# Lab1

李远航 PB20000137

## 实验1.1

### 一、实验内容

使用 `A*` 算法求解$N \times N$的轮盘问题

### 二、实验设计

#### 1. 启发函数设计

- 假设 1 是一块一块的消除，而消除每一个 1 所需要的最小代价是 $\frac{1}{3}$，所以消除一个连通块内 1 所需要的代价是一定不会低于 $\lceil \frac{n}{3} \rceil$
- 每一次操作，都会使总的剩余 1 的个数变化奇数个，也就是说，一定会改变剩余 1 的奇偶性，所以奇数个 1 需要奇数步消除，偶数个 1 需要偶数步消除。比如如果剩余 1 的个数为偶数，而计算出来的估值函数为奇数，估值函数可以自增 1

根据上面的描述，可以判断该启发函数是 admissible 的，下面论述 consistent 性质：

- 假设当前 `N` 时的状态和 `N'` 的状态如下

```
0 0 0 1        0 0 0 1
0 1 0 0        0 0 1 0
0 0 0 1   ==>  0 1 0 1
1 0 1 0        1 0 1 0
   N              N'
```

- 此时$h(N)=5$，而$h(N')=2$，$c(N,N')=1$，此时显然有$c(N,N')+h(N') \leq h(N)$，即该启发函数是不满足 consistent 性质的

#### 2. 算法主要思路

- 定义起点 $s$，终点 $t$，从起点（初始状态）开始的距离函数 $g(x)$，到终点（最终状态）的距离函数 $h(x)$，$h^*(x)$，以及每个点的估价函数 $f(x)=g(x)+h(x)$。
- A * 算法每次从优先队列中取出一个 $f$ 最小的元素，然后更新相邻的状态。
- 如果 $h \leq h*$，则 `A*` 算法能找到最优解。
- 上述条件下，如果 $h$ 满足三角形不等式，则 `A*` 算法不会将重复结点加入队列。
- 设计一个优先队列，按照$f(x)$的大小存储，每次弹出$f(x)$最小的节点，并扩展节点，再次入队，直到搜到目标情况结束搜索，由于本题不可能出现无解的情况，所以未考虑无解情形
- 为了搜索的效率，由于操作执行的顺序不会影响最后的答案，所以每一层节点只搜索和轮盘中第一个 1 有关的 6 个 L 的操作，减少程序运行的时间和内存的消耗，解释如下：

```
# 对于第 k 行出现的第一个 1，那么 k-1 行应该均为 0 ，此时以下几种操作互相抵消：
110  011  010  010  100  001
010  010  110  011  110  011
000  000  000  000  000  000
# 只需要考虑以下 6 种 L
000  000  000  000  000  000
010  010  110  011  110  011
110  011  010  010  100  001
```

### 3. 数据结构设计

- 存储节点 (相同估计代价时，优先访问深度更深的节点)

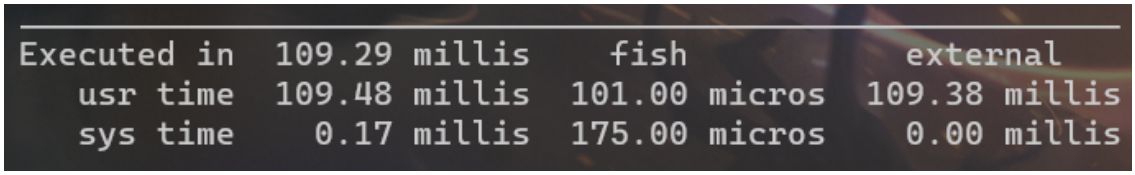
```
struct node
{
    int i, j, s;
    int g;
    int hv;
    std::queue<std::tuple<int, int, int>> vis;
    // 用三元组存储访问的路径
    friend bool operator<(struct node n1, struct node n2)
    {
        if (n1.g + n1.hv == n2.g + n2.hv)
            return n1.g < n2.g;
        return n1.g + n1.hv > n2.g + n2.hv;
    }
};
```

- 状态的存储结构设计

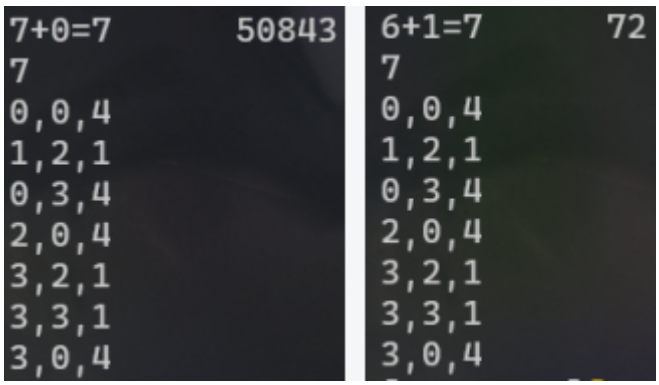
```
std::vector<int> data(n, 0);
//状态压缩，每一个位数表示
```

