0, 1, only_n1);
        }
        return;
    }
    else if(d == D-1 && s == S-1)
    {
        available[d][s-1][n] = false;
        short only_n1 = -1;
        if(count_available(d, s-1, only_n1) == 1 && assign[d][s-1]==0)
        {
            assign[d][s-1] = only_n1;
            forward_check(d, s-1, only_n1);
        }
        return;
    }
    else if(s == 0)
    {
        d1 = d-1;
        s1 = S-1;
        d2 = d;
        s2 = 1;
    }
    else if(s == S-1)
    {
        d1 = d;
        s1 = s-1;
        d2 = d+1;
        s2 = 0;
    }
    else
    {
        d1 = d;
        s1 = s-1;
        d2 = d;
        s2 = s+1;
    }
    //注销两个节点中n的可用性
    available[d1][s1][n] = false;
    available[d2][s2][n] = false;
    //如果某个节点只有一个可用且未赋值，那么更新赋值
    short only_n1 = -1;
    short only_n2 = -1;
    if(count_available(d1, s1, only_n1) == 1 && assign[d1][s1]==0)
    {
        assign[d1][s1] = only_n1;
        forward_check(d1, s1, only_n1);
    }
    if(count_available(d2, s2, only_n2) == 1 && assign[d2][s2]==0)
    {
        assign[d2][s2] = only_n2;
        forward_check(d2, s2, only_n2);
    }
    return;
}
```

在不考虑边界条件和跨天情况时，要将d天s班次的n赋值约束传播到它前后两个班次，即注销掉 `available` 中它们的可用性
如果注销掉可用性时它刚好只有一个赋值可用，那么就可以用这个来赋值，并递归地触发下一个约束传播

最小剩余值启发式

```
//MRV启发式，返回目前的可用赋值最少的未赋值节点d，s
void MRV_find(short &d_find, short &s_find)
{
    int count_min = N+1;
    short i_find, j_find;
    short nouse;
    for(short i=0; i<D; i++)
    {
        for(short j=0; j<S; j++)
        {
            //遍历assign表找未指派的节点
            if(assign[i][j]==0)
            {
                int count = count_available(i, j, nouse);
                if(count < count_min)
                {
                    count_min = count;
                    i_find = i;
                    j_find = j;
                }
            }
        }
    }
    d_find = i_find;
    s_find = j_find;
}
```

依据当前 `assign` 表寻找未赋值的节点，并统计它们的可用赋值数，最终返回可用赋值最少的节点

将值班请求作为值域优化

这里涉及到两个部分：

- 用值班请求初始化CSP中每个节点的值域
- 在当前CSP无解时拓展哪个节点的值域

实现了类 `Request` ，其中 `r` 记录了各个值班请求

```
//请求表 N*D行，每行S个元素表示请求
struct Request
{
    vector< vector<bool> > r;
    Request(){}
    //设置状态S的可用赋值
    void set_available(Status &State)
    //拓展状态S的可用赋值
};
Request Req_in;
```

在读入文件时初始化 `Req_in` ，然后调用 `set_available` 初始化CSP中每个节点的值域

拓展节点值域时，因为我们希望拓展后尽快找到有解的CSP，所以优先拓展值域最小的节点，这部分也可以复用MRV启发式的代码

运行结果（以input0为例）

有意思的是，input0是唯一一个可满足的请求数达不到上限$D*S$的样例，所以按照*用请求初始化值域*的方法来做，会发现第4天的第0个班次没有值域
此时需要拓展该班次的值域为1~N并重新运行即可
此时有解：

```
D0: 1 2 1
D1: 2 3 1
D2: 3 2 1
D3: 3 1 2
D4: 3 2 1
D5: 3 2 3
D6: 1 2 3
```

按照前面论证的，在每个节点的值域都不拓展的时候无解，而拓展了一个节点的值域就有解的，说明其最大的可满足请求数$D*S-1$