# 实验 1.1

# 一、问题描述

$N \times N$ 的拨轮锁盘，每一个格点上有一个可转动的拨轮，上面刻着数字，1 表示锁定，0 表示非锁定。

只能同时转动相邻呈 "L" 字形的三个拨轮，将其数字翻转。

目标：

- 设计一个合适的启发式函数，证明是 admissible 的，并论证是否满足 consistent
- 根据上述启发式函数，设计 A* 算法找到一个解
- 设置启发式函数为 0，此时算法退化为 Dijkstra，比较并分析使用 A* 算法的优化效果

**输入**：在 `astar/input` 下共 10 个文件

**输出**：在 `astar/output` 下，对应 10 个文件，若不能解锁，输出 “No valid solution”，否则输出解锁步骤。

```
Line 1: 第一行包含一个正整数：T，表示解锁所需的步骤数
Line 2 to T+1: 每行包含三个整数 i、j、s，用 ',' 隔开，表示该步切换的中间拨轮位置为第 i 行，第 j 列，朝向为第 s 种。
```

四种朝向和数字的映射关系为：

```
s == 1: (i,j), ( i ,j+1), (i-1, j );
s == 2: (i,j), (i-1, j ), ( i ,j-1);
s == 3: (i,j), ( i ,j-1), (i+1, j );
s == 4: (i,j), (i+1, j ), ( i ,j+1).
```

# 二、设计启发式函数

## 2.1 第一种设计

$h(n)$ = 当前状态中拨轮为 1 的数量除以 3，然后上取整。

**证明是 admissible**

首先 $h(n) \geq 0$

由操作的设定可知，每次转动相邻的 “L” 形的三个转轮，**至多**将三个 1 变成 0，所以对任意状态 $n$，若当前锁盘上有 $x$ 个 1，至少需要 $\lceil \frac{x}{3} \rceil$ 次转动，即 $h^*(n) \geq \lceil \frac{x}{3} \rceil = h(n)$

所以是 admissible 的

**证明是 consistent**

由操作的设定可知，每次转动相邻的 “L” 形的三个转轮，**至多**将三个 1 变成 0，所以对任意状态 $n$，若当前锁盘上有 $x$ 个 1，转动一次后得到其后继结点 $n'$，其后继节点至少有 $x-3$ 个 1.

所以 $h(n') + c(n,a,n') \geq \lceil \frac{x-3}{3} \rceil + 1 = \lceil \frac{x}{3} \rceil = h(n)$

所以是 consistent 的.

**效果**

这个启发式函数可以运行出前九个测试样例。最后一个测试样例会因为使用的空间过大而报错。

## 2.2 第二种设计

首先，有一个简单的观察，如果一组 “1” 被 0 包围，那么这组 “1” 与其他 “1” 可以视为独立的。即考虑八联通分量，八联通，是指考虑上下左右，左上左下右上右下八个方向视为相邻，一个由 “1” 组成的八联通分量彼此可以看作独立，可以独立考虑。

这种情况下，我们可以优化 2.1 中的启发式：

$h(n)$ = 每个八联通分量中的 1 的个数除以 3 上取整，再求和。

**证明是 admissible**

首先 $h(n) \geq 0$

由操作的设定可知，每次转动相邻的 “L” 形的三个转轮，**至多**将三个 1 变成 0，而且一个 L 性操作的三个数字都是相邻的（八联通意义下），所以每个 L 行操作只会对一个八联通分量有影响。所以对当前状态的每个八联通，可以独立考虑，每个八联通至少需要 1 的个数除以 3 上取整次操作。

所以是 admissible 的

**证明是 consistent**

对于一个 L 行操纵，分两种情况：

第一种情况，如果该操作只对一个八联通分量有影响，即该操作不会使得原本的两个八联通分量相连，那么执行该操作至多使得一个八联通分量减少 3 个 1，与 2.1 中的分析类似地可以证明，执行该操作后，其他八联通分量没有变化， 当前八联通分量的 1 的个数记为 x，
$h(n') + c(n, a, n') \geq \Sigma(\text{其他八联通分量}) + \lceil \frac{x-3}{3} \rceil + 1 = \Sigma(\text{其他八联通分量}) + \lceil \frac{x}{3} \rceil = h(n)$

