人工智能基础 实验报告

PB17050965 朱子琦

一、实验任务

任务1：数码问题

1. 为该问题寻找一个可采纳的启发式函数，并证明你的结论。
2. 依照你设计的启发式函数，分别实现 A*搜索算法和迭代 A*搜索算法（迭代 A*搜索算法见附页）进行以下 3 个不同难度的初始状态求解，并输出合法的解。
3. 问题的求解并不一定需要严格服从以上两个算法的流程，可以根据问题的性质加入你自己策略对算法进行微调，但算法的大致框架仍要属于 A*搜索与迭代 A*搜索算法。例如，对于该问题的一个可行但并不一定最优的解法可以为：a.采用搜索策略将“7”型数字块移动到正确的位置；b.将“7”型数字块固定视为障碍物，采用搜索策略把其他数字块移动到对应的位置。

任务2：数独问题

1. 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。
2. 对上述实现的算法进行优化，包括：
   a) 设计一个启发式来决定选择变量的顺序，如度启发式。
   b) 利用约束条件来提前减小搜索空间，如前向检验方法。
   c) 或者其他一些你认为可行的优化方法。
3. 将优化过的算法与原始的算法结果进行比较分析，分析角度：
   a) 搜索遍历的节点数
   b) 搜索所花的具体时间
   c) …
4. 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。
   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？

二、实验环境

本次实验要求使用C/C++进行编程，实验中使用全部的编译及编辑环境为

Dev-C++ *5.11*
TDM-GCC *4.9.2 32-bit Release*
Visual Studio Code *1.45.1*
C/C++ Compile Run *1.0.8*

三、实验内容

1.数码问题

1. 启发式函数

本实验的一个可采纳的启发式函数中 $h(x)$ 是数码块与目标位置间的曼哈顿距离。其中数码块7以右上角位置作为参照。每次移动其曼哈顿距离至多减小1，则这是一个乐观的估计，可采纳。
而在本次实验中，当涉及到输入3时，使用原始的曼哈顿距离搜索效率较低，于是在此基础上，将非7数字的曼哈顿距离x2，7数码的曼哈顿距离x4，得到一个基于参数与曼哈顿距离的新的启发函数。

2. 算法实现

评价函数的计算

使用参数**SCALE1 = 2 SCALE2 = 4**，用于修正对数码7和其他数码计算时采纳的权重。通过比较当前位置与其目标位置间的加权曼哈顿距离，得到本次实验中使用的启发式函数。

```
int measure(char map[SIZE*SIZE]){
    int value = 0;
    int i = 0, j = 0;
    int t1, t2;
    int num;
    for( i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++){
            num = map[i*SIZE+j];
            if(num == 7){
                if (map[(i+1)*SIZE+j] == 7){
                    t1 = i - x[num];
                    t2 = j - y[num];
                    t1 = (t1 >= 0) ? t1 : -t1;
                    t2 = (t2 >= 0) ? t2 : -t2;
                    value += SCALE1 * (t1 + t2);
                    continue;
                }
                else {
                    continue;
                }
            }
            if(num == 0){
                continue;
            }
            t1 = i - x[num];
            t2 = j - y[num];
            t1 = (t1 >= 0) ? t1 : -t1;
            t2 = (t2 >= 0) ? t2 : -t2;
            value += SCALE2 * (t1 + t2);
        }
    }
    return value;
}
```

A*算法的实现

伪代码如下所示

```
void astar(){
    int i,min;
    int k = 1;
    while(finish != 1) {
        for (i = 0, min = INFTY; i <= position; i++){       // 寻找当前f最小的节点
            if(maplist[i].f_value < min){
                min = maplist[i].f_value;
                cur_pos = i;            // 选取当前f最小的节点，进行拓展
            }
        }
        search(cur_pos);          // 进行搜索i
        k++;
    }
    get_result(final_pos); // 根据最终得到的目标状态，递归输出其父节点，得到完整的搜索路径。
}
```