## 三、实验结果

- 编写脚本，测试输入样例，可以看到程序可以在1s之内完成对所有测试点的搜索




- 使用 `dijkstra` 算法，即将启发函数设为0，运行上述的测试点
  - 经过测试发现，程序可以较为快速的搜索出 `input0` ~ `input5` 的结果，而对于后续的测试点， `dijkstra` 算法在内存耗尽之前，均无法搜出结果。
  - 同时我们可以输出搜索相同的测试点，两种不同算法需要访问的测试点数目，以 `input5` 为例， `dijkstra` 算法搜索出最优解需要访问 50843 个节点，而使用 `A*` 搜索只需要访问 72 个节点。




- 从上述的测试结果中可以看到，使用 `A*` 搜索和 `dijkstra` 算法均可以获得正确的结果，但 `A*` 搜索减少了对内存的消耗，以及搜索所需涉及的节点数，提升了搜索算法的运行效率，节省了时间和空间

## 实验1.2

### 一、实验内容

开发一个csp算法，完成对学校宿管阿姨的排班，尽可能满足阿姨的需求

### 二、实验设计

#### 1. 数据结构设计

- 由于排班的天数和轮次没有本质的区别，所以这里选择直接将总天数和总轮次乘起来，用一个二维数组记录阿姨对排班的请求

```
std::vector<std::vector<int>> order(D * S, std::vector<int>(N));
```

- 需要一个数据结构用于存储不同排班时间能够被排班的人等等信息，设计如下结构体

```
struct Node
{
    int loc;
    int Remain;
    std::priority_queue<struct SaNode> ReLoc;
    friend bool operator<(struct Node n1, struct Node n2)
    {
        if (n1.Remain == n2.Remain)
            return n1.ReLoc.size() > n2.ReLoc.size();
        else
            return n1.Remain > n2.Remain;
    }
};

```

- 同一个排班时间能被安排的人由于自身的性质不同，同样设计一个结构体来存储

```cpp
struct SaNode
{
    int people;
    int isTrue;
    int Already;
    friend bool operator<(struct SaNode n1, struct SaNode n2)
    {
        if (n1.isTrue == n2.isTrue)
            return n1.Already > n2.Already;
        else
            return n1.isTrue < n2.isTrue;
    }
};
```

**2. 搜索策略设计**

- 为了减小搜索空间，所以优先搜索能被安排合理请求最少的排班时间
- 由于题目要求了每位阿姨的排班应当尽量平均，所以在对一个时间安排阿姨时，优先选择满足请求当前排班最少的阿姨，最后选择不满足请求的阿姨。
- 为了避免部分不必要的搜索，记录当前所能满足的最多请求，如果当前安排不可能让请求数多于此时的最多请求，则直接搜索下一个可能的值

**3. 具体实现**

- 每一次进入搜索时，先统计尚未排班的时间可以被安排的阿姨进入优先队列，而这些阿姨同样使用优先队列按照已经排班从小到大排序
- 从队列中弹出可以安排人数最少的时间，然后再弹出这一时间，当前排班最少的阿姨，递归进入下一次搜索
- 记录当前所能满足的最大请求，一但本次搜索会让最后的结果不可能超过已经找到最大请求，返回。

**4. 约束满足问题的基本要素**

- 变量集合：即为D*S个时间的排班，对应了`int ans[D*S]`的数据结构
- 值域集合：每次搜索会更新所有变量可能的取值，即：`std::priority_queue<struct Node> q`
- 约束集合：
    - 同一个阿姨不能连续排班，体现在分支语句：

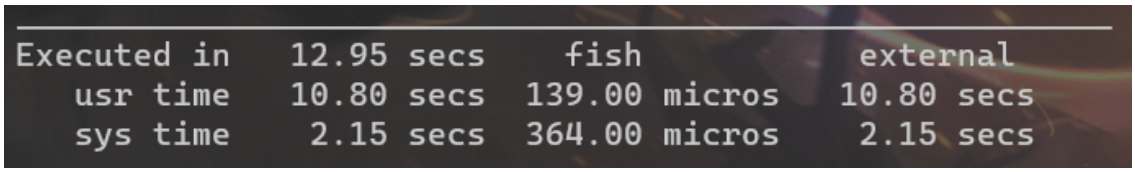
    ```cpp
    if ((i - 1 < 0 || ans[i - 1] != j + 1) && (i + 1 >= D * S || ans[i + 1] != j + 1)){}
    ```

    - 每一个阿姨的排班数不能少于$\lfloor \frac{D \cdot S}{N} \rfloor$，体现在：

    ```cpp
    for (int i = 0; i < N; i++)
    {
        if (HaveOrder[i] < D * S / N)
            return 0;
    }
    ```

## 三、实验结果

- 编写脚本，测试输入样例，可以看到程序可以在二十秒之内完成对所有测试点的搜索。观察输出结果，只有`input0`不能满足所有的请求，观察测试点发现，有一个排班没有任何阿姨请求




- 实验中首先使用了MRV策略，优先考虑最少剩余值的排班时间，该策略大大提升了搜索的速度，不使用该策略，基本不可能在有限时间输出结果
- 考虑到实验要求每个阿姨排班尽量平均，所以优先考虑当前排班最少的阿姨，代码只能较快的跑出前几个测试点，而对于后续的测试点，程序无法在有限的时间内得到一个合理的解
- 为了避免不必要的搜索，代码中使用了前向检验策略，一但发现本次取值不会让当前的最大请求数提高，直接搜索下一个取值
- `input0`的排班:

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