A*

JL22110003 邱文韬

启发式函数

三种启发式函数：

1. DisjointL：不相交的"L"个数
2. NumOf_1: 1的个数
3. Manhattan：所有1之间的曼哈顿距离两两求和总和

admissible

1. 其中DisjointL启发式是可采纳的，因为在一个矩阵中，将变换一个“L”代价视作1，而统计不相交的L个数，相当于原问题的松弛化，即将锁盘内所有"L"都视作不相交，即最简单的情况逐个把“L”由全1变成全0，但实际上可能还存在分散的没有构成“L”的1，所以实际代价一定比这种方式要更多，所以DisjointL是可采纳的。
2. 如果将变换一个"L"的代价视作3，则NumOf_1启发式也是可采纳的，与DijointL类似，考虑问题的松弛化，即锁盘内如果恰所有“L”都不相交，那么逐个转换1的代价最多和实际代价相同，如果1是分散的，不存在任何一个“L”，那么统计的1个数会远小于实际代价，而不会超过实际代价。
3. 第三个启发式函数Manhattan不满足可采纳性，采用曼哈顿距离，很容易举出一个例子，比如如果只存在一个“L”，那么无论实际代价视作1或者3，得到的曼哈顿距离都会超过这个实际代价，所以估价可能会远大于实际代价，但此时运行速度很快，但一般不是最优解。

consistent

1. 假设存在一个状态n和它的后继状态n'，使得DisjointL(n) > c(n,n') + DisjointL(n')， 其中c(n, n')是从状态n到状态n'的实际代价。状态n到n'的实际代价为1，即转动一个“L”，并且转动后锁盘n'满足：DisjointL(n) -1≤DisjointL(n')≤DisjointL(n) +1，即DisjointL(n')最小为DisjointL(n) -1，带入上式，DisjointL(n) > 1 + DisjointL(n)-1 = DisjointL(n) > DisjointL(n)，DisjointL(n) > DisjointL(n)不成立，与原假设矛盾，所以不存在一种状态使得假设DisjointL(n) > c(n,n') + DisjointL(n')成立，即该启发式函数满足一致性。
2. 第二个启发式函数与第一个类似，假设存在一个状态n和它的后继状态n'，使得NumOf_1(n) > c(n,n') + NumOf_1(n')， ， 即 NumOf_1(n) - 3≤NumOf_1(n')≤NumOf_1(n) + 3， 带入假设中， NumOf_1(n) > 1 + NumOf_1(n)-1 = NumOf_1(n) > NumOf_1(n)，NumOf_1(n) > NumOf_1(n)不成立，与原假设矛盾，所以满足一致性。
3. 第三个启发式不满足可采纳性，所以也不满足一致性。

算法的主要思路

- **数据结构**：

1. **锁盘数据结构体**，N：锁盘大小、g：从父结点到当前节点的代价、h：从当前结点到目标状态结点的估计代价即启发函数值、parent：父结点、pre：从父结点到当前结点采取的动作，一个三元组表示选择的坐标和对应的变换类型。存在有参构造函数和无参构造函数，用于初始化结点或者在后续拓展结点时选择继承属性的拓展方式。

```
struct LockMatrix {
    int N;
    TwoDMatrix matrix;
    int g;
    int h;
    LockMatrix* parent;
    pair<pair<int, int>, int> pre;

    LockMatrix() {
        N = 0;
        matrix =     TwoDMatrix();
        g = 0;
        h = 0;
        pre = make_pair(make_pair(-1, -1), -1);
        parent = NULL;
    }

    LockMatrix(const int& g, const int& h, const  TwoDMatrix& matrix) {
        this->N = matrix.size();
        this->matrix = matrix;
        this->g = g;
        this->h = h;
        this->pre = make_pair(make_pair(-1, -1), -1);
    }
};
```

2. **封闭列表**：VisitedList，存储每个已拓展过的结点对应的锁盘状态，避免重复计算
3. **优先队列**：priority_queue<LockMatrix,*vector<LockMatrix>*, cmp>，边缘队列用一个优先队列实现，存放待拓展的结点，自定义比较方法，根据f = g+h值从小到大入队，每次拓展结点都是选择估计代价最小的结点，优先队列排序时间复杂度为$O(\lg n)$，如果通过自己手动时间排序算法，可能取得冒泡排序$O(n^2)$或者快速排序$O(n \lg n)$，但即使是线性时间排序也比优先队列更慢，所以选择优先队列加快这个过程，也是我在A*算法实现中采取的一个优化手段之一。

