AILab1

姓名：徐宇鸣

学号：PB17111636

数码问题

Astar搜索算法：

算法流程：

```
do{
    //在优先队列中找一个F最小的节点作为n
    select a state as N in openlist;
    //如果就是终点，结束
    if N == endState then return endState;
    //这里我将closelist和openlist的visit标记进行了合并，里面存入了每个访问过或者仍然在openlist中的节点的深度，意思就是 拥有一个openlist优先队列， 以及一个字典用来确认访问过与否或者在不在openlist中
    //这样做是为了减少访问openlist的时候需要花大量的时间遍历优先队列去确认是否存在
    if N in closelist and openlist then {
        if the privious N's depth bigger than current N then {
            //我们在字典中取得的之前访问这个节点或者在openlist中的这个节点的深度比我们刚取出来的N来的大
            //更新这个节点的最小深度，这样下次碰到深度大的相同状态节点可以不需要访问
            update closelist and openlist;
        } else {
            //这个节点比我们之前搜的相同节点的深度来的大，跳过，不需要展开
            continue;
        }
    } else {
        //将节点插入
        update closelist and openlist;
    }
    //先展开节点，不是带7的滑块的移动
    for direction = "rlud", generate neighbor state without 7 {
        if N in closelist and openlist then {
            if the privious N's depth bigger than current N then {
                //我们在字典中取得的之前访问这个节点或者在openlist中的这个节点的深度比我们刚取出来的N来的大
                //更新这个节点的最小深度，这样下次碰到深度大的相同状态节点可以不需要访问
                //所用启发式为经过线性冲突加强后的曼哈顿
                update closelist and openlist;
            }
```

```
        } else {
            //将节点插入
            update closelist and openlist;
        }
    }
    //考虑移动滑块7
    for direction = "rlud", generate neighbor state with 7 {
        //与上面做一样的事
        //所用启发式为经过线性冲突加强后的曼哈顿
        do the same things
    }
}while(openlist is not empty)
```

所选启发式：

在这里所使用的启发式为经过线性冲突加强后的曼哈顿，线性冲突举个例子，就是比如我们给定状态要求顶行为（1，2，...），但是他实际出现为（2，1，...），那么我们如果要反转他们，必须将其中一个滑块移出顶行，允许其通过再移回顶行，由于这些移动不会计入任意块的曼哈顿距离，因此将这两个移动加入曼哈顿距离计算出来的值后并不会影响可采纳性

算法的时间复杂度：

这仍然是个NPC问题，A星算法只是通过启发式更早的搜索到了结果

在这里openlist我使用的是c++的priority_queue,因此top()和pop()操作的时间复杂度为O(1)，为常数

而close_and_openlist使用的是++的map，因此查找操作的时间复杂度为O(log(searched_size*(1+neighbor_factor)))，这里的含义是我们的map包含的应该是所有访问过的节点以及在openlist中的节点，及访问过的节点+访问过的节点扩展出来的节点，即访问过的节点+访问过的节点*分支因子，而扩展邻居节点自然是O(neighbor_factor)，也为常数，因此访问一个节点的总操作即为O(log(searched_size*(1+neighbor_factor))),

因此总的时间复杂度应为O(log(1*(1+neighbor_factor) + ... +log(total_search_size*(1+neighbor_factor))),假设总访问的节点数为n，那么就为O(log(n!)+nlog(1+neighbor_factor)) = O(log(n!)) = O(nlogn);

而空间复杂度：

很显然，在我们找到终点前我们不能丢掉任何节点的信息，因此所需要的空间复杂度为O(n),n为访问的总节点数（注：事实上在这里我所用的空间应该比这里理论值来的大，因为同一个节点多次访问的时候我是开了多块空间来存，在写报告的时候才觉得应该能通过修改我们记录的值去减少一定的存储空间

输出结果：

```
input1.txt:
total use: 0.001708s
得到解步数：24步
input2.txt:
total use: 0.000696s
得到解步数：12步
input3.txt:
//实际执行时间是后来加上
//这个解所需要的状态空间太大，自己的电脑上内存不足（大概需要60G以上），最后是通过找其他同学帮忙跑出来的结果，
//因此在攥写报告的时候并没能记录实际具体的执行时间，由同学给出的数据大概在30分钟左右
total use: unknown
得到解步数：57步
```

