Lab1 实验报告

PB20111689 蓝俊玮

Lab1 实验报告
Lab 1.1
- 1. 启发式函数
- 2. 算法设计
  - 2.1 数据结构
  - 2.2 扩展状态
  - 2.3 A* 搜索
- 3. 探究比较

Lab 1.2
- 1. 实验描述
- 2. 算法设计
  - 2.1 数据结构
  - 2.2 回溯算法以及优化策略
  - 2.3 剪枝策略
- 3. `input0.txt` 安排方式

Lab 1.1

本次算法使用 A* 算法来求解二进制迷锁问题。在 A* 算法中，每个状态都有三个函数：耗散函数 $g(n)$ 表示从起点到当前状态 n 的耗散值，启发函数 $h(n)$ 表示从当前状态 n 到目标状态的耗散估计值，估计函数 $f(n)$ 表示经过状态 n 的最低耗散的估计值，即 $f(n) = g(n) + h(n)$。算法按照 $f(n)$ 的值从小到大扩展状态，直到找到目标状态或者已经扩展了所有的状态。

1. 启发式函数

在本次实验中，我规定了转动一次的耗散值为 3。原因为每次转动时会同时转动三个拨轮，将它们同时由各自的锁定切换为非锁定状态，或从非锁定切换为锁定状态。因此可以将每次操作的耗散值视为 3，即可以认为转动一个拨轮的耗散值为 1。

那么就可以定义启发式函数 $h(n)$ 定义为 1 的个数。可以证明，该启发式函数 $h(n)$ 是可采纳的。

假设在当前状态 n 下还有 $s(n)$ 个锁定状态的拨轮锁盘，即当前 1 的个数为 $s(n)$ 个，为了将这些拨轮转动成为非锁定状态，至少需要对每个拨轮转动一次，即转动 $s(n)$ 次。而由于我们的转动操作每次要同时转动三个相邻的拨轮，因此无法对一个单独的拨轮锁盘只使用 1 次耗散操作就完成目标。所以为了完成目标，我们至少需要耗散 $s(n)$ 次，也就是说，从状态 n 到目标状态的实际耗散 $h^*(n) \geq s(n) = h(n)$。由于在这里的状态 n 是任意的，因此我们就知道了对于任意状态 n 都满足 $h(n) = s(n) \leq h^*(n)$，即这个启发式函数从来不会过高的估计到达目标的耗散值。因此它是可采纳的。

同时我们也可以该启发式函数 $h(n)$ 是一致的。

因为在每个相邻的状态 n 和 n'，假设它们之间经过一次转动解为 $a = (i, j, s)$，则它们之间的状态转化耗散值为 $c(n, a, n') = 3$。而 $h(n)$ 经过一次转动之后，至多减少 3 (将 3 个未解锁的拨轮锁盘解锁)，因此 $h(n') \geq h(n) - 3$，因此移项便可以得到 $h(n) \leq c(n, a, n') + h(n')$， 那么因此就可以得知该启发式函数是一致的。

2. 算法设计

2.1 数据结构

在 A* 算法中，为了能够表示不同状态的信息，因此定义了 `State` 来表示每个状态的信息：

```
class State {
public:
    State(vector<vector<bool>> maze, int g, int cnt, State *parent, int x, int
y, int s) :
    maze(maze), g(g), cnt(cnt), parent(parent), x(x), y(y), s(s) {
        h = cnt;
        f = g + h;
    }

public:
    int x, y, s; // position and solution
    int g, f, h, cnt; // cost
    State *parent; // parent state
    vector<vector<bool>> maze; // maze
};
```

这段代码定义了一个名为 `State` 的类，用于表示迷锁问题中的状态。它包含以下几个成员变量：

- `maze`：表示当前状态对应的迷锁；
- `g`, `h`, `f`：表示当前状态对应的耗散；
- `cnt`：表示当前状态下对应的 1 的个数；
- `parent`：表示当前状态的上一个状态；
- `x`, `y`, `s`：表示当前状态的解过程；