搜索过程如下所示，检查符合条件的移动，进行拓展。

```c
void search(){
    int i;
    int flag[4] = {0, 0, 0, 0};   // 用于检查是否是对7进行的移动
    mapmsg cur_map = maplist[cur_pos];
    char base[2] = {
        cur_map.zero[0].x * SIZE + cur_map.zero[0].y,
        cur_map.zero[1].x * SIZE + cur_map.zero[1].y
    };

    // 检查可行的移动的方式
    for (i = 0; i < 2; i++){
        if ((cur_map.zero[i].x > 0) && cur_map.map[base[i] - SIZE] != 0)
        {
            if (cur_map.map[base[i] - SIZE] != 7) move(i, 'd');
            else flag[0]++;
        }
        if ((cur_map.zero[i].x < 4) && cur_map.map[base[i] + SIZE] != 0)
        {
            if (cur_map.map[base[i] + SIZE] != 7) move(i, 'u');
            else flag[1]++;
        }
        if ((cur_map.zero[i].y > 0) && cur_map.map[base[i] - 1] != 0)
        {
            if (cur_map.map[base[i] - 1] != 7) move(i, 'r');
            else flag[2]++;
        }
        if ((cur_map.zero[i].y < 4) && cur_map.map[base[i] + 1] != 0)
        {
            if (cur_map.map[base[i] + 1] != 7) move(i, 'l');
            else flag[3]++;
        }
    }
    if(flag[0] == 2)
        move(2, 'd');
    if(flag[1] == 2)
        move(2, 'u');
    if(flag[2] == 2)
        move(2, 'r');
    if(flag[3] == 2)
        move(2, 'l');
    maplist[cur_pos].f_value = INFTY; //set invalid
}
```

A*算法的时空复杂度分析

A*算法的时间复杂度为 $O(a^{2n})$ 其中 n 为搜索深度，a为常数

空间复杂度为 $O(a^n))$ 所搜索的空间最大为一个n层的树

注意，其中n为搜索深度。在astar()函数中，每次需要寻找最小f值的节点，用时$O(a^n)$，a为一个常数参数这一过程最坏情况下会调用a^n次。 在search()函数中，每一步时间固定，用时O(1)。measure函数用时O(1)。search()函数中将会调用move()函数， 由于在搜索中要对状态进行去重，move函数中会执行check()函数检查当前已得到状态中是否有新状态

IDA*算法的实现

IDA*算法思路基本与A*算法相同，核心在于搜索方式以及对下一节点的选取方式不同。A*算法中选择f值最小的进行扩展，而IDA*在这里不做选择，但会截断超过f值阈值的分支。

伪代码如下所示

```
void idastar(){
    d_limit = maplist[0].f_value;
    while(finish != 1){
        mid_time = time(0);
        if(dfsearch() == 0){
            max = 0;
            d_limit = next_d_limit;
            next_d_limit = INFTY;
            position = 0;
            cur_pos = 0;
        }

        else {
            return;
        }
    }
}
```

当没有搜索到目标解时，更新f阈值，并进行深度优先搜索。

IDA*算法的时空复杂度分析

IDA*算法的时间复杂度为 $O(a^{2n})$ 其中 n 为搜索深度，a为常数

空间复杂度为 $O(n)$ 所搜索的空间最ID大为一个n层的树的一支

注意，其中n为搜索深度。在idastar()函数中，每次需要进行深度优先搜索，最坏情况下搜索n层用时$O(a^n)$，a为一个常数参数这一过程最坏情况下会调用a^n次。 在search()函数中，每一步时间固定，用时O(1)。measure函数用时O(1)。search()函数中将会调用move()函数，时间开销为O(1)。

对比

尽管IDA*和A*最坏情况下时间复杂度相同，但是平均情况下，A*搜索通常能够在将树填满前，甚至可能只是填了某一支的某一部分就已经找到了答案，而IDA*则至少需要填满其前一层(指根据f层面)，找到所

有f小于所求解的可能，故平均效率上A*算法要更好一些，但IDA*算法可以极大节省空间，二者各有利弊。

2.数独问题

1.回溯搜索算法

原始的回溯搜索算法如下所示

```
int sudoku(){
        int i;
        count++;
        char min_pos;
        if(num == 0) return 1;
        for(i = 0; i < 81; i++){
                if(filled[i] == 0){
            min_pos = i;
            break;
        }
        }

        for (i = 1; i < 10; i++){
                if(selection[min_pos][i]!=0){
                        input[min_pos] = i;
            if(check(min_pos)){
                filled[min_pos] = 1;
                num--;
                if(sudoku() == 1){
                    return 1;
                }
                else {
                    input[min_pos] = 0;
                    num++;
                    filled[min_pos] = 0;
                        }
            }
            input[min_pos] = 0;
        }
        }
}
```

在这一过程中，当遇到第一个无指定值的块时检查其可能取值表，选择其中的值检验当前是否合法。若合法则递归调用用sudoku()函数，进入下一层搜索，否则清除状态，继续检查下一个值。

2.使用度启发式进行优化

可以发现，在这样的计算过程中，无法通过提前减小搜索空间来减少搜索量，因此，对算法min_pos的选取进行了优化

```
for(i = 0; i < 81; i++){
    if(filled[i] == 0){
        if(degree[i] == 0) return -1;
        if(degree[i] < min){
            min = degree[i];
            min_pos = i;
        }
    }
}
```

