AI lab1 Report

PB17111577 宁雨亭

实验环境：

MacBook Pro (13-inch, 2019)

Memory：16 GB

CPU：2.4 GHz 四核Intel Core i5

> 所有代码编译运行方法在根目录下README中给出

P1:数码问题

启发式

可采纳的启发式是**Manhattan距离**，$h=所有数字的Manhattan距离之和$，下面对其证明：

对应的松弛问题为：假定不管想要移动的块的相邻位置是否为空，都进行移动。这一启发式即为原问题的松弛问题需要移动的步数。因此启发式自然是可采纳的。

但是，在实验过程中，为了优化运行时间，我们使用的实际启发式**使用Manhattan的简单加权** $h=1.2*(7的Manhattan距离*3+其余数字的Manhattan距离之和)$。在实验结果部分，对不同启发式的结果进行了比较。

算法思想

A*

算法主要部分**伪代码**：

```
// visited_state保存所有已经访问过的状态，visited_states[state]=f
// f表示此state对应的最小f值，用于在更新评价函数时进行判断
创建unordered_map<string, float> visited_states;

// astar_search返回最终搜索到goal时的节点。
// 在输出路径时从此节点开始通过parent域即可找到所有节点
// start为搜索的初始状态
function astar_search(start){
  // 在open中每个节点以f值从小到大组成优先队列
  创建优先队列open;
  根据start创建节点init_node;
  step = 0;
  open.push(init_node);
  // 将init_node的state（也就是start）和对应的f值存入visited_states
```

```
  visited_states[*(init_node->state_str)] = init_node->f;
  step ++;
  while (open is not empty ){
    //得到f最小的节点，并将其移出open
    node = open.top();
    open.pop();
    // 判断是否已经到达goal的状态
    if node->is_goal then {
      break;
    }
    else {
      // 扩展相邻节点的算法：
      // 1. 首先扩展所有合法的移动数字和方向：
      // 对两个空格上下左右的块分别进行检查能否移动到空格处，如果可以移动，则记录(i,j)和方
向
      // 2. 扩展children节点：
      // 根据记录的(i, j)和对应的方向，进行移动得到新的子节点
      // 得到新的子节点时需要更新state、f、g、h属性，并将parent设为当前被扩展节点
      // 子节点的num、d域记录从parent到当前节点需要向哪个方向移动哪个数字块
      // 得到新的子节点时，如果是已访问过的state，且f更大，则抛弃
      // 如果f更小或未访问过，则加入children列表，且更新visited_states
      扩展每一个相邻节点，得到node的children列表node->children;
      for(each child in node->children) {
        open.push(child);
        step += 1;
      }
    }
    // 每个node被扩展之后，state就不再重要（因为已在visited_states里记录f和state）
    // 因此在扩展之后，可以释放state和state_str域以节省空间
    // 为了实现这一释放，在node结构体中需要将state定义成指针
    // 释放时释放指针所指的空间，并将指针置为NULL
    free掉当前node中不再需要的属性;
  }
  return node;
}
```

在实现A*算法时定义了Node类，扩展子节点、释放属性等在类中实现，实现思路已在上述伪代码中描述。

IDA*

IDA*的实现和A*类似，仍然使用相同定义的Node类，IDA*算法按照实验指导提供的伪代码框架实现。

算法主要部分**伪代码**：

```
// visited_state保存所有已经访问过的状态，visited_states[state]=f
// f表示此state对应的最小f值，用于在更新评价函数时进行判断
创建unordered_map<string, float> visited_states;

// idastar_search返回最终搜索到goal时的节点。
```

```
// 在输出路径时从此节点开始通过parent域即可找到所有节点
// start为搜索的初始状态
function idastar_search(start) {
  int d_limit = 0;
  根据start创建节点init_node;
  step = 0;
  d_limit = init_node->f;
  while (true) {
    int next_d_limit = INT_MAX;
    // 每次迭代清空访问过的节点
    visited_states.clear();
    // 在open中每个节点以f值从小到大组成优先队列
    创建优先队列open;
    open.push(init_node);
    visited_states[*(init_node->state_str)] = init_node->f;
    step ++;
    while (open is not empty) {
      // 得到f最小的节点，并将其移出open
      node = open.top();
      open.pop();
      // 如果大于截断值
      if(node->f > d_limit) {
        // 下一次截断值是这一次超过截断值的节点中的最小f
        next_d_limit = min(next_d_limit, node->f);
      }
      else {
        // 小于截断值时
        if (node->is_goal()){
          // 已搜索到goal
          return node;
        }
        else{
          // 扩展相邻节点的算法和A*一致
          扩展每一个相邻节点，得到node的children列表node->children;
          for(each child in node->children) {
            open.push(child);
            step += 1;
          }
        }
      }
      free掉当前node中不再需要的属性;
    }
    // 更新截断值
    d_limit = next_d_limit;
    // 把当前生成的所有节点删除，重新生成初始节点
    delete init_node;
    init_node = new Node(start, NULL);
  }
  return NULL;
```

```
}
```