第二种情况，如果该操作后，连接了原本不同的八联通分量，那么该操作至少使得一个 0 变成 1，即使他使得两个 1 变成 0，记这两个八联通分量的 1 的个数为 x，x'，那么：

$$h(n') + c(n, a, n') \geq \Sigma(\text{其他八联通分量}) + \lceil \frac{x-1}{3} \rceil + \lceil \frac{x'-1}{3} \rceil + 1 = \Sigma(\text{其他八联通分量}) + \lceil \frac{x+x'-1}{3} \rceil \geq h(n)$$

所以是 consistent 的。

**效果**

可以运行出全部测试样例。

# 三、设计 A* 算法

## 3.1 结点的表示

每个结点用一个二维数组表示，描述当前锁的转台，数组的元素为 0 或 1，分别表示未锁定和锁定。

在实际代码中，还引入了其他变量来描述一个结点：

```
struct Node {
    int board[MAXN][MAXN];
    int f = 0, g = 0, h = 0; // f = g + h
    int xx = 0, yy = 0, ss = 0;
    Node* parent = nullptr;
};
```

`board` 就是表示当前锁的状态的二维数组。

f、g、h：A* 算法需要的三个参数，g 是到当前节点的 cost，h 是启发函数，f = g + h

xx、yy、ss：分别是当前节点由其父节点转换而来时，其父结点做了什么操作，即最后要输出的坐标和朝向。

**启发函数求解：**

对于第一种启发函数，直接求所有 1 的数量，除以 3 上取整。

对于第二种启发函数，采用 DFS，寻找所有八联通分量，对每个八联通分量求 1 的数量，除以 3 上取整，再求和。

## 3.2 边的表示

边没有显示的表示，而是尝试对当前节点做全部的 L 形变换，得到的二维数组即为当前点的相邻点。

## 3.3 核心算法

伪代码为：

```
open list = [start node]
do {
    if open list is empty then {
        return no solution
    }
    n = heuristic best node
    if n == final node then {
        return path from start to goal node
    }
    for each direct available node do {
        add current node to open list and calculate heuristic
        set n as his parent node
    }
} while(open list is not empty)
```

Open List 采用**优先队列**实现，按照每个结点的 f 从小到大排序，这样每次 while 循环都是取队首元素。

## 3.4 剪枝

注意到，假如存在一个最优执行操作序列，可以使得当前状态变换到目标状态，这个序列是由一系列 L 形操作组成，那么这些 L 形操作的顺序是可以任意调换的。

那么，从可以选择当前状态最左上的 1，最优操作序列中一定存在一个 L 形操作使得这个 1 变成 0，又因为操作是顺序无关的，那么对于每个结点，可以只扩展使得其最左上的 1 变成 0 的 12 种 L 性操作。这 12 种操作中必有一种位于最优操作序列中。

这个剪枝的效果是巨大的，尤其是在内存的节省上。在引入这个剪枝之前，只能运行出 $n < 9$ 的测试样例，即使引入存储受限的操作放弃寻找最优解，也无法求解 $n > 9$ 的样例。而引入这个剪枝后，无需进行存储受限的操作，可以在秒级别的时间内寻找出最优解。

## 3.5 输出

每个结点都记录了当前节点由其父节点转换而来时，其父结点做了什么操作，即最后要输出的坐标和朝向。并且每个结点都记录了当前搜索路径上其父节点。

那么如果找到全零的状态，那么沿其父节点的路径，就是操作的逆序列，所以只需要用一个栈保存，再输出，就是从初始状态开始的操作序列。

## 3.6 其他被弃用的尝试

在采用 3.4 的剪枝之前，曾尝试过多种技巧来求解本问题，以下列出的技巧在最后版本的代码中没有使用，只是记录如下。

但是我在附件中附上了被弃用版本的代码，其名字中带有 `_DEPRECATED`。

### 3.6.1 IDA*

即迭代加深的 A star 算法，即在 DFS 搜索的过程中，设置一个阈值，每次搜索深度不超过这个阈值。然后不断提到阈值，直到找到解。

在实际代码中，就是将 f 值作为阈值，初始时是将 start 结点的 f 作为阈值。

在搜索过程中，如果当前节点的 f 值大于阈值，则直接返回，return 当前节点的阈值。如果当前节点 f 值不大于阈值，则可以扩展其节点，返回其所有子节点路径上的返回值的最小值。