- **算法**：

1. **Astar算法**：朴素的Astar算法，根据设计的启发式函数选择结点拓展，但性能受启发式函数影响很大，并且很难得到一个接近实际代价的估价函数。
    - 算法流程：
        1. 传入起始结点对应的锁盘状态
        2. 将锁盘状态加入边缘队列，再加入封闭列表
        3. 判断边缘队列是否为空，不为空则队头结点出队
        4. 判断状态是否是目标状态，是则根据父结点沿着链表记录解路径，算法结束，否则继续下一步
        5. 通过`ExtendFrontier`函数将当前状态的子结点加入边缘队列 (每个结点有4种“L”的变换方式，也就是每个状态有四个子结点)
        6. 当前状态加入封闭列表
        7. 回到步骤3

2. **IDAstar算法**：迭代Astar算法：与A*算法类似，但加入一个limit限制，每次选取一个limit值，对于超过这个阈值的结点，不加入边缘队列，这样队列中的结点数大大减少，如果在一个较小的深度就找到了一个解则直接返回，如果没找到，将这个阈值设置为上一次f = g + h中超过阈值的最小值，再重新从初始状态开始向下拓展结点，每一轮迭代的深度都会加深。
   - 算法流程：
     1. 传入起始结点对应的锁盘状态
     2. 判断当前的limit值是否小于`INT_MAX`（即不溢出int），并且`flag!=true`(即未到达目标状态)，满足则继续步骤3，否则算法结束
     3. 初始化边缘队列，初始化封闭列表，将初始状态入队，再将该状态加入封闭列表
     4. 判断边缘队列是否为空，不为空则队头结点出队
     5. 判断状态是否是目标状态，是则flag = true，否则回到步骤2
     6. 剩余步骤与A*类似，拓展边缘队列的函数使用`ExtendFrontierWithLimit`，在这个函数中修改limit的值，因为没有找到目标结点，所以需要更新limit

启发式函数为 0，此时 A* 退化为 Dijkstra 算法

当启发式函数为0时，每个状态转移的估价f(n) = g(n)，而每一步操作的代价都是相同的，即转变一个“L”，可以设置为1或者3，此时相当于在一个权值均相同的起始状态和目标状态之间找一个最短路径，此时运行的非常非常慢，基本只能够运行出input1的结果。

情况	函数	结果
$\hat{h}(n)=0$，即 $\hat{f}(n)=\hat{g}(n)$	A*算法退化为Dijkstra算法	保证能找到最短路径
$\hat{h}(n) \leq$ 实际代价	$\hat{h}(n)$越小，A*扩展的节点越多，运行的越慢	保证能找到一条最短路径，但运算更快了
$\hat{h}(n)=$ 实际代价	仅寻找最佳路径，而不扩展任何别的节点	保证能找到一条最短路径，并且运算非常快
$\hat{h}(n)>$ 实际代价	寻找最佳路径且扩展别的任何节点	不能保证找到一条最短路径，但运算更快了
$\hat{h}(n)>>\hat{g}(n)$	A*算法退化为BFS算法	不能保证找到一条最短路径，但运算非常快

比较并分析使用 A* 方法带来的优化效果

Dijkstra

输入	运行时间	解步骤数
input0	超时	无
input1	7.414s	4
input2	超时	无
input3	超时	无
input4	超时	无
input5	超时	无
input6	超时	无
input7	超时	无
input8	超时	无
input9	超时	无

三种启发式函数分别命名为：DisjointL、NumOf_1、Manhattan，其中DisjointL最慢，但能保证找到最短路径，而NumOf_1在某些输入下可以找到最多路径，并且运算相对较快，而Manhattan不能保证找到最短路径，但运行非常快，其中IDA*的效率和A *对比有一定提升，但因为输入数据的原因，在后续输入规模较大时仍然运行时间很长也无法给出结果。

A*+DisjointL

输入	运行时间	解步骤数
input0	超时	无
input1	0.573s	4
input2	832.273s	5
input3	超时	无
input4	超时	无
input5	超时	无
input6	超时	无

输入	运行时间	解步骤数
input7	超时	无
input8	超时	无
input9	超时	无

IDA*+DisjointL

输入	运行时间	解步骤数
input0	超时	无
input1	0.424s	4
input2	32.273s	5
input3	超时	无
input4	超时	无
input5	超时	无
input6	超时	无
input7	超时	无
input8	超时	无
input9	超时	无