IDAstar搜索算法：

算法流程：

```
idAstarSearch {
    //将当前深度路径的节点n取出来
    n = path[depth];
    //如果是超过指定深度的节点就不需要去扩展，而是考虑更新下一次的深度
    if n.F > limit then next_limit = min(f,next_limit)
    else {
        if n is end state then return true;
        //先展开节点，不是带7的滑块的移动
        for direction = "rlud", generate neighbor state without 7 {
            //这里的openlist的含义与上面的类似，就是包含了所有访问过以及在openlist中的节点
            //在这里为了加快计算时间，我拿空间换了时间，其实还是把所有访问过的节点的信息，比如g，h，目前是否在栈中，都存了下来，然后另外加了一个times，表示这是第几次迭代的记录
            //h的主要作用是因为线性冲突的计算会很麻烦，比起A*我们完全有能力将所有的h都存下来，因此为了节省时间我将所有计算过的节点的h值都记录了下来
            if N in openlist then {
                if N is not in the path {
                    //如果openlist中的记录是我们这轮迭代得到的结果，那么我们就像A*一样通过检查深度来判断是不是需要放入栈中
                    if the record of N is this time's record {
                        //这里做的事就跟A*是一样的了
                        if the privious N's depth bigger than current N then {
                            update openlist;
                            //递归，这里主要是不太习惯写迭代的DFS
                            if idAstarSearch then return true;
                            pop neighbor state;
                        }
                    } else {
                        //我们得到的记录是上一轮的记录
                        //更新记录中的信息
```

```
                        update openlist;
                        if idAstarSearch then return true;
                        pop neighbor state;
                    }
                }
            } else {
                //是没有计算过的节点，需要重新计算一遍h
                caculate h
                update openlist;
                if idAstarSearch then return true;
                pop neighbor state;
            }
        }
        //考虑移动滑块7
        for direction = "rlud", generate neighbor state with 7 {
            //与上面做一样的事
            //所用启发式为经过线性冲突加强后的曼哈顿
            do the same things
        }
    }
}
```

所用启发式：

没有改变，仍是线性冲突加强后的曼哈顿

算法的时间复杂度：

其实就是深度受限的DFS，因此也是需要遍历O(n)个节点，而每次遍历的时候因为要去字典查找是否存在该节点，因此也为O(nlogn),虽然和Astar没有差别，但是少了优先队列的插入时间，以及内存占用变少，没有了优先队列也不会因为优先队列的变长使得c++需要花时间去修改优先队列的内存分配或者进行优先队列的重分配

空间复杂度：

在这里由于我们用空间换了时间，我们将所有访问过的节点信息都存了下来，因此总空间使用为O(n)，n为访问过的节点，虽然看起来和Astar差不多，但是在Astar中我们任何一个节点存放的信息都包含了整个Board的信息，parent的信息，两个空格的信息等需要30多个字节（并且在Astar中我的写法里面同一个节点可能会存多个值），但是在这里我们要存的只需要4个字节，分别是是否在栈中，f值，g值，以及是第几次迭代的记录，并且没有了优先队列而是一个深度最大也只有57的vector

输出结果：

```
input1.txt:
total use: 0.001472s
得到解步数：24步
input2.txt:
total use: 0.000679s
得到解步数：12步
input3.txt:
total use: 601.780179s
得到解步数：57步
```

数独问题

sudoku.cpp：

最简单的回溯搜索算法：

首先最简单的就是普通的回溯, 每次选择空格只是选择了一行一行从左到右搜索的第一个空格

这样的结果为：

```
Normal:
sudoku01:
total use: 0.000289s
total search: 111 nodes

sudoku02:
total use: 0.005866s
total search: 14853 nodes

sudoku03:
total use: 1.583090s
total search: 4233934 nodes
```

加上启发式：

这里就是在原来代码的基础上修改了选择填空位置的方式，记录了每个数独盘上的行、列、宫格、对角线的空格数量(left数组)，优先选择最少空格的行/列/宫格/对角线，然后遍历这部分，找到第一个0

结果为：

```
add heuristic:
sudoku01:
total use: 0.000644s
total search: 99 nodes

sudoku02:
total use: 0.001308s
total search: 1830 nodes

sudoku03:
total use: 0.110417s
total search: 266841 nodes
```

可以看到，对于第一个我们减少了10%的节点，对于第二个减少了约85%的节点遍历，第三个减少了95%的节点遍历，在时间上，第一个节点使得允许时间增加到了原来的4倍多，而第二个减少到了原来了1/4，第三个减少到了原来的1/15