一轮搜索结束后，以返回值作为下一轮迭代搜索的阈值。

**实际效果：**比朴素算法速度要快一些，空间上也更节省。

### 3.6.2 采用 Graph Search

对于每个节点，计算一个 hash 值，该值是当前节点的二维矩阵展开成一维对应的二进制数，在模上一个大素数。每次扩展结点时，如果该带扩展结点计算出的 hash 值已经存储过，则不扩展。

本意是希望采用这个方法降低空间需求，但是对于 n >= 9 时，对应的二位数字可以达到 $2^{81}$，远大于我是用的大素数，所以会出现 hash 碰撞。

### 3.6.3 存储受限

在扩展结点时，不全部扩展。

如果当前节点存在一个 L 形全 1 的局部，则对其执行 L 操作，并扩展结点。对于每个节点，它的这类相邻节点只扩展 2 个，若超过 2 个，则直接 return.

如果上面那类结点数目没达到 2 个，则考虑当前节点存在一个 L 形有 2 个 1 的局部，则对其执行 L 操作，并扩展结点。对于每个节点，它的这类相邻节点加上第一种只扩展 10 个，若超过 10 个，则直接 return.

如果上面两种结点没超过 10 个，则考虑当前节点存在一个 L 形有 1 个 1 的局部，则对其执行 L 操作，并扩展结点，三种节点总数不超过 20 个。即使最后搜索结束也没有超过 20 个也 return。因为不需要对全零的局部进行 L 操作。

# 四、运行结果

最终输出的步数：

| 输入 | input0 | input1 | input2 | input3 | input4 | input5 | input6 | input7 | input8 | input9 |
|---|---|---|---|---|---|---|---|---|---|---|
| steps | 5 | 4 | 5 | 7 | 7 | 7 | 11 | 14 | 16 | 23 |

# 五、与 Dijkstra 对比

运行时间比较：时间单位：毫秒 ms

| 输入 | input0 | input1 | input2 | input3 | input4 | input5 | input6 | input7 | input8 | input9 |
|---|---|---|---|---|---|---|---|---|---|---|
| n | 5 | 3 | 4 | 5 | 5 | 5 | 9 | 9 | 10 | 12 |
| A star | 1.33 | 0.95 | 0.67 | 1.65 | 1.33 | 4.28 | 47.11 | 39.77 | 13.15 | 3610 |
| Dijkstra | 17.05 | 3.11 | 12.56 | 1110 | 2215 | 2230 | | | | |

可以看出，A star 算法的时间性能要明显优于普通的 Dijkstra 算法，在输入规模比较大时差别尤为明显，在 n = 5 时有三个数量级的差别，在 n >= 9 后，普通的 Dijkstra 算法会因为内存不够而无法求出结果。

# 实验 1.2

# 一、问题描述

获得以下信息：宿管阿姨数量 N，值班天数 D，每日轮班次数 S，轮班请求 $Request \subset \{0,1\}^{N \times D \times S}$

轮班表必须满足：

1. 每天分为 S 个值班班次
2. 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
3. 每个宿管阿姨在整个排班周期中，应至少被分配到 $\lfloor \frac{D \cdot S}{N} \rfloor$ 次排班

目标：

- 构造一个排班表 $Shifts \subset \{0,1\}^{N \times D \times S}$
- 在满足上述约束的条件下，尽可能最大化满足请求数

