A*

JL22110003 邱文韬

启发式函数

三种启发式函数：

1. DisjointL：不相交的"L"个数
2. NumOf_1: 1的个数
3. Manhattan：所有1之间的曼哈顿距离两两求和总和

admissible

1. 其中DisjointL启发式是可采纳的，因为在一个矩阵中，将变换一个“L”代价视作1，而统计不相交的L个数，相当于原问题的松弛化，即将锁盘内所有"L"都视作不相交，即最简单的情况逐个把“L”由全1变成全0，但实际上可能还存在分散的没有构成“L”的1，所以实际代价一定比这种方式要更多，所以DisjointL是可采纳的。
2. 如果将变换一个"L"的代价视作3，则NumOf_1启发式也是可采纳的，与DijointL类似，考虑问题的松弛化，即锁盘内如果恰所有“L”都不相交，那么逐个转换1的代价最多和实际代价相同，如果1是分散的，不存在任何一个“L”，那么统计的1个数会远小于实际代价，而不会超过实际代价。
3. 第三个启发式函数Manhattan不满足可采纳性，采用曼哈顿距离，很容易举出一个例子，比如如果只存在一个“L”，那么无论实际代价视作1或者3，得到的曼哈顿距离都会超过这个实际代价，所以估价可能会远大于实际代价，但此时运行速度很快，但一般不是最优解。

consistent

1. 假设存在一个状态n和它的后继状态n'，使得DisjointL(n) > c(n,n') + DisjointL(n')， 其中c(n, n')是从状态n到状态n'的实际代价。状态n到n'的实际代价为1，即转动一个“L”，并且转动后锁盘n'满足：DisjointL(n) -1≤DisjointL(n')≤DisjointL(n) +1，即DisjointL(n')最小为DisjointL(n) -1，带入上式，DisjointL(n) > 1 + DisjointL(n)-1 = DisjointL(n) > DisjointL(n)，DisjointL(n) > DisjointL(n)不成立，与原假设矛盾，所以不存在一种状态使得假设DisjointL(n) > c(n,n') + DisjointL(n')成立，即该启发式函数满足一致性。
2. 第二个启发式函数与第一个类似，假设存在一个状态n和它的后继状态n'，使得NumOf_1(n) > c(n,n') + NumOf_1(n')， ， 即 NumOf_1(n) - 3≤NumOf_1(n')≤NumOf_1(n) + 3， 带入假设中， NumOf_1(n) > 1 + NumOf_1(n)-1 = NumOf_1(n) > NumOf_1(n)，NumOf_1(n) > NumOf_1(n)不成立，与原假设矛盾，所以满足一致性。
3. 第三个启发式不满足可采纳性，所以也不满足一致性。

算法的主要思路

- **数据结构**：

1. **锁盘数据结构体**，N：锁盘大小、g：从父结点到当前节点的代价、h：从当前结点到目标状态结点的估计代价即启发函数值、parent：父结点、pre：从父结点到当前结点采取的动作，一个三元组表示选择的坐标和对应的变换类型。存在有参构造函数和无参构造函数，用于初始化结点或者在后续拓展结点时选择继承属性的拓展方式。

```
struct LockMatrix {
    int N;
    TwoDMatrix matrix;
    int g;
    int h;
    LockMatrix* parent;
    pair<pair<int, int>, int> pre;

    LockMatrix() {
        N = 0;
        matrix =     TwoDMatrix();
        g = 0;
        h = 0;
        pre = make_pair(make_pair(-1, -1), -1);
        parent = NULL;
    }

    LockMatrix(const int& g, const int& h, const  TwoDMatrix& matrix) {
        this->N = matrix.size();
        this->matrix = matrix;
        this->g = g;
        this->h = h;
        this->pre = make_pair(make_pair(-1, -1), -1);
    }
};
```

2. **封闭列表**：VisitedList，存储每个已拓展过的结点对应的锁盘状态，避免重复计算
3. **优先队列**：priority_queue<LockMatrix,*vector<LockMatrix>*, cmp>，边缘队列用一个优先队列实现，存放待拓展的结点，自定义比较方法，根据f = g+h值从小到大入队，每次拓展结点都是选择估计代价最小的结点，优先队列排序时间复杂度为$O(\lg n)$，如果通过自己手动时间排序算法，可能取得冒泡排序$O(n^2)$或者快速排序$O(n \lg n)$，但即使是线性时间排序也比优先队列更慢，所以选择优先队列加快这个过程，也是我在A*算法实现中采取的一个优化手段之一。

