# 人工智能EXP1实验报告

**PB19071535**徐昊天

## 实验1.1

### 一.设计可采纳启发式

将曼哈顿距离变种：

对5*5的方格进行一次二维循环遍历，若start和target在同一坐标下存在错位星球，则取以下五个距离：

1. 同一星球在start和target中的曼哈顿距离。
2. 同一星球从start到达方格A的曼哈顿距离，加上A通过星际通道到达B的距离1，再加上方格B到达target中的曼哈顿距离。
3. 同一星球从start到达方格B的曼哈顿距离，加上B通过星际通道到达A的距离1，再加上方格A到达target中的曼哈顿距离。
4. 同一星球从start到达方格C的曼哈顿距离，加上C通过星际通道到达D的距离1，再加上方格D到达target中的曼哈顿距离。
5. 同一星球从start到达方格D的曼哈顿距离，加上D通过星际通道到达C的距离1，再加上方格C到达target中的曼哈顿距离。

取以上五个距离的最小值作为min_distance，将所有错位星球的min_distance相加，即可作为启发函数h2(n)。

证明**h2**可采纳性：

考虑每一个错位星球，由于从初始状态到达目标状态最多有五种路径：

1. 不穿过星际通道。
2. 从A通过星际通道到达B。
3. 从B通过星际通道到达A。
4. 从C通过星际通道到达D。
5. 从D通过星际通道到达C。

而以上考虑的五种距离分别是以上五种情况的最短距离，故对以上五种距离取的最小值一定不会高于该错位星球到达目标的代价。

根据以上结论，考虑所有星球，所有错位星球的min_distance之和一定不会高于5*5方格从初始状态start到达目标状态target的代价。

故可证该启发式是可采纳的。

### 二.算法思想

1. A* 是一种知情搜索算法，它对结点的评估结合了g(n)，即到达此结点已经花费的代价，和h(n)，从该结点到目标结点所花代价：f(n)=g(n)+h(n)。

   由于g(n)是从开始结点到结点n的路径代价，而h(n)是从结点n到目标结点的最小代价路径得估计值，因此f(n)=经过结点n的最小代价解的估计代价。

   这样，如果我们想要找到最小代价的解，首先扩展g(n)+h(n)值最小的结点是合理的。

2. IDA*与A*不同，不使用动态规划，因此通常最终会多次探索相同的节点。IDA*在每次迭代时，执行深度优先搜索，当其总成本达到给定的阈值时切断一个分支。该阈值从初始状态的成本估计开始，并随着算法的每次迭代而增加。在每次迭代中，用于下一次迭代的阈值是超过当前阈值的所有值的最小成本。
3. 本次实验针对星球分布问题，令g(n)为飞船已经走过的步数，h(n)为以上两种启发式函数，通过对结点进行“上下左右”四种方向的拓展，寻找到达目标星球分布的最短距离方案。

## 三.样例结果

对提供的测试用例进行分析，每个样例的运行结果如下表所示(用time.h计时,部分测试样例较快，输出时间为0)：

**A_h1:**

| 样例编号 | 运行时间(s) | 移动序列 | 总步数 |
| --- | --- | --- | --- |
| 00 | 0 | DDRUR | 5 |
| 01 | 0 | ULLUULDD | 8 |
| 02 | 0 | DDLUULLURR | 10 |
| 03 | 0.001 | DLDRRURRRUUURR | 14 |
| 04 | 0 | LUUURULLURDDRDR | 15 |
| 05 | 0.003 | LLUURRRUURDDDDLUURDD | 20 |
| 06 | 0.003 | DRDLLULULUUURDRURDRDRRR | 23 |
| 07 | 0 | URRRRDLLLLDRRRRDLLLLDRRRR | 25 |
| 08 | 0.011 | DRDLULLLDRUUUULDRRRRULDDDRD | 27 |
| 09 | 0.181 | RDRDLUUUURRUUURDRUUULDLDDDRR | 28 |
| 10 | 0.012 | DDRRUUUULLULLUULLLLLUURRDDDDRR | 30 |
| 11 | 0.87 | DRUURDRRDRUULDLULDLDRDLDRURDRURD | 32 |

