实验 1.1 Report

1 启发式函数

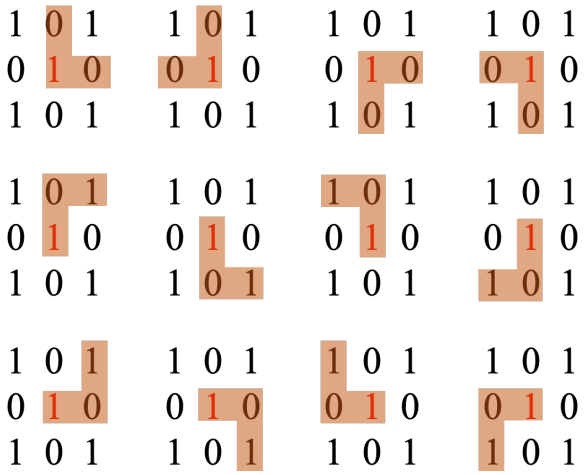
针对这个问题，如果设状态 s 的 1 的数量为 c_s，使用 $f_1(s) = c_s/3$ 作为启发式函数。

显然 f_1 是可采纳的，因为一次操作最多将 3 个 1 变成 0，将状态 s 所有的 1 变成 0 至少需要 $c_s/3$ 步，不会过高估计。

这个启发式是一致的，因为 $c(n, a, n') = 1$，且从 n 到 n'，依据后文所述的算法规则，$c_{n'} - c_n \in [-3, 1]$，所以 $-1 \leq h(n') - h(n) \leq \frac{1}{3}$，即 $h(n) \leq 1 + h(n') = c(n, a, n') + h(n')$。

2 算法的主要思路

- 算法记录所有已经搜索过的状态表示，防止重复搜索；
- 算法对于没达到终点的状态，找到其中一个 1 的位置（在图中以红色标记），以 12 种方式设法将这个 1 翻转成 0，以此作为后继入队。



3 启发式优化分析

在使用启发式函数的情况下，求解情况如下所示：

测试用例	0	1	2	3	4	5	6	7	8	9
求解步数	5	4	5	7	7	7	11	14	16	23
队列长度峰值	540	77	210	795	183	1522	28599	56348	104482	2309001

在不使用启发式函数的情况下，求解情况如下所示（「——」表示求不出解）：

测试用例	0	1	2	3	4	5	6	7	8	9
求解步数	5	4	5	7	7	7	——	——	——	——
队列长度峰值	2057	122	730	8018	5608	4707				

可以看到，引入启发式函数，不仅可以在规模相对较大时求出解，而且在求出解的情况下需要搜索的节点数更少。

4 编译运行说明

使用 C++ 20 标准编译，如：

```
g++ astar.cpp -o star -std=c++20
```

使用重定向算符在 Shell 中运行并获得结果，如：

```
./astar < ../input/input0.txt > ../output/output0.txt
```

实验 1.2 Report

1 实验集合描述

实验需要给每个班次 $Shift$ 安排阿姨，所以实验的变量集合是各班次 $Shift_j(j = 0, 1, \ldots, d \times s - 1)$，其中 d 是天数、s 是每天的班次数。

变量的值域集合是阿姨的编号，即 $\{0, 1, \ldots, n - 1\}$，其中 n 是阿姨数量。

约束集合：

1. 所有的班次都要排班：$\forall j, Shift_j \in [0, d \times s)$
2. 相邻位置取值不能相同：$Shift_j \neq Shift_{j+1}, \forall j \in [0, d \times s - 1)$
3. 所有阿姨至少排班 $M = \left\lfloor \frac{d \times s}{n} \right\rfloor$ 次：$\sum_{j=0}^{d \times s - 1} \mathbb{I}(Shift_j = k) \geq M, \forall k \in [0, n)$

2 算法的主要思路

算法综合使用 MRV 启发式、前向检查、约束传播等技术快速求解这个问题，使用一个栈进行任务节点的取出和生成。

大致过程如下（伪代码）：

```
stack<mission> stk; // 初始化一个栈
mission root_node;  // 生成根结点
while (!stk.empty) {
    // 取出一个任务
    auto current_mission = stk.top(); stk.pop();
    // 检查这个任务是否结束（满足约束 1、3）
    if (...) { ...; break; }
```

```
    // 选取一个班次（变量）进行排班
    //  优化启发式 1：当前没有被排班的、且要求在此班排班的阿姨数最少的班次优先
    shift& target_shift = current_mission.select();
    // 为当前变量选取阿姨排班列表
    vector<aunt> aunt_list = target_shift.select();
    // 按启发式排序列表
    //  优化启发式 2：未受约束的、排班数少的、且要求在此班排班的阿姨优先
    sort(aunt_list, compare);
    for (auto& name : aunt_list) {
        // 创建一个当前节点的拷贝
        mission new_mission = copy(current_mission);
        new_mission.assign(name);
        // 约束 2 传播
        new_mission.front.ban(name);
        new_mission.next.ban(name);
        // 新任务节点入栈
        stk.push(new_mission);
    }
}
```