算法分析

对Node类进行**时空复杂度分析**

```
class Node {
    public:
        // 每个Node节点需要空间O(n^2)
        int (*state)[5] = NULL;
        string* state_str = NULL;
        int pos_i_0s[2] = {-1, -1};
        int pos_j_0s[2] = {-1, -1};
        Node* parent = NULL;
        vector<Node*> children;
        vector<int> moves_i;
        vector<int> moves_j;
        vector<direction> moves_d;
        int num;
        direction d;
        int f = 0;
        int g = 0;
        int h = 0;

        Node(int cur_state[5][5], Node *cur) // O(n)
        ~Node() // O(1)
        void free_state() // O(1)
        void expand_node() // O(n^2)
        void move(int i, int j, direction move_direction) // O(1)
        bool is_goal() // O(1)

    private:
        void get_center(int i, int j, int& center_i, int& center_j) //O(1)
        bool move_check(int i, int j, direction move_direction) // O(1)
        bool move_exist(int i, int j, direction move_direction) // O(1)
        void add_move(int i, int j, direction move_direction) // O(1)
        void setup_moves() // O(1)
        void setup_children() // O(n^2)
        void update() // O(n)
        int h3() // O(n)
};
```

A*

每次循环中需要调用expand_node函数，因此每次循环的时间复杂度是$O(n^2)$，其中n为网格边长。

A*算法的时间复杂度为$O(b^d)$，其中b和启发式有关，是有效分支因子；d是解所在深度。

因此**总时间复杂度为**$O(b^d * n^2)$

IDA*

每次循环中需要调用expand_node函数，因此每次循环的时间复杂度是$O(n^2)$，其中n为网格边长。

IDA*算法的时间复杂度为$O(b^d)$，其中b和启发式有关，是有效分支因子；d是解所在深度。

因此**总时间复杂度为**$O(b^d*n^2)$

实验结果

我们使用了两种启发式进行测试：记$h1=\text{所有数字}Manhattan\text{距离之和}$，$h2=1.2*(7\text{的}Manhattan\text{距离}*3+\text{其余数字的}Manhattan\text{距离之和})$

	1.txt	2.txt	3.txt
h1 A*	step=499, time=5343us	step=108, time=1014us	x
h1 IDA*	step=534, time=4609us	step=108, time=1011us	x
h2 A*	step=231, time=2402us	step=91, time=1172us	step=2669542, time=16259439us
h2 IDA*	step=266, time=3121us	step=91, time=1154us	step=4501596, time=29197251us

使用h1时，第三个测试样例由于占用内存过大、时间太久无法完成。

最终我们使用启发式h2，对三个问题进行求解。前两个测试样例均可以得到**最优解**，分别为24、12步。第三个测试样例无法得到最优解，但是可以**快速解出(A*只需要16s)**，得到的解为63步。

在进行第三个测试样例时，内存最大占用量情况为：

实际内存大小：	1.26 GB
虚拟内存大小：	5.37 GB
共享内存大小：	348 KB
专用内存大小：	1,009.8 MB

P2:X 数独问题

算法思想

通过实现一个 CSP 问题的**回溯搜索算法**(backtracking search)来解给定的 X 数独问题，并通过**MRV启发式**决定选择变量的顺序、**利用约束条件和前向检验提前减少搜索空间**方法进行优化。

首先实现一个**基本的回溯搜索算法**，其主要部分为：

```
void BackTracking() {
```

```
    int next = SelectUnassigned();
    if (next == -1) {
        PrintSudoku();
        flag = 1;
        return;
    }
    int row = next / 9;
    int col = next % 9;
    while ((map[row][col] = GetNextPossibleAns(row, col)) != 0) {
        assign[row][col] = 1;
        step ++;
        BackTracking();
        if(flag) break;
        assign[row][col] = 0;
    }
    return;
}
```

其中`SelectUnassigned()`函数找到下一个未赋值的位置，并返回其位置（用0-80表示），若返回-1则表示已经全部赋值。

```
int SelectUnassigned() {
    // 找到下一个未解决的位置，若返回-1则全部解决
    for (int i = 0; i < 9; i++){
        for (int j = 0; j < 9; j++){
            if (assign[i][j] == 0)
                return i * 9 + j;
        }
    }
    return -1;
}
```

`GetNextPossibleAns(int row, int col)`函数计算`(row, col)`处下一个合法的赋值，若无合法的赋值则返回0。

```
int GetNextPossibleAns (int row, int col){
    for(int i = map[row][col]+1; i < 10; i++){
        if (IsConsistent(row, col, i)) {
            return i;
        }
    }
    return 0;
}
```