输入格式：

```
Line 1: 第一行包含三个正整数：N、D、S。 200，400，6
Line 2 to N*D+1: 接下来的 N*D 行，每行包含 S 个整数，表示轮班请求（Requests）。轮班请求的每个值取 0 或 1。若第 n 位宿管阿姨请求值第 d 天的第 s 轮班，则第 2+n*D+d 行的第 s 个数字为 1；否则为 0。
```

输出格式：

- 如果没有有效排班表，请输出 “No valid schedule found.”。
- 否则，输出排班表 Shifts 如下：

你的算法应该利用最小剩余值（Minimum Remaining Values，MRV）启发式、前向检查（Forward Checking）或约束传播（Constraint Propagation）等优化技术，以快速解决该问题，并最大化满足的请求数。

# 二、算法设计

## 2.0 基本概念

### 2.0.1 最小剩余价值

Minimum Remaining Values，MRV。每次选择有最少可选值的变量。

### 2.0.2 前向检验

> Keep track of remaining legal values for unassigned variables
> Terminate search when any variable has no legal values

记录所有未被赋值的变量有多少可选值，如果某个未赋值变量的可选值为空，那么当前搜索终止。

### 2.0.3 约束传播

> using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on

## 2.1 算法中的变量设置

以下设计中，并没有区分“天”，即将所有班次排成一维。只需在输入输出时做一些处理，但只这样设计可以在考虑“相邻班次不同阿姨”时更简便。

```
int N, D, S; // N 个阿姨，D 个值班天数，每日轮班次数 S
bool request[2400][200]; // 第 i 个班次，第 j 个阿姨是否申请，若申请则为 1
int shifts[2400];   // 当前搜索到的排版安排
int counts[200];    // 每个阿姨的已经排班数目
int ans = -1;       // 已经搜索到的最优解的满足数
int ans_shifts[2400];   // 已经搜索到的最优解的排版安排，即每个班次的
int base;           // 每个阿姨最终被排班数目的下界
int Top;            // 每个阿姨最终被排班数目的上界
int resum[2400];    // 后缀和，剪枝用，具体请见 2.2.3
bool HaveRequests[2400];    // 某个班次是否有申请
int begin_index;    // 开始搜索的起点，具体请见 2.2.4
int turn;           // 两种策略，具体见 2.3.3
```

## 2.2 核心算法

### 2.2.1 CSP 主体部分

按时间顺序给各个班次选择阿姨。

必须要满足的条件只有，即**约束集合**:

1. 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
2. 每个宿管阿姨在整个排班周期中，应至少被分配到 $\lfloor\frac{D\cdot S}{N}\rfloor$ 次排班

**先只考虑第一个约束**，每个班次可选的阿姨只会被其相邻班次的选择的阿姨影响，那么顺序排班，排完第 i 班后，第 i + 1 班只有 N - 1 种选择，而第 i + 2 以及之后的班次均有 N 中选择，所以顺序排班符合了**约束传播和 MRV 启发式。**

再考虑进第二个约束，因为每个阿姨应至少被分配到 $\lfloor\frac{D\cdot S}{N}\rfloor$ 次排班，那么如果一个阿姨被排班次数超过了 $\lfloor\frac{D\cdot S}{N}\rfloor + D\cdot S - N\cdot\lfloor\frac{D\cdot S}{N}\rfloor$ 次，那么一定有阿姨的排班次数不满足最低要求，这是应用了**前向检验**的思想。

此外，对于第二个约束，顺序给每个班次安排阿姨时，优先考虑排班数最少的阿姨，以实现要求的均匀排班。

在顺序搜索的前提下，递归搜索函数 Search 需要的参数有:

- `num`: 当前正在排的班次，当 num = D*S 时一次排班完成，更新答案。
- `pre`: 前一个班次安排的阿姨，当前安排的阿姨不能与 pre 相同。
- `cnt`: 截至到当前排班，已经满足的要求数目

### 2.2.2 寻找尽可能优的解

如上面提到的，如果当前 num == D*S，那么说明完成了一次排班，但是未必符合“每个阿姨应至少被分配到 $\lfloor\frac{D\cdot S}{N}\rfloor$ 次排班”，所以需要做一次检查，若满足这个条件，那么就将当前 cnt 与存储的 ans 进行比较，若 cnt > ans，则更新答案。

