AI Lab1

PB17111585 张永停

P1、数码问题

一、启发式函数

曼哈顿距离(三个7移动算一次)

- 由于曼哈顿距离是没有障碍物时候移动到目标的距离，故实际移动距离大于等于曼哈顿距离，从而h可采纳

二、算法

(一)A*搜索

- 对于前两个测试样例，采用可采纳的启发式函数（曼哈顿距离)得到最优解
- 对于对后一个测试样例，由于步数多，该启发式函数在有限内存下无法得到解(16GB)，故修改启发式函数为

  $$曼哈顿距离(非7)+3*曼哈顿距离(7)$$

  之所以这样修改，是因为7所在块是一个整块，增大7的权重，可以使搜索更倾向先放7，以及将7移近目标后不再移远
- 使用stl优先队列存储open list，使用stl的`unordered_map`存储closed list(自定义Hash)
- 自定义数据类型`node`，代表访问节点，其中`OPER`存储二元组(数字，移动方向)

```
typedef struct node
{
    OPER father_operate;
    node* father;
    node* itself;
    vector<int> status;
    int zero_pos[2];
    int seven_pos; //only record pos of the first seven
    int g;
    double h;

    friend bool operator < (node a, node b)
    {
        return (a.g + a.h) > (b.g + b.h);
    }
}NODE;
```

- 部分代码如下

  `main`

```
    priority_queue<NODE> q;
    Mymap status_is_explored;
    q.push(*root);
    while (!q.empty())
    {
        NODE t = q.top();
        q.pop();
        result= expand_status(t);
        if (result != nullptr)
            break;
    }
```

`expand_status`

```
    Find the direction that zero can move
    if the status after moving is finish
        if not
            Find the status in closed list
                if find, then
                    if status cost less than it in close list, then
                            change the cost and push it into q
                    end if
                else
                    push status into q
                    and push it into close list
                end if
        if finish
            return the node
    end if
```

(二)IDA*

- IDA*所用的截断值是 f 耗散值 (g+h),每次迭代，截断值是超过上一次迭代阶段值的节点中最小的 f 耗散值
- 由于IDA*采用递归的方法实现，故IDA*不需要使用指针，只需在找到路径的时候返回，增添全局栈`path`，在找到合法状态时压入栈
- 启发式函数仍然选用曼哈顿距离

```
typedef struct node
{
    vector<int> status;
    OPER op;
    int zero_pos[2];
    int seven_pos; //only record pos of the first seven
    double h;

    friend bool operator < (node a, node b)
    {
        return  a.h >  b.h;
    }

}NODE;
```

- 代码

`main`

```
while (limit < MAX_DEEPTH)
    {
        next_limit = MAX_DEEPTH;
        stack<OPER>().swap(path);
        OPER result = dfs(limit, next_limit, 0, root, -1);
        if (result.dir != -1)
        {
            path.push(result);
            break;
        }
        limit = next_limit;
    }
```

`dfs`

```
OPER dfs(double limit, double& next_limit, int depth, NODE state,int dir)
{
    if ((depth + state.h) > limit)
    {
        if (next_limit > (depth + state.h))
            next_limit = (depth + state.h);
        return { -1,-1 };
    }
    Find all the direction that zero can move
        OPER road = dfs(limit, next_limit, depth + 1, next_state, next_state.op.dir);
        if (road.dir != -1)
        {
            path.push(road);
            return next_state.op;
        }
}
```

(三)优化

- A*曾经尝试先移动`1,2`再移动`6,7`，但这种方法没有7的曼哈顿距离*3快
- IDA*也试过先把`6,7`移动到一起，将原来的"7"形块改成方形块，但发现将'6'移动到'7'就很久
- IDA*迭代时候会出现很多重复状态，比如$(1,up),(1,down),(1,up)$，这一点在步数比较少的时影响不大，但对于第三个样例，就会迭代很久迭代不出来(2天)。为了减少一次dfs重复状态访问，增加全局变量`Mymap status_is_ex`，类型为`unordered_map`，在每次重新深搜时清零
- 在IDA*每一层搜索时，优先搜索`f`小的结点，这样找到的解更倾向最优解
- 对于test3，IDA*的启发式修改成

  $1.5*(\text{曼哈顿距离}(\text{非}7)+3*\text{曼哈顿距离}(7))$
- 之所将返回值*1.5，是为了让搜索趋向目标，而不是在每一层都分支太多，导致迭代时间长

(四)实验结果

由于Test1，2比较简单，故直接使用可采纳的启发式

数据	A*时间	A*步数	A*空间	IDA*时间	IDA*步数	IDA*空间
Test1	0ms	24	不计	0ms	24	不计
Test2	0ms	12	不计	0ms	12	不计

对于Test3

算法	启发函数	时间	内存	步数
A*	曼哈顿距离	>3天	>16GB	57
A*	7曼哈顿距离*3	148s	8GB	59
A*	1.2*(7曼哈顿距离*3+else)	12s	1GB	63
IDA*	曼哈顿距离	>2天	500MB	57
IDA*	1.2*(7曼哈顿距离*3+else)	80s	700MB	77
IDA*	4*(7曼哈顿距离*3+else)	0.438s	40MB	165