在 A* 算法中，我们可以将所有可能的状态表示为 `State` 对象的集合，并通过比较 `f` 值大小来决定搜索的顺序。在搜索过程中，每个状态记录了它的父状态，因此可以通过回溯父状态的方式还原整个路径。

2.2 扩展状态

在 A* 算法中，我们需要不断地扩展状态，直到找到目标状态或者搜索完所有的状态。因此我们需要通过 `get_successors()` 来获取当前状态的所有后继待扩展状态。

```
inline std::vector<State *> State::get_successors() {
    vector<State *> successors;
    int n = maze.size();
    // for each successor
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            // solution 1
            if (i - 1 >= 0 && j + 1 < n) {
                if (maze[i][j] == 1 || maze[i - 1][j] == 1 || maze[i][j + 1] ==
1) {
                    vector<vector<bool>> maze1 = maze;
                    maze1[i][j] = maze1[i][j] ^ 1;
                    maze1[i - 1][j] = maze1[i - 1][j] ^ 1;
                    maze1[i][j + 1] = maze1[i][j + 1] ^ 1;
                    int curr_cnt = cnt - maze[i][j] - maze[i - 1][j] - maze[i][j
+ 1];
                    curr_cnt = curr_cnt + maze1[i][j] + maze1[i - 1][j] +
maze1[i][j + 1];
```

```
                    State *new_state = new State(maze1, g + 3, curr_cnt, this,
i, j, 1);
                    successors.push_back(new_state);
                }
            }
            // solution 2 ......
            // solution 3 ......
            // solution 4 ......
        }
    }
    return successors;
}
```

函数中使用了两个嵌套的 `for` 循环来遍历当前状态的所有拨轮。对于每个拨轮，函数尝试四种不同的解法，分别对应着将当前格子与其相邻的三个拨轮进行翻转。这四种解法分别对应着四个不同的 `if` 语句块，其中每个 if 语句块都检查了当前拨轮及其相邻的两个拨轮是否都为非锁定状态 0，若是则尝试进行翻转。这样做的原因在于，如果这三个拨轮都是处于非锁定状态 0 的话，那么将其又转动成为锁定状态 1 是完全没有必要的多余操作。然后在转动的过程中，重新计算处于锁定状态 1 的拨轮个数 `curr_cnt`。

对于每个合法的后继待扩展状态，函数创建一个新的 `State` 对象，并将其加入到 `successors` 向量中。`State` 对象的构造函数中需要传递以下参数：

- `maze`：表示转动后新状态对应的迷锁；
- `g + 3`：表示新状态的实际耗散值，即当前状态的实际耗散值加上从当前状态到新状态的耗散值 3 ；
- `curr_cnt`：表示转动后新状态对应的迷锁剩余未被解锁的拨轮个数；
- `i`, `j`, `solution`：表示解法

该函数的作用是获取当前状态的所有合法后继待扩展状态，用于在 A* 算法中进行搜索。在 A* 算法中，我们需要不断扩展当前最优状态的后继带扩展状态，直到找到目标状态或者搜索完所有状态。因此，获取合法后继待扩展状态是 A* 算法的核心操作之一。

2.3 A* 搜索

在 A* 算法中，我们首先定义了一个状态集 `open_list`，用来存储所有的待扩展状态。

```
static bool compare(const State *a, const State *b) {
    return a->f < b->f;
};

multiset<State *, decltype(compare)*> open_list{compare}; // open list
```

函数中使用了 `multiset<State*, decltype(compare)*>` 类型的 `open_list`，用于存储待扩展的状态。`multiset` 容器内部使用红黑树实现，可以自动按照 `f` 值从小到大排序。容器中存储的元素是 `State` 类型的指针，所以需要指定比较函数 `compare`。由于在本次实验中受限于存储空间，因此我们需要使用到存储受限的 A* 算法 (SMA*)。由于 `multiset` 比 `priority_queue` 更方便进行删除操作，所以使用 `multiset` 来存储这些待扩展状态。