### 2.2.3 剪枝

按照上述过程，理论上是可以搜索到所有解的，也就能通过比较找到最优解。但是分析可知，如果不加入任何剪枝，那么顺序搜索的时间复杂度为 $O((D\cdot S)^N)$，共 D*S 个班次，每个班次有 N 或 N - 1 个阿姨考虑选择。这个时间复杂度在输入规模稍微大一些时就搜不出结果。

那么需要考虑剪枝。

最核心的剪枝是提前结束。与前向检验不同的是，前向检验是有未赋值变量的可选值为空时结束，但是这里应用到的剪枝是:

记录一个后缀和，resum[i] 表示如果 i 和 i 之后的排班都是“按照申请去满足的”，即如果一个排班有申请，那么默认其可以得到满足。可以看出，resum[i] 维护了一个上界，是从当前排班 i 以及之后，可以增加的满足数的上界。

那么在搜索到第 i 个排班时，如果 i + resum[i] <= ans，也就是说即使从这个点开始所有有申请的班次都按申请满足了，也不会超过之前搜索到的最大可满足树，那么就无需继续搜索，因为沿这条路径不会找到更优的解。

**这个剪枝的效果是巨大的，在进行这个剪枝之前，只有 input0.txt 能跑出来，其他的均不可。加入这个剪枝后，本次实验给定的所有输入均可跑出来，且时间均在毫秒级。**

### 2.2.4 搜索起点设置

这一部分**在最后的代码中并没有使用**。原本是采用了 2.2.1、2.2.2、2.2.3 的设计，但是不包含 2.2.1 中每个班次优先选择排班数最少的阿姨，此时对于后几个输入是跑不出来的，为了寻找可行解，采用了 2.2.4 节的策略。但是在应用 2.2.1、2.2.2、2.2.3 的全部设计后，是可以跑出最优解的，所以 2.2.4 不再采用，以下是原本的设计。

然而即使是应用了 2.2.3 中提到的两个方法，还有四个输入是无法跑出最优解，甚至是无法搜索结果的。这四个文件也比较有代表性，那就是**规模比较大的输入**。即使加上了 2.2.3 的技巧，其搜索空间仍然是巨大的，无法在合理的时间内搜索出哪怕是一个可行解。

但是分析题目的特点，如果仅考虑两个必须要满足的限制，是可以构造可行解的。

即**所有阿姨严格轮班。**

那么以这个构造解作为搜索起点，进行搜索。最起码是可以找到可行解的，在这个可行解的基础上，应用上面提到的剪枝技巧，尽可能搜索更优的解。

受这个思想的启发，比较自然的一个优化就是，怎么轮班。

如果有三个阿姨，那么可以 012012012 这个顺序轮班，也可以 120120120 这样轮班，总选择数目为 N，即轮班起点的选择。那么可以搜索所有 N 个轮班选择，时间为 $O(NDS)$，选择最优的轮班作为起点进行搜索，以此为基础找更优解。

## 2.3 实际运行与结果

### 2.3.1 输入

输入需要手动输入一个字符串，即输入的文件名。

逐行读入文件，用逗号作为分隔符，读入到相应的变量。

对于读到的“第 i 个阿姨申请第 j 天第 k 个班次”，写入数组 `request[j * S + k][i] = 1`，其中 S 为每天的班次。

### 2.3.2 输出

最终得到的是：

`ans`：当前搜索到的最优的方案的总满足数

`ans_shifts[i]`：第 i 个班次安排的阿姨。这个在输出时需要转换为天和班次。与输入是类似的。

注意到，**我的下标选取是从 0 开始的，即阿姨标号从 0 到 N - 1**

### 2.3.3 最终输出

以 input0.txt 的输出为例：

```
0,1,2
0,2,1
2,1,0
2,0,1
0,1,0
2,1,2
0,1,2
20
```

即第一天按 0、1、2 的顺序安排阿姨，第二天按 0、2、1 顺序安排，以此类推。最终可以满足的数目为 20.