P2、X数独问题

一、回溯算法

- 使用深搜的方法，按坐标遍历，对每个为0的块，将1-9填入，判断是否满足要求，若满足，则接着填下一个块，若不满足，则回溯。
- 为了减少判断一个数字是否可以填入当前空格的时间，使用四个二维数组`rows[9][10]`,`cows[9][10]`，`grid[9][10]`，`diag[2][10]`，其中`rows[i][j]==1`表示第`i`行已经填过`j`了。这样可以在O(1)的时间内判断一个数是否可以填入一个空格，并且维护该数组的时间也是O(1)的

二、带启发式的回溯

- 之前回溯，按下标顺序来填空，但其实可以先选择可选数字少的空格填。
- 维护一个包含当前所有0的数组，同时还需要维护每个0的可选数目
- 为了删除方便，使用c++自带的`make_heap`来代替stl库的优先队列
- 使用结构体

```
typedef struct zero
{
    int x, y;
    int avaliable_num[10] = {0};
    int avaliable_sum = 0;
    friend bool operator < (zero a, zero b)
    {
        return a.avaliable_sum > b.avaliable_sum;
    }
}ZERO;
```

其中`avaliable_num[i]==1`表示`i`可以填在当前空格

`vector<ZERO> zero_list;`

- dfs代码

```
bool dfs_imp(void)
{
    if (zero_list.empty())
        return true;//填完所有空格

    sum_node++;
    ZERO z = zero_list.front();//取可选数最少的空格


    pop_heap(zero_list.begin(), zero_list.end());
    zero_list.pop_back();//删除该空格


    for (int k = 1; k <= 9; k++)
    {
        if((z.avaliable_num[k])==1)//k可以填入空格
        {
            vector<ZERO> before_change=zero_list;//保存状态
            vector<ZERO> change_can;
            change_status(z.x, z.y, k);//将空格处填k，同时修改其他空格的状态
            bool can_conti = candidate_delte();//使用侯选数法来减少状态，返回1代表当前状态不与侯选数法冲突
            if (can_conti)
            {
                sudoku[z.x][z.y] = k;
                bool result = dfs_imp();//继续递归
                if (result)
                    return true;
            }
            sudoku[z.x][z.y] = 0;//回溯
            zero_list = before_change;
            make_heap(zero_list.begin(), zero_list.end());
        }
    }

    zero_list.push_back(z);//回溯
    push_heap(zero_list.begin(), zero_list.end());


    return false;
}

```

- 状态维护

```
void change_status(int x, int y, int num)
{
    int block1 = 3 * (x / 3) + (y / 3);//空格所在宫

    for (int i = 0; i < zero_list.size(); i++)//对每个未填的空格维护状态
    {
        ZERO z = zero_list[i];
        int block2 = 3 * (z.x / 3) + (z.y / 3);
```

```
        bool flag = false;

        if ((z.x == x) && (z.avaliable_num[num] == 1))//在同一行，且候选数包括num
            flag = true;
        else if ((z.y == y) && (z.avaliable_num[num] == 1))//同列
            flag = true;
        else if ((block1 == block2) && (z.avaliable_num[num] == 1))//同宫
            flag = true;
        else if ((z.y == z.x) && (x == y) && (z.avaliable_num[num] == 1))
            flag = true;
        else if (((z.y + z.x) == 8) && ((x + y) == 8) && (z.avaliable_num[num] == 1))
            flag = true;

        if (flag)//修改状态
        {
            zero_list[i].avaliable_num[num] = 0;
            zero_list[i].avaliable_sum--;
        }
    }

    make_heap(zero_list.begin(), zero_list.end());
}
```

三、其他优化

- 侯选数法:若一个空格可选数目只有一个，那空格必须填这个数

```
bool candidate_delte(void)
{
    while(1)
    {
        if (!zero_list.size())//如果通过侯选数填完了
            return true;

        ZERO z = zero_list.front();

        if (z.avaliable_sum > 1)
        {
            bool delte_num = false;
            bool can_f = hidden_candidate_delte(delte_num);
            if (!delte_num)
                return can_f;
            continue;
        }

        if (z.avaliable_sum == 0)//如果没有侯选数
            return false;

        if (z.avaliable_sum == 1)//只有一个侯选数
        {
            int l;
            for (l = 1; l <= 9 && z.avaliable_num[l]!=1; l++);
            pop_heap(zero_list.begin(), zero_list.end());
```

```
            zero_list.pop_back();
            change_status(z.x,z.y,l);
            sudoku[z.x][z.y] = l;
        }
    }
}
```