在函数开始时，将起始状态 `start` 加入到 `open_list` 中。接着函数中使用了一个 `while` 循环，循环条件是 `open_list` 非空。在循环中，首先取出 `open_list` 中 `f` 值最小的状态，即当前最优状态。然后判断当前状态是否为目标状态，如果是则说明找到了有效解，将解写入到指定的文件中并返回 `true`。

```
open_list.insert(start); // add the start state to the open list
while (!open_list.empty()) {
    // get the state with the smallest f value
    auto current = *open_list.begin();
    open_list.erase(open_list.begin());
    if (current->is_goal()) { // if the current state is the goal state
        // get path ......
        // write path to file ......
        return true;
    }
    // generate successors ......
}
```

如果当前状态不是目标状态，则需要生成当前获取的最优状态的所有合法后继带扩展状态，并将其加入到 `open_list` 中。在加入后继状态之后，为了避免 `open_list` 过于庞大，函数中使用了一个 Simplified Memory Bounded A* 的技巧，在 `open_list` 中的状态数量达到一定阈值时，删除 `f` 值最大的一些状态，使得 `open_list` 的大小保持在一定范围内。即使丢弃了状态，我们仍然可以通过其它路径去到达目标状态。对于 SMA* 算法来说：如果最浅的目标结点的深度 d 小于内存大小，则 SMA* 算法是完备的；如果最优解是可达的，则算法是最优的，否则算法会返回可以到达的最佳解；从实用角度来看，SMA*算法是最好的寻找最优解的通用算法。

```
// generate successors
vector<State *> successors = current->get_successors();
// for each successor
for (auto successor : successors) {
    // if the successor is not in the open list
    open_list.insert(successor);

    // Simplified Memory Bounded A*
    if (open_list.size() > 10000) {
        for (int k = 0; k < 200; k++) {
            auto it = open_list.end();
            it--;
            open_list.erase(it);
            delete *it;
        }
    }
}
```

该函数的核心是 A* 算法的实现。A* 算法是一种启发式搜索算法，它使用了启发式函数来指导搜索方向，可以有效地搜索到最优解。在该函数中，使用了 Simplified Memory Bounded A* 的技巧，可以提高算法的效率和可用性。

3. 探究比较

Dijkstra 算法是一种贪心算法，它使用了一种启发式函数 $h(n) = 0$ 来指导搜索方向，即不考虑目标状态，只考虑从起始状态到当前状态的实际代价，因此它适用于没有明确目标状态的搜索问题。与 Dijkstra 算法相比，A* 算法能够利用启发式函数来减少搜索空间，从而提高搜索效率。在二进制迷锁问题中，使用未解锁的拨轮个数作为启发式函数，可以快速地找到最优解。

Dijkstra 算法是一种无信息搜索算法，它以起点为中心，逐步扩展搜索范围，直到找到目标或者遍历完所有可达状态。因此，当搜索空间较大时，Dijkstra 算法的效率会比较低。与此相比，A* 算法可以利用启发式函数来估计每个状态到目标状态的距离，从而优先搜索更有可能接近目标状态的状态，减少搜索空间。在实际运行中，A* 算法的搜索效率通常比 Dijkstra 算法更高。

对比两个算法在 `input0.txt` 的运行时间和运行结果：在我的测试下，Dijkstra 算法在 `input0.txt` 的运行时间长，且最后搜索出来的步骤需要 87 步，而 A* 算法在 `input0.txt` 的运行时间短，且最后搜索出来的步骤需要 5 步。

将我的部分测试结果记录下来：

输入文件	Dijkstra 算法	A* 算法
`input0.txt`	87	5
`input1.txt`	4	4
`input2.txt`	7	5
`input3.txt`	523	7

可以明显的看到，当搜索空间变大后，Dijkstra 算法不仅搜索的时间变长，而且搜索出来的步骤也十分大。这种无信息的搜索方式，使得它扩展了需要没有必要扩展的状态。实际上可以认为它会把同层的操作全部都扩展一遍 (因为这些状态的实际耗散 `g` 是相同的)，才会接着下一层的操作。