A* +NumOf_1

输入	运行时间	解步骤数
input0	0.03s	5
input1	0.007s	4
input2	0.279s	7
input3	0.192s	7
input4	16.428s	9
input5	1.018s	9
input6	超过100s！	无

输入	运行时间	解步骤数
input7	超过100s！	无
input8	超过100s！	无
input9	超过100s！	无

IDA* + NumOf_1

输入	运行时间	解步骤数
input0	0.033s	5
input1	0.01s	4
input2	0.347s	7
input3	0.062s	7
input4	0.458s	9
input5	42.648s	9
input6	超时	无
input7	超时	无
input8	超时	无
input9	超时	无

A*+Manhattan

输入	运行时间	解步骤数
input0	0.107s	5
input1	0.009s	4
input2	0.001s	5
input3	0.138s	7
input4	0.182s	9
input5	0.086s	11
input6	9.733s	11
input7	1.878s	26

输入	运行时间	解步骤数
input8	2.545s	32
input9	7.054s	39

IDA*+Manhattan

输入	运行时间	解步骤数
input0	0.201s	5
input1	0.015s	4
input2	0.002s	5
input3	0.183s	7
input4	0.179s	9
input5	0.087s	11
input6	8.029s	11
input7	1.222s	34
input8	2.45s	32
input9	8.96s	39

结果:

以input0为例

解步骤数:5
坐标：(1,0) 方式：1
坐标：(2,2) 方式：2
坐标：(0,3) 方式：4
坐标：(1,3) 方式：4
坐标：(1,4) 方式：2

Debug

在测试A*算法实现时，遇到报错：`Process finished with exit code -1073741819 (0xC0000005)`

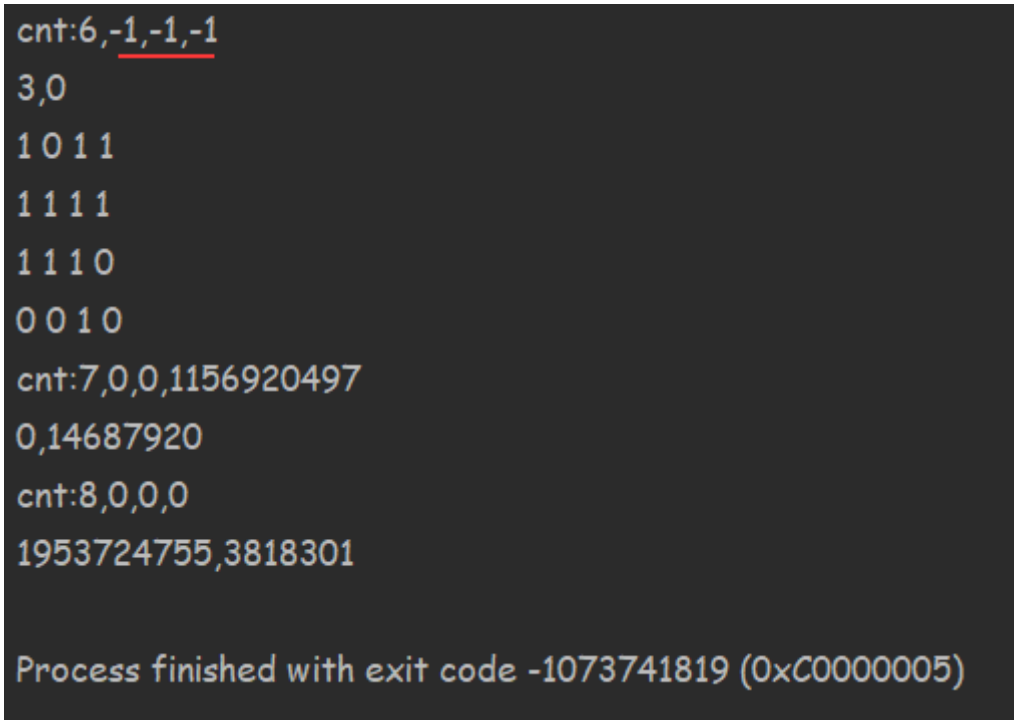
Process finished with exit code -1073741819 (0xC0000005) 是一种常见的错误代码，表示程序遇到了访问违规的情况，通常是由于以下几种原因之一导致的：

1. 空指针访问：在代码中使用了空指针进行访问，即尝试读取或写入空指针指向的内存地址。
2. 内存越界：访问了超出分配内存范围的数组元素或对象。

3. 栈溢出：递归或大量函数调用导致函数调用栈超出了系统限制。
4. 未初始化的变量：使用了未初始化的变量或对象。
5. 释放已释放的内存：重复释放已经被释放的内存块。