**A_h2:**

| 样例编号 | 运行时间(s) | 移动序列 | 总步数 |
| --- | --- | --- | --- |
| 00 | 0 | DDRUR | 5 |
| 01 | 0 | ULLUULDD | 8 |
| 02 | 0 | DDLUULLURR | 10 |
| 03 | 0.001 | DLDRRURRRUUURR | 14 |
| 04 | 0.002 | LUUURULLURDDRDR | 15 |
| 05 | 0 | LLUURRRUURDDDDLUURDD | 20 |
| 06 | 0 | DRDLLULULUUURDRURDRDRRR | 23 |
| 07 | 0.001 | URRRRDLLLLDRRRRDLLLLDRRRR | 25 |
| 08 | 0.001 | DRDLULLLDRUUUULDRRRRULDDDRD | 27 |
| 09 | 0.008 | RDRDLUUUURRUUURDRUUULDLDDDRR | 28 |
| 10 | 0 | DDRRUUUULLULLUULLLLLUURRDDDDRR | 30 |
| 11 | 0.009 | DRUURDRRDRUULDLULDLDRDLDRURDRURD | 32 |

**IDA_h1:**

| 样例编号 | 运行时间(s) | 移动序列 | 总步数 |
| --- | --- | --- | --- |
| 00 | 0 | DDRUR | 5 |
| 01 | 0 | ULLUULDD | 8 |
| 02 | 0.002 | DDLUULLURR | 10 |
| 03 | 0 | DLDRRURRRUUURR | 14 |
| 04 | 0.003 | LUUURULLURDDRDR | 15 |
| 05 | 0.004 | LLUURRRUURDDDDLUURDD | 20 |
| 06 | 0.007 | DRDLLULULUUURDRURDRDRRR | 23 |
| 07 | 0.002 | URRRRDLLLLDRRRRDLLLLDRRRR | 25 |
| 08 | 0.155 | DRDLULLLDRUUUULDRRRRULDDDRD | 27 |
| 09 | 1.524 | RRRRDRUUULDLDLLDRDLUUUURRURR | 28 |
| 10 | 0.276 | DDRRUUUULLULLUULLLLLUURRDDDDRR | 30 |
| 11 | 17.616 | DRUURDRRDRUULDLULDLDRDLDRURDRURD | 32 |

**IDA_h2:**

| 样例编号 | 运行时间(s) | 移动序列 | 总步数 |
| --- | --- | --- | --- |
| 00 | 0.001 | DDRUR | 5 |
| 01 | 0 | ULLUULDD | 8 |
| 02 | 0.002 | DDLUULLURR | 10 |
| 03 | 0 | DLDRRURRRUUURR | 14 |
| 04 | 0 | LUUURULLURDDRDR | 15 |
| 05 | 0 | LLUURRRUURDDDDLUURDD | 20 |
| 06 | 0.001 | DRDLLULULUUURDRURDRDRRR | 23 |
| 07 | 0 | URRRRDLLLLDRRRRDLLLLDRRRR | 25 |
| 08 | 0 | DRDLULLLDRUUUULDRRRRULDDDRD | 27 |
| 09 | 0.041 | RRRRDRUUULDLDLLDRDLUUUURRURR | 28 |
| 10 | 0 | DDRRUUUULLULLUULLLLLUURRDDDDRR | 30 |
| 11 | 0.047 | DRUURDRRDRUULDLULDLDRDLDRURDRURD | 32 |

## 四.优化方法

1. A*算法利用优先队列存储open，即尚未遍历的结点，并通过重载运算符使f=g+h较小的结点拥有较高的优先级，从而能够尽量优先遍历估计代价较小的结点，减少运行时间。利用set存储close，即已经遍历过的方阵，当从优先队列中弹出的结点中的方阵存在于close集合中，则表示此方阵已经遍历过，无需再考虑该结点，故应当跳过，选取下一个结点；若该方阵不存在与close集合中，则将其插入close集合中并遍历邻居结点。close集合用于去重，防止多次遍历相同的方阵，可减少运行时间。
2. IDA*算法利用栈存储open，即尚未遍历的结点，以此实现深度优先搜索的目的。利用set分别存储open中的方阵、已经遍历过的方阵close，当从栈中弹出的结点中的方阵存在于close集合中，则表示此方阵已经遍历过，无需再考虑该结点，故应当跳过，选取下一个结点；若该方阵不存在与close集合中，则将其插入close集合中并遍历邻居结点，对于邻居结点需判断其方阵是否存在于open_set和close集合中，若存在则不插入open栈中。open_set和close集合用于去重，防止多次遍历相同的方阵，可减少运行时间。