3 使用的优化方法

使用了启发式优化、剪枝优化两类优化方式，优化的效果将以**满足率 > 栈的最大深度 > 能否解决问题**的优先级给出。

这里所说的**能否解决问题**指的是能否在数秒内给出答案。

3.1 启发式优化

3.1.1 启发式 1

启发式 1 是在选择排班的班次（变量）时让**当前没有被排班的、且要求在此班排班的阿姨数最少的班次优先**。

使用启发式 1 时的情况如下：

测试用例	0	1	2	3	4	5	6	7	8	9
满足率	20/21	60/60	33/33	114/114	69/69	575/576	1008/1008	378/378	2160/2160	720/720
栈的最大深度	25	543	596	1259	1109	46084	97778	22306	306724	139684
能否解决问题	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

不使用启发式 1，而是从第 0 班开始按顺序排的情况如下：

测试用例	0	1	2	3	4	5	6	7	8	9
满足率	20/21	60/60	33/33	114/114	69/69	576/576	1008/1008	378/378	2159/2160	720/720
栈的最大深度	23	542	596	1256	1106	46082	97778	223064	306722	139682
能否解决问题	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

经观察发现，在给定的测试样例上使用启发式 1 与否差别不大，主要的原因可能是测试样例有偏好，调试过程中也发现回溯次数很少，如果在回溯次数较大的测试样例上运行算法，启发式 1 可能会发挥较好的作用。

3.1.2 启发式 2

启发式 2 是在选择排班阿姨（为变量选择值）时让**未受约束的、排班数少的、且要求在此班排班的阿姨优先**。

使用启发式 2 时的情况如下：

测试用例	0	1	2	3	4	5	6	7	8	9
满足率	20/21	60/60	33/33	114/114	69/69	575/576	1008/1008	378/378	2160/2160	720/720
栈的最大深度	25	543	596	1259	1109	46084	97778	22306	306724	139684
能否解决问题	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

不使用启发式 2 的情况，而是随机选择一个要求在此班排班的阿姨，如无法做到，就随机选择一个未要求在此班排班的阿姨，这时的情况如下：

测试用例	0	1	2	3	4	5	6	7	8	9
满足率	18/21	—	—	—	—	—	—	—	—	—
栈的最大深度	23	—	—	—	—	—	—	—	—	—
能否解决问题	Y	N	N	N	N	N	N	N	N	N

可见启发式 2 对搜索非常重要，直接决定了问题能否在正常时间内完成搜索。

3.2 剪枝优化

3.2.1 约束上的剪枝

由约束 3，所有阿姨至少排班 $M = \left\lfloor \dfrac{d \times s}{n} \right\rfloor$ 次，据此可以提出新的约束

4. 不能有某个阿姨排班次数超过 $M' = M + (d \times s - n \times M)$ 次

在约束传播时就进行约束 4 的检查，如果不满足就丢弃下一个要入栈的任务节点，这样做到了提前剪枝，有利于问题的快速解决。

使用约束剪枝优化时的情况如下：

测试用例	0	1	2	3	4	5	6	7	8	9
满足率	20/21	60/60	33/33	114/114	69/69	576/576	1008/1008	378/378	2159/2160	720/720
栈的最大深度	23	542	596	1256	1106	46082	97778	22304	306722	139682
能否解决问题	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

不使用剪枝优化，在任务完成时再检查约束 3 时的情况如下：

测试用例	0	1	2	3	4	5	6	7	8	9
满足率	20/21	60/60	33/33	114/114	69/69	575/576	1008/1008	378/378	2160/2160	720/720
栈的最大深度	25	543	596	1259	1109	46084	97778	22306	306724	139684
能否解决问题	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

经观察发现，在给定的测试样例上使用提前剪枝与否差别不大，主要的原因可能是测试样例有偏好，如果在比较难的测试样例上运行算法，提前剪枝可能会发挥较好的作用。

3.2.2 内存上的剪枝

因为节点的分支因子几乎全部大于 1，在使用启发式 1/2 的情况下，最不优先的节点可以慢慢淘汰，即将栈改为双向队列，每次取出栈顶任务时丢弃栈底若干个任务，这样可以减小最大的栈深度，节约内存使用。

使用内存剪枝优化时的情况如下：

测试用例	0	1	2	3	4	5	6	7	8	9
满足率	20/21	60/60	33/33	114/114	69/69	576/576	1008/1008	378/378	2159/2160	720/720
栈的最大深度	17	25	276	136	426	40332	87708	18534	285132	132492
能否解决问题	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

不使用内存剪枝优化的情况如下：

测试用例	0	1	2	3	4	5	6	7	8	9
满足率	20/21	60/60	33/33	114/114	69/69	575/576	1008/1008	378/378	2160/2160	720/720
栈的最大深度	25	543	596	1259	1109	46084	97778	22306	306724	139684
能否解决问题	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

虽然在许多测试样例上，内存剪枝都取得了较好的优化效果，但是还是需要指出，测试样例有偏好。如果在比较难的测试样例上运行算法，内存剪枝可能会剪去最优解所在的分支。

4 安排方式举例

使用 `input0.txt` 进行安排的结果如下：

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

5 编译运行说明

使用 C++ 20 标准编译，如：

```
g++ csp.cpp -o csp -std=c++20
```

使用重定向算符在 Shell 中运行并获得结果，如：

```
./csp < ../input/input0.txt > ../output/output0.txt
```