解决方法：

1. 仔细检查代码中可能存在的空指针访问或内存越界的情况。确保在访问指针之前进行有效性检查，并在必要时对指针进行初始化。
2. 检查是否有未初始化的变量，并确保在使用它们之前正确初始化。
3. 检查代码中的递归或大量函数调用情况，确保不会超出系统限制。
4. 检查是否存在重复释放已释放的内存的情况，确保每个内存块只被释放一次。



总结

通过AStar算法的实现，加深了对A*搜索的理解，以及IDAstar算法的理解，但是发现迭代Astar并不是在任何情况能够取得比Astar更好的效果，并且感觉启发式函数的选择对Astar算法影响非常大，也很难取得一个又快又能够找到最短路径的启发式函数，也就是找到一个能够完美估计实际代价的函数，在过程主要遇到的问题就是指针的使用要十分小心，在函数传递时传递的引用类型也需要注意，本次Astar实现的的代码量大概在670行左右，虽然用的数据是结构体，但是这个过程也是体现了面向对象的思想，对于用C++实现一个稍大规模的编程设计也加强了理解。

CSP

变量集合、值域集合以及约束集合

变量集合：

每一天的每一班是一个变量，变量数量为D*S，D为天数，S为每天的班数

值域：

用“班次”代表每一天的每一班，则班次的值域为所有阿姨的编号，初始为-1表示未赋值，有N个阿姨，则值域为{0,1,2,...,N}。

约束集合

- **每天分为轮班次数个值班班次**：
  1. 在读入数据时，就初始化需要填充的排班表为一个D*S的矩阵，D为天数，S为轮班次数
- **不连续值班**：(i,j)表示第i天第j班
  1. 两天交界处：(i,j)为前一天最后一班
  2. 两天交界处：(i,j)第二天第一班
  3. 同一天连续两班： 前一班和(i,j)相同
  4. 同一天连续两班： 后一班和(i,j)相同
- **公平性**：
  1. 每个宿管阿姨在整个排班周期中，应至少被分配到$\lfloor \frac{D \cdot S}{N} \rfloor$(向下取整) 次值班：对于得到的排班表，统计每个阿姨值班的次数，如果小于该阈值，则该值班方案不合法。
- **每一班都有阿姨值班**：
  1. 检查得到排班表：是否每个班次都有阿姨值班，即每个班次的阿姨编号都 >=0，如果没赋值则为 -1
- **尽可能最大化满足的请求数**：
  1. 初始设置每个班次的剩余值为对应有请求的阿姨编号，如果剩余值为空，再去寻找是否还有存在请求但未满足的阿姨，虽然她没有请求当前班次，但当前班次没有阿姨值班，必须要选一个阿姨值班

算法的主要思路

- **数据结构**：为了更直观地操作变量和赋值，将阿姨和每天的排班分别设计为两个结构体
  1. 阿姨：Aunt只有一个属性，request：阿姨的请求列表，是一个二元组的容器，存放对应阿姨的请求班次(i,j)：第i天第j班。这里阿姨不需要阿姨编号属性，因为后续将阿姨放入一个数组中，对应的下标就是阿姨编号。
  2. 班次信息：DayShift有两个属性，分别是auntNo：值班阿姨编号，remainValue：剩余可选的阿姨，用一个数组表示。
  3. 初始化：从文件中读取矩阵，根据N的值分割矩阵，得到每个阿姨的请求矩阵，再遍历每个阿姨的为1的坐标，存入阿姨的请求列表中，排班表根据阿姨的请求列表初始化，对应的阿姨编号初始化为-1。

```
//结构体：阿姨
struct Aunt {
    vector<pair<int,int>>   request;//阿姨的请求列表
    Aunt() {
        request = vector<pair<int,int>>request();
    }
};
//结构体：某天的排班信息
struct DayShift{
    int auntNo; //值班的阿姨编号
    vector<int> remainValue; //剩余可选的阿姨
    DayShift(){
        auntNo = -1;
        remainValue  = vector<int>();
    }
};
```