- **算法**：

1. **Astar算法**：朴素的Astar算法，根据设计的启发式函数选择结点拓展，但性能受启发式函数影响很大，并且很难得到一个接近实际代价的估价函数。
    - 算法流程：
        1. 传入起始结点对应的锁盘状态
        2. 将锁盘状态加入边缘队列，再加入封闭列表
        3. 判断边缘队列是否为空，不为空则队头结点出队
        4. 判断状态是否是目标状态，是则根据父结点沿着链表记录解路径，算法结束，否则继续下一步
        5. 通过`ExtendFrontier`函数将当前状态的子结点加入边缘队列 (每个结点有4种“L”的变换方式，也就是每个状态有四个子结点)
        6. 当前状态加入封闭列表
        7. 回到步骤3

2. **IDAstar算法**：迭代Astar算法：与A*算法类似，但加入一个limit限制，每次选取一个limit值，对于超过这个阈值的结点，不加入边缘队列，这样队列中的结点数大大减少，如果在一个较小的深度就找到了一个解则直接返回，如果没找到，将这个阈值设置为上一次f = g + h中超过阈值的最小值，再重新从初始状态开始向下拓展结点，每一轮迭代的深度都会加深。
   - 算法流程：
     1. 传入起始结点对应的锁盘状态
     2. 判断当前的limit值是否小于`INT_MAX`（即不溢出int），并且`flag!=true`(即未到达目标状态)，满足则继续步骤3，否则算法结束
     3. 初始化边缘队列，初始化封闭列表，将初始状态入队，再将该状态加入封闭列表
     4. 判断边缘队列是否为空，不为空则队头结点出队
     5. 判断状态是否是目标状态，是则flag = true，否则回到步骤2
     6. 剩余步骤与A*类似，拓展边缘队列的函数使用`ExtendFrontierWithLimit`，在这个函数中修改limit的值，因为没有找到目标结点，所以需要更新limit

启发式函数为 0，此时 A* 退化为 Dijkstra 算法

当启发式函数为0时，每个状态转移的估价f(n) = g(n)，而每一步操作的代价都是相同的，即转变一个“L”，可以设置为1或者3，此时相当于在一个权值均相同的起始状态和目标状态之间找一个最短路径，此时运行的非常非常慢，基本只能够运行出input1的结果。

情况	函数	结果
$\hat{h}(n)=0$，即 $\hat{f}(n)=\hat{g}(n)$	A*算法退化为Dijkstra算法	保证能找到最短路径
$\hat{h}(n) \leq$ 实际代价	$\hat{h}(n)$越小，A*扩展的节点越多，运行的越慢	保证能找到一条最短路径，但运算更快了
$\hat{h}(n)=$ 实际代价	仅寻找最佳路径，而不扩展任何别的节点	保证能找到一条最短路径，并且运算非常快
$\hat{h}(n)>$ 实际代价	寻找最佳路径且扩展别的任何节点	不能保证找到一条最短路径，但运算更快了
$\hat{h}(n)>>\hat{g}(n)$	A*算法退化为BFS算法	不能保证找到一条最短路径，但运算非常快

比较并分析使用 A* 方法带来的优化效果

Dijkstra

输入	运行时间	解步骤数
input0	超时	无
input1	7.414s	4
input2	超时	无
input3	超时	无
input4	超时	无
input5	超时	无
input6	超时	无
input7	超时	无
input8	超时	无
input9	超时	无

三种启发式函数分别命名为：DisjointL、NumOf_1、Manhattan，其中DisjointL最慢，但能保证找到最短路径，而NumOf_1在某些输入下可以找到最多路径，并且运算相对较快，而Manhattan不能保证找到最短路径，但运行非常快，其中IDA*的效率和A *对比有一定提升，但因为输入数据的原因，在后续输入规模较大时仍然运行时间很长也无法给出结果。

A*+DisjointL

输入	运行时间	解步骤数
input0	超时	无
input1	0.573s	4
input2	832.273s	5
input3	超时	无
input4	超时	无
input5	超时	无
input6	超时	无

输入	运行时间	解步骤数
input7	超时	无
input8	超时	无
input9	超时	无

IDA*+DisjointL

输入	运行时间	解步骤数
input0	超时	无
input1	0.424s	4
input2	32.273s	5
input3	超时	无
input4	超时	无
input5	超时	无
input6	超时	无
input7	超时	无
input8	超时	无
input9	超时	无

A* +NumOf_1

输入	运行时间	解步骤数
input0	0.03s	5
input1	0.007s	4
input2	0.279s	7
input3	0.192s	7
input4	16.428s	9
input5	1.018s	9
input6	超过100s！	无

输入	运行时间	解步骤数
input7	超过100s！	无
input8	超过100s！	无
input9	超过100s！	无

IDA* + NumOf_1

输入	运行时间	解步骤数
input0	0.033s	5
input1	0.01s	4
input2	0.347s	7
input3	0.062s	7
input4	0.458s	9
input5	42.648s	9
input6	超时	无
input7	超时	无
input8	超时	无
input9	超时	无

A*+Manhattan

输入	运行时间	解步骤数
input0	0.107s	5
input1	0.009s	4
input2	0.001s	5
input3	0.138s	7
input4	0.182s	9
input5	0.086s	11
input6	9.733s	11
input7	1.878s	26