而 A* 算法作为一种有信息的所搜方式，当搜索空间变大时，A* 算法相比于 Dijkstra 算法具有以下优点：

1. 利用启发式函数，减少搜索空间：A* 算法利用启发式函数来估计每个状态到目标状态的距离，从而优先搜索更有可能接近目标状态的状态，减少搜索空间。这种启发式搜索方式能够更快地找到最优解。
2. A* 算法能够避免搜索无关状态：由于启发式函数的引入，A* 算法能够减少搜索与目标状态无关的状态，从而更快地找到最优解。这一点在处理大规模搜索问题时尤为重要。
3. 更好的搜索效率：由于 A* 算法通过启发式函数估计每个状态到目标状态的耗散距离，因此能够在搜索过程中动态地调整搜索方向，从而更快地找到最优解，而 Dijkstra 就无法动态地调整搜索方向。因此在实际应用中，A* 算法通常比 Dijkstra 算法更快。

总之，当需要解决大规模搜索问题时，A* 算法比 Dijkstra 算法更适用。由于 A* 算法利用启发式函数来估计每个状态到目标状态的距离，能够更快地找到最优解，并且减少搜索与目标状态无关的状态，从而减少搜索空间，提高搜索效率。

Lab 1.2

本次实验是一个约束优化问题，涉及到多个变量、多个约束条件和多个取值范围。在该问题中，必须满足每个变量都满足其对应的约束条件，同时使得目标函数取得尽可能优的解。在该实验中，我使用了最少剩余值算法。最少剩余值算法是一种常用的 CSP 求解算法，它的基本思路是优先扩展最难分配的变量，即剩余值最少的变量。该算法通过不断减小剩余值，缩小搜索空间，从而提高搜索效率。

1. 实验描述

在本实验中，主要使用的是最少剩余值算法 (minimum remaining values) 进行启发式搜索。其 CSP 问题的组成如下：

变量集合：所有的轮班班次；

值域集合：所有的宿管阿姨；

约束集合：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；每个阿姨至少需被分配 $\lfloor \frac{D \cdot S}{N} \rfloor$ 次 (感觉并不算严格意义上的约束，因为它不是变量之间的约束关系)；

除了这些约束条件，实际上本次实验还有个最优化问题，我们需要在满足 CSP 问题的前提下，尽可能地最大化所有阿姨的排班请求。即：

$$\max_{\text{Shifts}} \sum_{n\in N,d\in D,s\in S} \text{Requests}_{n,d,s} \times \text{Shifts}_{n,d,s}$$

在最小剩余值算法中，剩余值指的是当前变量的取值范围大小减去已经分配的值的个数。每次选择剩余值最小的变量进行扩展，可以使得剩余变量的剩余值最小化，从而缩小搜索空间，提高搜索效率。需要注意的是，最少剩余值算法并不能保证一定能找到最优解，因为它只是通过优先扩展剩余值最小的变量来缩小搜索空间，但并没有保证局部最优解一定能够扩展到全局最优解。

2. 算法设计

本次实验中使用的算法主要是最少剩余值算法，其主要步骤为：

1. 初始化：将所有变量都标记为未分配，所有变量的剩余值等于其取值范围的大小。
2. 扩展变量：选择剩余值最小的未分配变量进行扩展，即选择剩余值最小的变量作为当前扩展变量。
3. 选择取值：选择当前扩展变量的一个取值，使得该取值能够满足当前变量的约束条件，并且能够尽可能地最大化优化目标。
4. 更新剩余值：对于所有与当前扩展变量有约束条件的未分配变量，更新其剩余值。
5. 重复步骤 2~4，直到所有变量都被分配。
6. 检查解是否满足约束条件：检查所有变量是否满足其对应的约束条件。

同时通过前向检测算法，可以提前检查一些变量是否还有值可以分配，从而达到提前停止失败操作的效果，从而实现剪枝的效果。

上述的前向检测算法检测的是每个变量中的值域是否还有值可以取，除了这个检测方法外，我们还可以实现一种剪枝策略。因为在排班的过程中，我们还要求了每个阿姨至少被分配 $\lfloor \frac{D\cdot S}{N} \rfloor$ 次值班，因此我们还可以通过检测阿姨是否能够满足该要求，从而进行剪枝。