- **算法**：回溯搜索+启发式
  1. 递归实现回溯搜索：传入的主要参数为排班表和阿姨的列表，在递归入口检查约束，如果满足则返回true，否则选择一种获取变量的方法，有枚举、minimum remain value、mrv+least constraint value、mrv + lcv +constraint propagation，返回值为一个班次(i,j)，在启发式方法中，还会修改对应班次的剩余值取值范围以及剩余值选择顺序等，选择好要赋值的变量后，就遍历该变量的剩余值，对每个取值检查是否与当前已选值相容，即阿姨不连续上班，如果满足的话，需要删去对应的剩余值取值，以及该取值(即阿姨)请求列表中的对应请求(i,j)，然后再进入下一个分支，即继续递归，前向检验的思想在回溯算法中体现，即不相容则当前分支提前失败，返回false。
  2. 获取变量的方式分别为enumerate、mrv、mrv+lcv、mrv+lcv+cp，其中mrv选择最少阿姨请求的班次，lcv+mrv先将最少剩余的变量选出来再对变量的剩余值进行排序，mrv+lcv+cp则将变量的剩余值根据阿姨已经排班数从小到大排序。

优化方法

1. 为了能够最大化满足阿姨的请求数，选择对每个班次剩余值设置为对应请求该班次的阿姨编号，这样在使得变量有取值的同时能够优先满足请求。
2. 一开始将检查是否连续上班放在约束检查中，即得到完整方案才检查，发现可以通过约束传播，在每次选取变量赋值时就检查，这样能够更快发现失败的情况。
3. 检查值是否相容，提前发现失败的情况，如果当前位置取值会导致阿姨连续值班则直接返回false，而不用继续赋值。
4. 使用最少剩余值选择变量进行赋值，即优先选择阿姨请求最少的班次。

5. 最少约束值，班次有多个取值可选，此时就考虑最少排班的阿姨，这样就能够减少对其他阿姨选择的约束，因为每个阿姨都需要满足一定的阈值。

优化效果

Enumerate

输入文件	运行结果	运行时间	遍历的方案数	满足请求数
input0.txt	Find A Solution !	0.003s	102	20/21
input1.txt	Run timed out !	22.745s	98414	28/60
input2.txt	Find A Solution !	0.026s	98502	33/33
input3.txt	Find A Solution !	10.299s	126162	114/114
input4.txt	Find A Solution !	0.026s	126234	69/69
input5.txt	Run timed out !	23.217s	127050	291/576
input6.txt	Run timed out !	66.725s	128157	503/1008
input7.txt	Run timed out !	18.301s	129351	191/378
input8.txt	Run timed out !	495.891s	131532	1068/2160
input9.txt	Run timedout !	103.592s	132288	352/720

Minimun Remain Value

输入文件	运行结果	运行时间	遍历的方案数	满足请求数
满足请求数input0.txt	Find A Solution !	0.002s	51	20/21
满足请求数input1.txt	Run timed out !	138.397s	34499	28/60
满足请求数input2.txt	Find A Solution !	0.085s	34683	33/33
满足请求数input3.txt	Find A Solution !	0.436s	35361	114/114
满足请求数input4.txt	Find A Solution !	0.036s	35448	69/69
满足请求数input5.txt	Run timed out !	231.395s	36213	291/576
满足请求数input6.txt	Run timed out !	163.298s	37300	503/1008
满足请求数input7.txt	Run timed out !	142.87s	38180	191/378

输入文件	运行结果	运行时间	遍历的方案数	满足请求数
满足请求数input8.txt	Run timed out !	512.216s	40359	1068/2160
满足请求数input9.txt	Run timedout !	103.977s	41109	352/720

MRV+Least Constraining Value

输入文件	运行结果	运行时间	遍历的方案数	满足请求数
input0.txt	Find A Solution !	0.005s	140	20/21
input1.txt	Find A Solution !	0.007s	200	60/60
input2.txt	Find A Solution !	0.002s	233	33/33
input3.txt	Find A Solution !	0.016s	347	114/114
input4.txt	Find A Solution !	0.009s	416	69/69
input5.txt	Run timed out !	130.652s	1147	291/576
input6.txt	Find A Solution !	3.425s	2155	1008/1008
input7.txt	Find A Solution !	0.377s	2533	378/378
input8.txt	Run timed out !	417.556s	4705	1068/2160
input9.txt	Find A Solution !	3.173s	5425	720/720

MRV+Least Constraining Value+Constraint Propagation

输入文件	运行结果	运行时间	遍历的方案数	满足请求数
input0.txt	Find A Solution	0s	21	20/21
input1.txt	Find A Solution	0.006s	81	60/60
input2.txt	Find A Solution	0.002s	114	33/33
input3.txt	Find A Solution	0.018s	228	114/114
input4.txt	Find A Solution	0.007s	297	69/69
input5.txt	Find A Solution	1.058s	873	576/576