综上，最终结果如下：

| 输入 | input0 | input1 | input2 | input3 | input4 | input5 | input6 | input7 | input8 | input9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 满足数 | 20 | 60 | 33 | 114 | 69 | 576 | 1008 | 378 | 2160 | 720 |
| 运行时间（毫秒） | 0 | 1 | 1 | 2 | 2 | 97 | 198 | 56 | 601 | 326 |

**可满足数的上界是所有有申请的班次数目，所以可知，以上的输出达到上界，所以是最优解。**

## 2.4 最初的设计版本

这一部分是描述我最初没有想到可以顺序遍历各个班次，而是每次遍历所有班次，选择可选阿姨数最少的。这一版的设计不够简洁，但这设计过程中确实也发现了不少供最终版本参考的内容，所以也把这个版本的核心设计列在下面，注意到，**这个版本最后被弃用**：

**先满足必须的要求，不考虑阿姨的需求。**

递归设计：

使用 MRV 启发式选择一个还没有分配足够多班次的宿管阿姨。

对于该阿姨，查找这一天中可用的班次，满足：

- 该班次为空
- 安排该宿管阿姨不会导致她在连续两个班次工作

安排阿姨在这个班次，填写在 Shifts 表中，然后递归搜索。

若有完整的安排表列出，则递归终止。

若有班次找不到符合条件的阿姨，即无法排班，则递归回溯。

结点是班次。

递归搜索分为两种。

先进行第一种，只考虑有申请的班次，每次选择申请人数最少的班次，尝试满足，然后递归。

当考虑完所有有申请的班次后，通常是递归达到一定的深度以后，进行第二种搜索，不考虑申请， 此时搜索的唯一目标就是寻找可行解，每次选择可选阿姨数最少的未排班的节点，尝试排班选择，然后递归搜索。

注意到，第二种搜索是跟在第一种搜索之后的，通常在递归深度比较浅的时候是第一种搜索，随着深度加深，会转化为第二种搜索。

此外，考虑有申请的班次时，不仅需要考虑安排那些申请了这个班次的阿姨，也要考虑不安排那些申请这些班次的阿姨，而如果是后者，那么这个节点在第一种搜索完成后仍没有排班，那么会进入到第二种搜索，此时将不考虑任何申请。

无论是第一种搜索还是第二种搜索，在搜索开始前，都有可能遇到：

- 所有班次均被排班，即找到一个可行解。那么就与原来存储的最佳可行解（初始为空）进行比较，若更优，那么更新答案。
- 有班次没被排班，但是它的可选阿姨数为零，即该班次不可能被排班，即**前向检验**不通过，当前搜索返回。

总结来说，这个版本因为没有发掘到题目的特性，没有采用顺序搜索，导致很多处设计比较麻烦，需要额外引入数据结构，比如结构体。

这个版本的代码我也附在了附件中，标记为 `csp_DEPRECATED.cpp`

# 附录

```
.
|-- PB20061343_\320\354\260\302_report.pdf
|-- astar
|   |-- output
|   |   |-- output0.txt
|   |   |-- output1.txt
|   |   |-- output2.txt
|   |   |-- output3.txt
|   |   |-- output4.txt
|   |   |-- output5.txt
|   |   |-- output6.txt
|   |   |-- output7.txt
|   |   |-- output8.txt
|   |   `-- output9.txt
|   `-- src
|       |-- IDAstar_DEPRECATED.cpp
|       |-- astar.cpp
|       `-- astar_DEPRECATED.cpp
`-- csp
    |-- output
    |   |-- output0.txt
    |   |-- output1.txt
    |   |-- output2.txt
    |   |-- output3.txt
    |   |-- output4.txt
    |   |-- output5.txt
    |   |-- output6.txt
    |   |-- output7.txt
    |   |-- output8.txt
    |   `-- output9.txt
    `-- src
        |-- csp.cpp
        `-- csp_DEPRECATED.cpp
```

注：在源文件中，带有 `_DEPRECATED` 是尝试过但是已经启用的版本。