# 实验1.2：作业调度问题

## 一.定义

1. 车间1

   变量集合X={周一_工人1,周一_工人2,...周一_工人7,周二_工人1,...周日_工人7}

   一共7*7=49个变量，分别是每一天对应每一位工人。

   值域集合D={0,1}

   0表示工人不在当天工作，1表示工人在当天工作。

   约束集合C={

   对任意一位工人i，周一_工人i...周日_工人i为0的数应大于等于2;

   对任意一位工人i，周一_工人i...周日_工人i不可有连续三个为0;

对任意一天i，周i_工人1...周i_工人7为1的不可小于4;

对任意一天i，周i_工人1...周i_工人7至少有一位级别为senior;

对任意一天i，周i_工人1,周i_工人4不可同时为1;

对任意一天i，周i_工人2,周i_工人3不可同时为1;

对任意一天i，周i_工人3,周i_工人6不可同时为1

}

2. 车间2

变量集合X={周一_工人1,周一_工人2,...周一_工人10,周二_工人1,...周日_工人10}

一共7*10=70个变量，分别是每一天对应每一位工人。

值域集合D={0,1}

0表示工人不在当天工作，1表示工人在当天工作。

约束集合C={

对任意一位工人i，周一_工人i...周日_工人i为0的数应大于等于2;

对任意一位工人i，周一_工人i...周日_工人i不可有连续三个为0;

对任意一天i，周i_工人1...周i_工人10为1的不可小于5;

对任意一天i，周i_工人1...周i_工人10至少有一位级别为senior;

对任意一天i，周i_工人1,周i_工人5不可同时为1;

对任意一天i，周i_工人2,周i_工人6不可同时为1;

对任意一天i，周i_工人8,周i_工人10不可同时为1

}

## 二.算法思想

CSP的主要思想是通过识别违反约束的变量/值的组合迅速消除大规模的搜索空间。利用回溯法来解决，它保持部分变数的赋值。一开始，所有的变数都还没被赋值。在每一个步骤中，先选取一个变数，并且将所有可能的值依次赋予该变数。对于每一个值，在限制条件下的局部赋值的无矛盾性都进行检查。在符合无矛盾的情况下，就会递回地往下呼叫。当所有的值都试过，演算法则回溯上层。

本次实验针对车间调度问题，采用从周一到周日的顺序逐一将工人插入对应日期的方式，并在插入的同时判断是否存在违反约束的情况，若违反约束则进行回溯并重新插入新的工人，如此递归执行，直至找到满足所有约束条件的调度方案。

## 三.优化方案

使用MRV启发式作为优化方案，以车间1为例，对应优化函数如下：

```
int mrv_1(int n)
{
    int next = 8;
    int min = 11;
    for (int i = 0; i < 7; i++)      // 找到满足约束的工人数最少的日期
    {
        int num = 0;
        for (int j = day[i][length_day[i] - 1]; j < n; j++)
        {
            if (check_1(i, j))
                num++;
        }
```

```
        if (num < min && num != 0)
        {
            min = num;
            next = i;
        }
    }
    return next;
}
```

通过对一周7天进行遍历，并遍历每一天还可能可以加入的工人，判断加入的工人是否满足约束条件。最后选取出满足约束条件的可以加入的工人存在但最少的日期，作为下一次搜索的日期，即实现了MRV启发式。

此方法选择最可能很快导致失败的变量，从而对搜索树剪枝，避免无意义的搜索继续进行，有助于最小化搜索树中的结点数。

## 四.思考题

以上车间调度问题可以通过模拟退火算法进行解决，对应伪代码如下：

```
function BACKTRACK(assignment)
    if assignment is complete then return assignment
    assignment_sa = assignment
    for each var in var_set
        add var to assignment
        next = assignment
        if check(next) > check(assignment_sa) || check(next) >
check(assignment_sa) * random(0,1)
            result<-BACKTRACK(next)
            if result != failure
                return result
     return failure
```

以上的check()函数用于计算满足的约束条件个数。