输入文件	运行结果	运行时间	遍历的方案数	满足请求数
input6.txt	Find A Solution	3.394s	1881	1008/1008
input7.txt	Find A Solution	0.345s	2259	378/378
input8.txt	Find A Solution	89.495s	4508	2160/2160
input9.txt	Find A Solution	3.125s	5228	720/720

input0.txt 的安排方式

在枚举选取变量进行回溯搜索的方式下，取得的安排方式为：

```
1,2,1
2,3,1
3,2,1
3,2,3
1,2,1
3,2,3
1,2,3
20
```

在后续几种方式中，取得的安排方式为：

```
1,2,3
1,3,2
3,2,1
3,1,2
1,2,1
3,2,3
2,1,3
20
```

Debug

1. 一开始发现算法连第一个都没法跑出来，通过debug打印过程中一些变量的信息，发现得到的排班方案总是有几个班次没有阿姨排班，检查程序发现，由于最初设置的每个班次剩余值为对该班次有请求的阿姨编号，自己的算法对于每个班次在选择剩余值的时候，如果该班次没有阿姨请求，那么就没有剩余值可选，就会返回false，并且在第一个样例中，恰好有一个班次所有阿姨都没有请求，所以在这种情况下，需要对剩余值为空的变量补充剩余值，具体方法是将还有请求的阿姨加入剩余值中。
2. 调整完毕后，发现自己的算法除了第一个后续都是一直跑不出来，但自己的启发式方法并没有设计错误，也想不到有什么还能更优化的方法，所以一度陷入的瓶颈，感觉自己甚至可能没法做出这个实验了。后来通过调试发现，自己的算法一直无法找到满足每个阿姨最小值班阈值的约束，但是打印出不满足的阿姨的值班次数发现，很多阿姨的值班次数其实已经满足要求，但是被判定为不满足，于是发现自己将$\lfloor\frac{D\cdot S}{N}\rfloor$在代码中写成了$\lfloor\frac{D\cdot N}{S}\rfloor$，

这导致后续的N都比S更大，阈值也就大很多，并且在第一个样例中，S和N相等，所以只有第一个样例可以通过！这也是困扰了我最久的一个bug，往往越简单的bug要找最久，因为它的影响比较不明显！

3. 在回溯的递归中，最初的枚举不是通过函数来获取一个对应的(i，j)，而是直接在回溯算法的递归程序中，使用双重for循环，遍历D*S个(i，j)取值，这导致搜索空间非常大并且存在非常多的重复状态！即使第一个很简单的样例也要运行很久！所以修改这个取值方式，算法效率得到很大改善。
4. 对于vector的使用，要非常注意初始化的分配空间大小，并且有时候很容易在循环遍历vector时出现越界的错误。

总结与体会

1. 通过这次实验，熟悉了CSP问题的定义和结构，进一步理解了什么是约束满足问题，发现人工智能的本质原来就是for循环+if语句。
2. 发现原来用最朴素的搜索算法加上一些针对问题设计的启发式方法，也能够很快地解决一些看起来很复杂的问题，而不一定需要设计多么精巧的数据结构或者使用很多trick的算法。
3. 并且有时候问题看起来很复杂，实际上真正去解决时，很多复杂的情况其实并不会遇到，或者遇到的概率很小，这在很大程度上降低了问题的复杂程度，让我的算法不需要优化到多么高效的地步。
4. 使用的回溯算法让我进一步理解了递归，虽然感觉递归还是不容易理解，比较难在脑海中debug递归的运行过程，但编程的过程还是帮助我加深了对递归的理解。
5. 和A*实验对比，这个实验我用的指针操作等比较少，出现的报错也更少一点，空指针和指针异常的报错很让人头疼。
6. 我个人认为我的代码结构比较容易理解，查看一些csp问题的代码示例，发现它们的结构大部分都是直接在同一个类里面定义各个属性及方法，这样就可以统一在一个域下发挥作用，也比较容易管理，减少了各种传参以及引用之类的，减少了代码的复杂度和出错的可能性，有值得我学习的地方，还有就是虽然这个问题比较简单，但是第一次编程实现csp问题，感觉主要困难是在于如何理解csp问题，当理解之后，代码的逻辑实现其实比较简单，而我认为如果问题更复杂的话，那么编写代码需要用到的优化方法也会更复杂。