- 隐侯选数法:
  - 若一行所有空格的可选数的并集小于该行的空格数目，则说明无可行方案(比如两个空格的候选数目都只有2)，对列、宫、斜同理
  - 若某个数字在一列的所有空格的的侯选数中只出现了一次时，那含这个数字的空格必填这个数
  - 为了实现隐侯选数的方法，使用四个二维数组以及八个一维数组：
    - `r[9][10]`：`r[i][j]==1`表示第`i`行`j`出现了
    - `r_num[9]`：`r_num[i]`表示第`i`行有多少个空格
    - `r_s[9]`：`r_s[i]`第`i`行出现了几个侯选数

```
bool hidden_candidate_delte(bool &delte_num)
{
    int r[9][10] = { 0 }, c[9][10] = { 0 }, g[9][10] = { 0 }, d[2][10] = { 0 };
    int r_num[9] = { 0 }, c_num[9] = { 0 }, g_num[9] = { 0 }, d_num[9] = {0};
    int r_s[9] = { 0 }, c_s[9] = { 0 }, g_s[9] = { 0 }, d_s[2] = { 0 };

    for (auto z : zero_list)
    {
        int x = z.x, y = z.y;
        int block = 3 * (x / 3) + (y / 3);
        r_num[x]++;
        c_num[y]++;
        g_num[block]++;
        if (x == y)
            d_num[0]++;
        if ((x + y) == 8)
            d_num[1]++;

        for (int k = 1; k <= 9; k++)
        {
            if (z.avaliable_num[k])
            {
                r[x][k] = 1;
                c[y][k] = 1;
                g[block][k] = 1;
                if (x == y)
                    d[0][k] = 1;
                if ((x + y) == 8)
                    d[1][k] = 1;
            }
        }
    }

    for (int i = 0; i < 9; i++)
```

```
    {
        for (int j = 1; j <= 9; j++)
            r_s[i] += r[i][j];
        if (r_s[i] < r_num[i])
            return false;

        for (int j = 1; j <= 9; j++)
            c_s[i] += c[i][j];
        if (c_s[i] < c_num[i])
            return false;

        for (int j = 1; j <= 9; j++)
            g_s[i] += g[i][j];
        if (g_s[i] < g_num[i])
            return false;
    }

    for (int i = 1; i <= 9; i++)
    {
        d_s[0] += d[0][i];
        d_s[1] += d[1][i];
    }
    if (d_s[0] < d_num[0])
        return false;
    if (d_s[1] < d_num[1])
        return false;

    int n = -1;
    for (auto z : zero_list)
    {
        n++;
        int x = z.x, y = z.y;
        int block = 3 * (x / 3) + (y / 3);

        if (r_s[x] == 1)
        {
            int j = 0;
            for (j = 1; j <= 9 && (!r[x][j]); j++);
            zero_list.erase(zero_list.begin() + n);
            change_status(x, y, j);
            sudoku[x][y] = j;
            delte_num = true;
            return true;
        }

        if (c_s[y] == 1)
        {
            int j = 0;
            for (j = 1; j <= 9 && (!c[j][y]); j++);
            zero_list.erase(zero_list.begin() + n);
            change_status(x, y, j);
            sudoku[x][y] = j;
            delte_num = true;
            return true;
        }

        if (g_s[x] == 1)
        {
```

```
            int j = 0;
            for (j = 1; j <= 9 && (!g[block][j]); j++);
            zero_list.erase(zero_list.begin() + n);
            change_status(x, y, j);
            sudoku[x][y] = j;
            delte_num = true;
            return true;
        }

        if (d_s[0] == 1)
        {
            int j = 0;
            for (j = 1; j <= 9 && (!d[0][j]); j++);
            zero_list.erase(zero_list.begin() + n);
            change_status(x, y, j);
            sudoku[x][y] = j;
            delte_num = true;
            return true;
        }
        if (d_s[1] == 1)
        {
            int j = 0;
            for (j = 1; j <= 9 && (!d[1][j]); j++);
            zero_list.erase(zero_list.begin() + n);
            change_status(x, y, j);
            sudoku[x][y] = j;
            delte_num = true;
            return true;
        }

    }

    return true;
}
```

四、实验结果

对于前两个数独，由于比较简单，只列出简单回溯法与最后优化的结果，这里每进行一次dfs,节点数增加一

测试文件	回溯时间	回溯访问节点数	优化时间	优化节点数
sudoku1	0ms	463	0ms	1
sudoku2	0ms	66263	0ms	3

对于第三个数独

算法	时间	访问节点数
简单回溯	1125ms	18515640
加启发函数	0ms	1710
添加侯选数法	0ms	163

五、思考题

可以通过爬山算法、退火、遗传算法解决

- 爬山算法、退火算法
    - 先将数独填好，按行不冲突的方式
    - 统计数独冲突数目
    - 修改一个空格填的数目，使得当前的冲突数目减少
    - 问题：对于爬山算法，可能得不到可行解
- 遗传算法
    - 对所有待定单元格随机生成数值，组成字符串，作为一个样本。生成50对。之后随机两两组合，随机位置交换，随机位变异，生成新的50对样本。
    - 和上一代一共200个样本，然后计算每个样本的冲突次数，选择最小的100个样本再生成下一代。
    - 只到有零冲突样本出现，结束
    - 问题:得到可行解的时间可能会很久