若不增加任何约束条件来提前减小搜索空间，则不需要IsConsistent函数，只需要对赋值递增进行搜索即可。但是我们在实现基本的回溯算法时，就**添加了约束条件，利用函数IsConsistent提前减少搜索空间。**

`IsConsistent(int row, int col, int value)`判断在`(row, col)`处赋值`value`是否合法：

根据X数独的规则，需要检查`(row, col)`所在行、列是否有重复值，检查所在3*3小方块内是否有重复值，如果在九宫格对角线上还需要检查所在对角线上是否有重复值，一旦出现重复值，则返回0表示不合法，否则返回1表示合法。

```
int IsConsistent(int row, int col, int value) {
    for (int i = 0; i < 9; i++) {
        // 行
        if (i!= col && map[row][i] == value) return 0;
        // 列
        if (i != row && map[i][col] == value) return 0;
    }
    //小方块
    int box_row = row / 3 * 3;
    int box_col = col / 3 * 3;
    for (int i = 0; i < 3; i++){
        for (int j = 0; j < 3; j++){
            if(box_row + i != row && box_col + j != col)
                if (map[box_row + i][box_col + j] == value)
                    return 0;
        }
    }
    // X
    if(row == col){
        for (int i = 0; i < 9; i++){
            if (i != row && map[i][i] == value) return 0;
        }
    }
    if (row == 8-col) {
        for (int i = 0; i < 9; i++){
            if (i != row && map[i][8-i] == value) return 0;
        }
    }
    return 1;
}
```

我们使用这一增加了约束条件的回溯算法作为优化前的算法，在此基础上使用启发式和前向检验进行优化。

这里我们使用**最小剩余值（MRV）启发式**，计算`(row, col)`处可取值的个数。

```
int MRVHeuristic(int row, int col) {
    // 返回(row, col)元素的可取值的个数
    int mrv = 0;
    for (int i = 1; i <= 9; i++){
        if (IsConsistent(row, col, i))
            mrv ++;
    }
    return mrv;
}
```

根据启发式的值修改选择变量的顺序，即在`SelectUnassigned()`函数中在各个未赋值的元素中选择mrv值最小的一个，作为下一个赋值的元素。

如果出现了**mrv值为0**的元素，也就是出现了无可取值的元素，则无解，不需要继续搜索，返回-2进行回退，这一过程相当于**前向检验**。如果mrv在函数最后仍为10，表示没有元素未赋值，所有元素均被解决，返回-1。

```
int SelectUnassigned() {
    // 找到未解决的位置中度最大的一个，若返回-1则全部解决，-2则无解
    int row = 0;
    int col = 0;
    int mrv = 10;

    for (int i = 0; i < 9; i++){
        for (int j = 0; j < 9; j++){
            if (assign[i][j] == 0) {
                if (MRVHeuristic(i, j)){
                    if (mrv > MRVHeuristic(i, j)) {
                        row = i;
                        col = j;
                        mrv = MRVHeuristic(i, j);
                    }
                }
                else {
                    // 如果mrv是0，表示出现无可取值的元素，无解，返回-2
                    return -2;
                }
            }
        }
    }

    // 如果mrv是10，表示没有未解决的元素，返回-1
    if (mrv != 10) return row * 9 + col;
    return -1;
}
```

实验结果说明与分析

优化前：

```
(base) ➜  src git:(master) ✗ ./basic
please input the test file name:
sudoku01.txt
2 8 1 5 3 4 9 6 7
5 6 4 9 7 8 1 3 2
7 9 3 1 6 2 4 8 5
9 2 5 7 1 6 8 4 3
4 1 7 3 8 5 6 2 9
6 3 8 2 4 9 7 5 1
3 4 9 6 2 1 5 7 8
8 5 2 4 9 7 3 1 6
1 7 6 8 5 3 2 9 4
step: 110
time: 161 us
```

```
(base) ➜  src git:(master) ✗ ./basic
please input the test file name:
sudoku02.txt
9 4 5 3 7 1 8 6 2
1 3 2 6 8 5 4 7 9
8 7 6 4 2 9 5 1 3
2 9 7 8 5 6 3 4 1
3 6 4 7 1 2 9 8 5
5 1 8 9 3 4 7 2 6
6 5 3 1 4 7 2 9 8
7 8 1 2 9 3 6 5 4
4 2 9 5 6 8 1 3 7
step: 14852
time: 8521 us
```

```
(base) ➜  src git:(master) ✗ ./basic
please input the test file name:
sudoku03.txt
6 7 5 3 1 8 2 4 9
8 9 4 6 2 5 7 1 3
1 3 2 4 7 9 5 6 8
9 4 1 5 3 6 8 7 2
7 5 3 8 4 2 6 9 1
2 8 6 7 9 1 4 3 5
5 1 8 9 6 7 3 2 4
4 2 7 1 5 3 9 8 6
3 6 9 2 8 4 1 5 7
step: 4233933
time: 1250018 us
```