其余详细的优化策略将在 2.2 回溯算法以及优化策略 中详细解释。

2.1 数据结构

在最少剩余算法中，我们使用 `ShiftSchedule` 类来描述宿管阿姨地排班情况：

```
class ShiftSchedule {
public:
    int staffs_num;
    int days_num;
    int shifts_num;
    vector<vector<int>> requests;
    vector<vector<int>> schedule;
    vector<int> assigned_counts;
    bool valid;
    int fulfilled_requests_num;
};
```

这段代码定义了一个 `ShiftSchedule` 类，用于表示宿管阿姨的排班情况。该类包含以下成员变量：

- `staffs_num`，`days_num`，`shifts_num`：阿姨数量，排班天数，轮班次数；
- `requests`：阿姨们的排班请求，表示每个阿姨对每个班次的请求意愿；
- `schedule`：阿姨们的轮班安排，表示每个轮班已经安排的阿姨；
- `assigned_counts`：每个阿姨已经被排班的次数；
- `valid`：表示当前排班方案是否有效；
- `fulfilled_requests_num`：已经满足的轮班请求数量；

2.2 回溯算法以及优化策略

在回溯算法中，首先检查是否所有班次都已经分配：

```
// check if all shifts are assigned
bool all_assigned = true;
for (int d = 0; d < schedule.days_num; d++) {
    for (int s = 0; s < schedule.shifts_num; s++) {
        if (schedule.schedule[d][s] == -1) {
            all_assigned = false;
            break;
        }
    }
    if (!all_assigned) {
        break;
    }
}
```

如果所有的班次都已经被分配，则计算出满足的轮班请求数量，并且将轮班安排返回。

```
// if all shifts are assigned, check if requests are fulfilled
if (all_assigned) {
    schedule.fulfilled_requests_num = schedule.count_fulfilled_requests();
    schedule.valid = true;
    return schedule;
}
```

如果存在任意班次没有被安排，那么采用最少剩余值启发式算法，选择剩余值最少的那个轮班班次，通过公平性原则，将轮班尽可能地分配给有请求的阿姨且当前排班次数最少的阿姨。

具体来说，首先遍历所有的班次，统计每个班次可分配的阿姨数量，然后将班次按照可分配阿姨数量从小到大进行排序，优先选择可分配阿姨数量最少的班次进行排班 (最少剩余值)。同时使用前向检验 (Forward Checking) 的技巧进行剪枝：在统计每个班次可分配的阿姨数量，如果发现有的班次在没有被分配的情况下，已经无法再安排阿姨的时候，即变量没有值可以取了，因此可以提前停止搜索。

```
vector<int> unassigned_staff_ids = schedule.get_unassigned_staff_ids();
priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>>,
               greater<pair<int, pair<int, int>>>> shift_candidates;

for (int d = 0; d < schedule.days_num; d++) {
    for (int s = 0; s < schedule.shifts_num; s++) {
        // use MRV heuristic to select the next shift to assign
        int count = 0;
        for (int staff_id : unassigned_staff_ids) {
            if (schedule.check(staff_id, d, s)) {
                count++;
            }
        }
        if (count > 0) {
            shift_candidates.push(make_pair(count, make_pair(d, s)));
        }
        // find an unassigned shift that is unable to assigned for 0 candidates
        // Forward Checking
        if (count == 0 && schedule.schedule[d][s] == -1) {
            schedule.valid = false;
```

```
                return schedule;
            }
        }
    }
```

通过这个优先队列 `shift_candidates`，我们就可以得到剩余值最少的班次。接着选择剩余值最少的班次进行安排:

```
// assign the selected shift to a staff member
while (!shift_candidates.empty()) {
    auto shift_candidate = shift_candidates.top();
    shift_candidates.pop();
    int day_id = (shift_candidate.second).first;
    int shift_id = (shift_candidate.second).second;
    // assign ......
}
```