加上前向检验：

这里因为是在原来代码的基础上加上的，所以前向检验做的很粗糙，就是赋值后直接检查整个棋盘是否会使得某个还为空的格子没有办法赋值

结果为：

```
add forward checking:
sudoku01:
total use: 0.000518s
total search: 50 nodes

sudoku02:
total use: 0.002423s
total search: 279 nodes

sudoku03:
total use: 0.279575s
total search: 36951 nodes
```

可以看到，这样的结果又使得第一个的遍历节点减少为原来的1/2，而因为节点减少的多了，所以时间上减少了1/6，而对于第二个，遍历的节点减少为原来的1/7，但是时间上增加为原来的2倍，对于第三个，遍历节点减少为原来的1/7，但是时间也增加为原来的2.5倍左右

加上MRV：

最后就是加上了mrv，这里mrv是在原来的选择完行/列/宫格/对角线后，选择了可选值最少的格子而不是第一个可选的格子（mark[i][j][input]记录了（i，j）位置input这个数字是否可用，而remain[i][j]记录i，j位置还有多少个数字可用）

结果为

```
add MRV:
sudoku01:
total use: 0.001026s
total search: 49 nodes

sudoku02:
total use: 0.004263s
total search: 270 nodes

sudoku03:
total use: 0.280161s
total search: 28275 nodes
```

可以看到我们的结果，对于第一个和第二个并没有在遍历节点上造成显著影响，但是却因此导致耗时加倍，而对于第三个，时间增加不多，遍历节点减少到原来的3/4左右

sudoku_rewrite.cpp

前面的3种优化都是很粗糙地在原来的代码基础上增加的，因此时间上会影响很多，因此我重新修改重写了一遍，而且在启发式方面，我再经过考虑之后，应该把整个9x9的每个格子视作图上一个点，有影响的点之间连上一条边，这样每个点之间的约束就能更好地表示为度，每当一个点被着色（格子被填上数字后），相邻点需要更新剩余值以及剩下的相邻空格数（度），这样可能更接近于度启发式，并且在前面的时候我完全可以把mark数字改为mark[9][9][10]，这样用0位置来表示剩余可选值的数量

先用度启发式，再用MRV：

```
rewrite: first Degree heuristic, than MRV
sudoku01:
total use: 0.000602s
total search: 58 nodes

sudoku02:
total use: 0.010894s
total search: 7022 nodes

sudoku03:
total use: 2.881143s
total search: 2043076 nodes
```

可以看到，这样的做法不光节点遍历没有上一版的最优解好，时间上甚至比最简单的回溯还来的久，可以想到的就是，我们这样的做法一般就会优先选到周围全都是0的格子，然后这样的情况下MRV就没有了用处（因为这时候基本上可选值都是9），起到了反效果

先用MRV，再用度启发式：

```
rewrite: first MRV, than Degree heuristic
sudoku01:
total use: 0.000558s
total search: 47 nodes

total use: 0.000708s
total search: 83 nodes

sudoku03:
total use: 0.001264s
total search: 433 nodes
```

可以看到，先使用MRV后用度启发式的做法就能很好地进行剪枝，可以看到，三个结果在节点和执行时间上都优于之前所有的版本，并且对于sudoku01，我们是直接找到了结果（因为最开始填入的格子就为35格），而对于sudoku03，我们只遍历了433个节点就找到了答案，并且也只花了1ms左右

思考题：

遗传算法：

使用遗传算法就需要考虑：初始化种群、交叉、变异、选择

初始化种群：

为了减少进化代数，我们需要产生较优的初始种群，在这里较优的定义就是行/列/宫格/对角线重复的数字较少。之后染色体由每个数独所缺的数字一行一行从左到右的顺序连接

交叉：

将染色体随机两两组合，随机取两个染色体中间相同的位置进行交换，交叉完之后再将未交叉的重复元素用另一个染色体的重复元素交换（因为该染色体重复的元素就是另一个染色体缺少的元素）

变异：

设定一个变异率，随机选择一定数量的个体，选取一个九宫格的方格将该节点左右翻转

选择：

父代、子代、变异三部分染色体和在一起，计算每个染色体还原到九宫格中行列重复数字的个数，初始分为8*（9+9）=144，每重复一次减去一分，选择分数最高的作为下一轮的父代，当分数为144的时候，就结束

爬山算法/模拟退火算法：

能量计算：同一个九宫格，同一行/列任何两个数字如果一样能量就为1，当一个数独能量为0的时候说明完成数独，且此时能量也是最低的。

每个数独就拥有了一个能量值，数独问题就变成寻找最低能量问题，因此可以使用爬山算法/模拟退火算法