优化后：

```
(base) ➜  src git:(master) ✗ ./Xsudoku
please input the test file name:
sudoku01.txt
2 8 1 5 3 4 9 6 7
5 6 4 9 7 8 1 3 2
7 9 3 1 6 2 4 8 5
9 2 5 7 1 6 8 4 3
4 1 7 3 8 5 6 2 9
6 3 8 2 4 9 7 5 1
3 4 9 6 2 1 5 7 8
8 5 2 4 9 7 3 1 6
1 7 6 8 5 3 2 9 4
step: 46
time: 1060 us
```

```
(base) ➜  src git:(master) ✗ ./Xsudoku
please input the test file name:
sudoku02.txt
9 4 5 3 7 1 8 6 2
1 3 2 6 8 5 4 7 9
8 7 6 4 2 9 5 1 3
2 9 7 8 5 6 3 4 1
3 6 4 7 1 2 9 8 5
5 1 8 9 3 4 7 2 6
6 5 3 1 4 7 2 9 8
7 8 1 2 9 3 6 5 4
4 2 9 5 6 8 1 3 7
step: 106
time: 3965 us
```

```
(base) ➜  src git:(master) ✗ ./Xsudoku
please input the test file name:
sudoku03.txt
6 7 5 3 1 8 2 4 9
8 9 4 6 2 5 7 1 3
1 3 2 4 7 9 5 6 8
9 4 1 5 3 6 8 7 2
7 5 3 8 4 2 6 9 1
2 8 6 7 9 1 4 3 5
5 1 8 9 6 7 3 2 4
4 2 7 1 5 3 9 8 6
3 6 9 2 8 4 1 5 7
step: 3792
time: 97083 us
```

优化前后比较：

	优化前	sudoku02.txt	sudoku03.txt
Step（优化前）	110	14852	4233933
Step（优化后）	46	106	3792
Time（优化前）	161us	8521us	1250019us
Time（优化后）	1060us	3965us	97083us

说明与分析：

1. 由于三个测试样例运行的时间都很短，优化前最长的一个测试样例也只需要1s左右，所以程序运行时间的测试误差很大，多次运行结果偏差不容忽略，因此程序运行时间的测量只能用作参考。
2. 观察搜索步数，可以看到三个测试样例在优化后搜索步数都明显减少，说明我们使用的优化方法是有效的。
3. 观察搜索时间，会发现测试样例2、3的搜索时间都明显减少，但测试样例sudoku01.txt的搜索时间反而增加。这是由于第一个测试样例的搜索步数在优化前后都很少，优化前运行时间就很短，但是我们进行了优化后，在搜索过程中的辅助计算增多，在搜索步数少时辅助计算的时间更长，导致搜索时间反而变长。当搜索步数增多时，优化效果就明显了。

思考题

a) X 数独这个问题是否可以通过爬山算法、模拟退火算法或是遗传算法等算法来解决?如果能的话，请给出大致的思路。

可以。

需要对数独状态的评估进行量化为h。设同一个九宫格，同一行，同一列、同一对角线任何两个数字如果一样那么h就+1。将数独问题转换成一个使h最小的问题。

- 爬山算法：

  首先对数独进行随机填充初始化，每个节点的后继即为修改一个数字后的状态节点，每次选取使h最小的后继进行继续循环。

- 模拟退火算法：

  首先对数独进行随机填充初始化，每个节点的后继即为修改一个数字后的状态节点，每次随机修改一个数字，如果修改后的节点h更小，则接受，否则以一个小于1概率的概率（如0.5）接受。

- 遗传算法：

  首先对数独进行随机填充初始化，产生多种填充方案，作为初始群体。h越小适应度越高，适应度越高被选择进行繁殖的概率越高。根据适应度函数选择两个个体进行交叉（比如随机选择某一行、某一列、某一块，将两种填充方案对应位置的值互换），产生新的填充方案后，进行变异（随机选择一些位置改变数字），以此得到新的群体。循环上述过程直到出现符合条件的数独状态。

b) 如果使用爬山算法、模拟退火算法或是遗传算法等算法来解决，可能会遇到哪些问题?

- 爬山算法：可能会陷于局部最优而无法得到解。
- 模拟退火算法：对参数依赖较强，而参数难以控制，不能保证一次就收敛到最优值，一般需要多次尝试才能获得，大部分情况下还是会陷入局部最优值。
- 遗传算法：可能通过交叉变异产生更差的状态，求解速度慢。需要根据经验控制交叉率、变异率等参数，而效果对这些参数依赖较强。