在选定班次之后，我们需要安排一个阿姨。首先我们通过 `unassigned_staff_ids` 中获取阿姨。

```
inline std::vector<int> ShiftSchedule::get_unassigned_staff_ids() const {
    vector<int> unassigned_staff_ids;
    for (int i = 0; i < staffs_num; i++) {
        unassigned_staff_ids.push_back(i);
    }
    sort(unassigned_staff_ids.begin(), unassigned_staff_ids.end(),
        [&](int pos1, int pos2) { return (assigned_counts[pos1] <
assigned_counts[pos2]); });
    return unassigned_staff_ids;
}
```

`unassigned_staff_ids` 从 `ShiftSchedule::get_unassigned_staff_ids()` 获得，该函数返回了所有阿姨的编号，并且按其已经被排班的次数进行排序，使得我们在接下来的安排过程中，能尽可能地优先安排轮班次数最少的阿姨，以满足公平性。

当然，除了公平性原则，优先安排轮班次数最少的阿姨也是一种启发式策略。因为我们需要让所有阿姨都满足 $\lfloor \frac{D \cdot S}{N} \rfloor$ 次排班请求，所以如果不采用优先安排轮班次数最少的阿姨的话，首先会将前面所有的阿姨分配完，那么就很容易导致在后续所有的轮班中只有少量阿姨还可以分配，这样就很容易与不能连续工作这一约束冲突。可想而知，这样的分配方式是很容易失败的，那么要想让其通过回溯的方式修改分配方案时，它一般需要回溯到搜索过程刚开始的阶段，那么这样回溯的时间成本太大。因为该 CSP 问题是 NP-hard 的，其复杂度是指数级别的，因此这种方式在大的状态空间下是几乎不能成功的。

因此对于该问题，我们需要优先安排轮班次数最少的阿姨，通过这种启发式策略可以快速的搜索出结果，而且这种方法的时间复杂度和空间复杂度都极低，几步不需要通过回溯便可以搜索得到结果。

在安排的过程中，我们首先需要检查该阿姨是否满足约束条件，判断她是否可以被安排在这个轮班中。同时，为了尽可能的最大化阿姨们的排班请求，我们首先考虑对该轮班有排班请求意向的阿姨进行排班。且被排班的阿姨不应该超过安排次数的上限。

```
bool assigned = false;
double temp = ((double)(schedule.days_num * schedule.shifts_num) /
(double)schedule.staffs_num);
int max_assigned_counts = ceil(temp);
for (int staff_id : unassigned_staff_ids) {
    // check if staff can be assigned
    if (schedule.check(staff_id, day_id, shift_id)
```

```
        && schedule.requests[staff_id][day_id * schedule.shifts_num + shift_id]
== 1
        && schedule.assigned_counts[staff_id] < max_assigned_counts) {
        assigned = true;
        schedule.assign(staff_id, day_id, shift_id);
        auto result = backtrack(schedule);
        // check if result is valid
        // ......
        schedule.unassign(staff_id, day_id, shift_id);
    }
}
```

即在上述的排班过程中，我们优先考虑的是满足下述条件的阿姨进行排班：

- 满足排班约束（必须满足）
- 有排班请求（也必须满足，因为我们的目标是尽可能最大化阿姨们的请求）
- 不超过排班次数上限的阿姨（避免一个阿姨轮班太多）
- 排班数量最少的阿姨（在满足前三个条件的情况下，如果前三个条件不满足，则选择排班数量第二少的阿姨，以此类推）

如果所有阿姨都没有这些排班请求，那么就按下述条件进行排班：

- 满足排班约束（必须满足）
- 排班数量最少的阿姨（在满足第一个条件的情况下）

即在之前并没有分配好阿姨，那么接下来按阿姨已经排班的数量进行排班，使得满足所有阿姨能够达到最少排班次数的要求（那么这时候就不会再考虑阿姨们是否有轮班请求的意愿了）：

```
if (!assigned) {
    for (int staff_id : unassigned_staff_ids) {
        // check if staff can be assigned
        if (schedule.check(staff_id, day_id, shift_id)) {
            schedule.assign(staff_id, day_id, shift_id);
            auto result = backtrack(schedule);
            // check if result is valid
            // ......
            schedule.unassign(staff_id, day_id, shift_id);
        }
    }
}
```