输入	运行时间	解步骤数
input8	2.545s	32
input9	7.054s	39

IDA*+Manhattan

输入	运行时间	解步骤数
input0	0.201s	5
input1	0.015s	4
input2	0.002s	5
input3	0.183s	7
input4	0.179s	9
input5	0.087s	11
input6	8.029s	11
input7	1.222s	34
input8	2.45s	32
input9	8.96s	39

结果:

以input0为例

解步骤数:5
坐标：(1,0) 方式：1
坐标：(2,2) 方式：2
坐标：(0,3) 方式：4
坐标：(1,3) 方式：4
坐标：(1,4) 方式：2

Debug

在测试A*算法实现时，遇到报错：`Process finished with exit code -1073741819 (0xC0000005)`

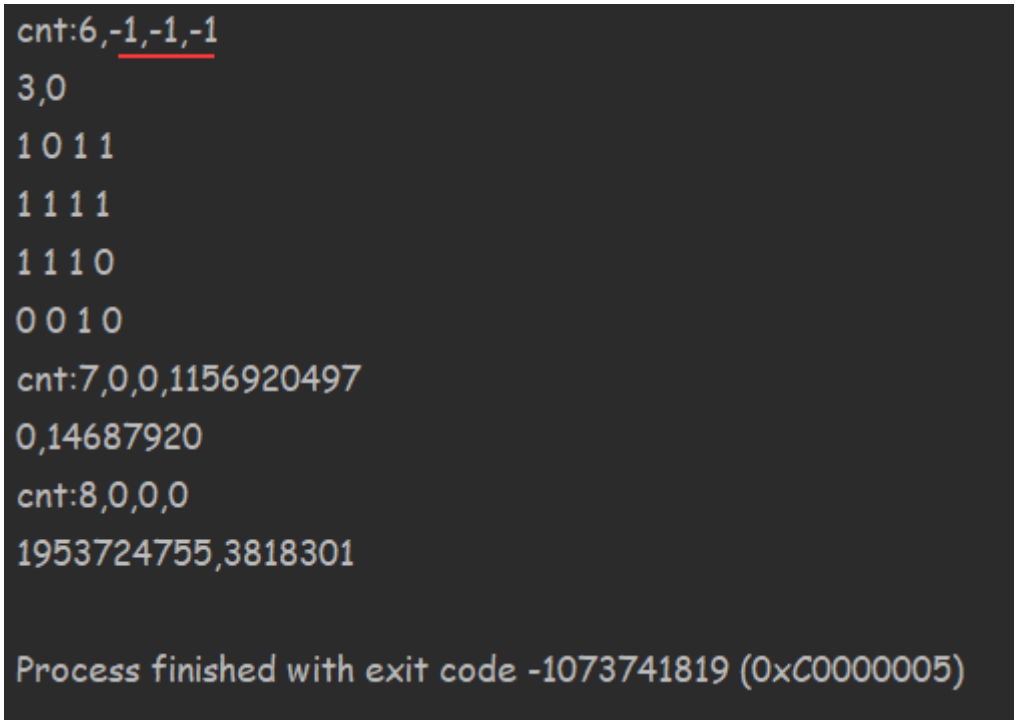
Process finished with exit code -1073741819 (0xC0000005) 是一种常见的错误代码，表示程序遇到了访问违规的情况，通常是由于以下几种原因之一导致的：

1. 空指针访问：在代码中使用了空指针进行访问，即尝试读取或写入空指针指向的内存地址。
2. 内存越界：访问了超出分配内存范围的数组元素或对象。

3. 栈溢出：递归或大量函数调用导致函数调用栈超出了系统限制。
4. 未初始化的变量：使用了未初始化的变量或对象。
5. 释放已释放的内存：重复释放已经被释放的内存块。

解决方法：

1. 仔细检查代码中可能存在的空指针访问或内存越界的情况。确保在访问指针之前进行有效性检查，并在必要时对指针进行初始化。
2. 检查是否有未初始化的变量，并确保在使用它们之前正确初始化。
3. 检查代码中的递归或大量函数调用情况，确保不会超出系统限制。
4. 检查是否存在重复释放已释放的内存的情况，确保每个内存块只被释放一次。



总结

通过AStar算法的实现，加深了对A*搜索的理解，以及IDAstar算法的理解，但是发现迭代Astar并不是在任何情况能够取得比Astar更好的效果，并且感觉启发式函数的选择对Astar算法影响非常大，也很难取得一个又快又能够找到最短路径的启发式函数，也就是找到一个能够完美估计实际代价的函数，在过程主要遇到的问题就是指针的使用要十分小心，在函数传递时传递的引用类型也需要注意，本次Astar实现的的代码量大概在670行左右，虽然用的数据是结构体，但是这个过程也是体现了面向对象的思想，对于用C++实现一个稍大规模的编程设计也加强了理解。