2.3 剪枝策略

因为在上述的分配策略中，我们为了尽可能最大化阿姨们的请求，首先将所有有排班需求的阿姨进行了排班分配。这样就会导致有些阿姨可能因为排班意愿少，从而导致其轮班安排数量不足，无法满足至少需被分配 $\lfloor \frac{D \cdot S}{N} \rfloor$ 次轮班。因此在分配的过程中，我们需要检查是否有阿姨在被分配次数不够的前提下，在后续所有的轮班中，都没有轮班请求。那么这样就很有可能会出现有其它有轮班请求的阿姨将轮班申请了，从而导致该阿姨无法被安排（因为我们首先满足的是有轮班请求意愿的阿姨）。

当然，这样的剪枝策略只是会提高我们的搜索效率，它并不会一定得到最大化的轮班请求次数。 因为就算该阿姨会可能因为其排班意愿太少，在后续的轮班中都没有请求意愿，但是只要其它所有阿姨都没有请求意愿，那么在该阿姨班次次数更少的前提下，她会优先进行第二种分配方式 (即不满足阿姨请求的轮班安排)。那么该搜索路径仍然可能搜索到最优的解。但是被剪枝后就有可能影响最优解的搜索。

```
// pruning the search tree
bool prune(int staff_id, int day_id, int shift_id) const{
```

```
    // if the staff member is already assigned enough shifts, do not prune the search tree
    if (assigned_counts[staff_id] >= (days_num * shifts_num) / staffs_num) {
        return false;
    }
    // only prune the search tree when the staff member does not assign enough shifts
    // and there are no requests for the rest shifts
    for (int d = day_id; d < days_num; d++) {
        for (int s = shift_id; s < shifts_num; s++) {
            // if the staff member is able to assigned in the following shifts, do not prune the search tree
            if (requests[staff_id][d * shifts_num + s] == 1) {
                return false;
            }
        }
    }
    // if any staff member is unable to assigned in the following shifts, prune the search tree
    return true;
}
```

这段代码实现了在回溯算法中的剪枝操作，用于减少搜索空间，提高算法效率。在每次分配班次之前，调用该函数，判断当前员工是否已经被分配到足够的班次，以及是否存在后续班次无法满足员工需求的情况，如果满足以上两个条件，则可以进行剪枝操作，跳过该员工进行下一步搜索。

```
for (int staff_id : unassigned_staff_ids) {
    // prune the search tree
    if (schedule.prune(staff_id, day_id, shift_id)) {
        break;
    }
    // check if staff can be assigned
    if (schedule.check(staff_id, day_id, shift_id)
        && schedule.requests[staff_id][day_id * schedule.shifts_num + shift_id] == 1
        && schedule.assigned_counts[staff_id] < max_assigned_counts) {
        // ...... has shown in the previous codes
    }
}
```

该段代码实现了回溯算法中的剪枝操作，通过判断员工是否已经被分配到足够的班次以及后续班次是否能够满足员工需求，有效地减少搜索空间，提高算法效率。

3. `input0.txt` 安排方式

在上述的介绍中，我们优先考虑的是：

- 满足排班约束（必须满足）
- 有排班请求（也必须满足，因为我们的目标是尽可能最大化阿姨们的请求）
- 不超过排班次数上限的阿姨（避免一个阿姨轮班太多）
- 排班数量最少的阿姨（在满足前三个条件的情况下，如果前三个条件不满足，则选择排班数量第二少的阿姨，以此类推）

则安排过程如下：

1. 则在刚开始的时候，所有的轮班都不会受约束影响，因此所有的轮班都有 3 个阿姨可以排班，则先对第 $(0,0)$ 轮班进行分配。此时所有阿姨的排班数量都是 0，因此得到的阿姨顺序为 $(1,2,3)$。按顺序来，阿姨 1 是有排班请求的，因此第 $(0,0)$ 轮班安排阿姨 1。

2. 在 $(0,0)$ 轮班安排完之后，重新计算所有轮班的剩余值。由于约束关系的影响，轮班 $(0,1)$ 只有剩余值阿姨 2 和阿姨 3，因为阿姨 1 不能连续工作两次。所以轮班 $(0,1)$ 的剩余值最小，而其余轮班都不会受约束值的影响，因此接下来选择轮班 $(0,1)$ 进行安排。此时阿姨 1 的排班数量是 1，而阿姨 2 和阿姨 3 的排班数量都是 0，因此返回得到的阿姨顺序是 $(2,3,1)$。按顺序来，阿姨 2 是有排班请求的，因此第 $(0,1)$ 轮班安排阿姨 2。
3. 在 $(0,1)$ 轮班安排完之后，重新计算所有轮班的剩余值。由于约束关系的影响，轮班 $(0,2)$ 只有剩余值阿姨 1 和阿姨 3，因为阿姨 2 不能连续工作两次。所以轮班 $(0,2)$ 的剩余值最小，而其余轮班都不会受约束值的影响，因此接下来选择轮班 $(0,2)$ 进行安排。此时阿姨 1 和阿姨 2 的排班数量都是 1，而阿姨 3 的排班数量是 0，因此返回得到的阿姨顺序是 $(3,1,2)$。按顺序来，阿姨 3 是有排班请求的，因此第 $(0,2)$ 轮班安排阿姨 3。

4. 在 $(0,2)$ 轮班安排完之后，重新计算所有轮班的剩余值。由于约束关系的影响，轮班 $(1,0)$ 只有剩余值阿姨 1 和阿姨 2，因为阿姨 3 不能连续工作两次。所以轮班 $(1,0)$ 的剩余值最小，而其余轮班都不会受约束值的影响，因此接下来选择轮班 $(1,0)$ 进行安排。此时所有阿姨的排班数量都是 1，因此得到的阿姨顺序为 $(1,2,3)$。按顺序来，阿姨 1 是有排班请求的，因此第 $(1,0)$ 轮班安排阿姨 1。
5. 在 $(1,0)$ 轮班安排完之后，重新计算所有轮班的剩余值。由于约束关系的影响，轮班 $(1,1)$ 只有剩余值阿姨 2 和阿姨 3，因为阿姨 1 不能连续工作两次。所以轮班 $(1,1)$ 的剩余值最小，而其余轮班都不会受约束值的影响，因此接下来选择轮班 $(1,1)$ 进行安排。此时阿姨 1 的排班数量是 2，而阿姨 2 和阿姨 3 的排班数量都是 1，因此返回得到的阿姨顺序是 $(2,3,1)$。由于阿姨 2 在轮班 $(1,1)$ 中没有请求意愿，因此跳过阿姨 2。接着选下一个排班次数最少的阿姨 3，而此时阿姨 3 有排班请求意愿，因此在这个时候，第 $(1,1)$ 轮班就安排阿姨 3。

重复上述操作，直到 $(4,0)$ 轮班会发现所有的阿姨都没有排班请求意愿，因此我们将采取第二种排班方式 (上述介绍中):

- 满足排班约束（必须满足）
- 排班数量最少的阿姨（在满足第一个条件的情况下）

此时所有的阿姨排班数量都是 4，因此得到的阿姨顺序为 $(1,2,3)$。按顺序来，因此第 $(4,0)$ 轮班安排阿姨 1。

接着在 $(4,2)$ 轮班的时候会发现，此时由于约束关系的影响，轮班 $(4,2)$ 只有剩余值阿姨 1 和阿姨 3，因为轮班 $(4,1)$ 安排的是阿姨 2。此时阿姨 1 和阿姨 2 的排班数量都是 5，而阿姨 3 的排班数量是 4，因此返回得到的阿姨顺序是 $(3,1,2)$。按顺序来，首先考虑阿姨 3，但是发现阿姨 3 没有排班请求，所以它反而会选择排班数量更多的阿姨 1 进行排班 (因为阿姨 1 此时有排班请求)，所以第 $(4,2)$ 轮班会安排阿姨 1。在这轮安排结束后，阿姨 1 的排班数量是 6，阿姨 2 的排班数量是 5，而阿姨 3 的排班数量是 4。

而后续也是没有什么特殊情况了，其安排方式都是按照上述正常流程进行的，所以在整个算法结束后，`output0.txt` 的安排方式如下:

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