这时，选择当前可选方案最少的块进行指派值，这样可以使得最大可能选择正确分支。

3.使用前向搜索进行优化

但依然可以看出，有时候当前一步已经选取一个值的时候，后一步无法及时得知该信息，可能重复选取值，这样是不合算的，于是，对算法加入前向检验功能。具体实现如下：

```
for (i = 1; i < 10; i++){
    if(selection[min_pos][i]!=0){
        input[min_pos] = i;
        filled[min_pos] = 1;
        getselection();
        getdegree();
        if(sudoku() == 1){
            return 1;
        }
        else {
            input[min_pos] = 0;
            selection[min_pos][i] = 0;
            filled[min_pos] = 0;
            getselection();
            getdegree();
        }
    }
}
```

每次搜索前，更新节点的度和取值范围，这样可以尽快得到度最小的节点且不会出现无意义的取值，使性能得到优化。

四、实验结果

数码问题实验结果

输入1

使用A*和IDA*结果均如下：

```
(1,u);(1,u);(6,u);(19,l);(15,d);
(7,l);(14,d);(11,d);(3,r);(2,r);
(1,u);(14,d);(11,d);(8,r);(7,u);
(6,u);(14,l);(14,l);(15,u);(20,l);
(16,l);(17,u);(21,l);(21,l);
```

共24步。用时均为1s。

输入2

使用A*和IDA*结果均如下：

```
(14,d);(6,d);(15,d);(7,l);(8,l);
(9,l);(10,u);(11,u);(16,u);(21,l);
(13,u);(18,u);
```

共12步。用时均为1s。

输入3

使用A*算法结果如下：

```
(6,l);(15,l);(21,r);(17,r);(16,d);
(3,d);(13,d);(4,l);(7,l);(5,u);
(5,u);(10,r);(10,u);(13,r);(13,r);
((4,d);(4,r);(9,r);(15,d);(9,d);
(7,l);(4,u);(4,u);(9,r);(9,u);
(3,u);(11,r);(3,r);(2,r);(15,d);
(15,d);(1,r);(8,d);(6,d);(7,l);
(2,u);(2,u);(1,r);(1,u);(8,r);
(6,d);(3,l);(9,d);(4,d);(2,r);
(1,u);(3,u);(8,r);(7,d);(1,l);
(1,l);(2,l);(2,l);(3,u);(8,u);
(11,u);(16,u);(4,u);(9,u);(12,u);
(17,u);(21,l);(21,l);
```

共63步。用时12630s。

由于程序的优化不是很好，时间有限，没有得到使用IDA*的结果。预计时间开销应该会大于A*算法。

在这一步还尝试过修改曼哈顿距离的权值来对算法进行优化。本结果是由SCALE1=2，SCALE2=4得出的。而当SCALE1=1，SCALE2=4时，也能得到同样的结果，但时间开销为27080s，说明计算过程比较

保守，进行了较多无用计算。

而当SCALE1=2，SCALE2=8时，得到了一个97步，用时4711s的结果。计算效率大大提高，但其步数距离最优解较远。通过对几组参数进行综合考虑，选择设定SCALE1=2，SCALE2=4作为程序的参数。

数独问题实验结果

数独1

```
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
```

数独2

```
1 4 5 3 6 9 8 7 2
8 3 2 1 7 5 4 6 9
9 7 6 8 2 4 5 1 3
2 9 7 5 8 6 3 4 1
3 6 4 2 1 7 9 8 5
5 1 8 4 9 3 7 2 6
6 5 3 7 4 1 2 9 8
7 8 1 9 3 2 6 5 4
4 2 9 6 5 8 1 3 7
```

数独3

```
6 7 2 1 3 5 8 4 9
1 9 4 6 2 8 5 7 3
5 3 8 4 7 9 6 1 2
2 4 1 3 5 6 7 9 8
7 5 3 9 8 4 2 6 1
8 6 9 2 1 7 4 3 5
9 8 5 7 6 1 3 2 4
4 2 7 5 9 3 1 8 6
3 1 6 8 4 2 9 5 7
```

搜索节点数

	输入1	输入2	输入3
进行优化	47	72	68
不进行优化	111	3269	47076

可以看出，优化后极大减小了搜索空间,使节点搜索加速。

时间开销

单位：秒

	输入1	输入2	输入3
进行优化	0.2373	0.2243	0.2206
不进行优化	0.2493	0.255	0.2418

由于本身计算量不是很大，主要时间开销在文件I/O而非计算上，因而优化前后时间差距不明显，但可以看出经过优化后的仍要比不优化的快一些。

五、思考题

对于数独问题，我认为可以采用爬山算法、模拟退火算法或是遗传算法等算法来解决。

大致思路如下：1.先随机为未摆放数码赋值。2.设置启发式函数：不合法的数码摆放数。3.修改其中的数码值，检查其启发式函数值。4.当函数值为0时停止修改，否则重复第3步。这里的启发式函数可能有很多种选法，也可能需要设置相应的参数使其效率得到提升。

难点在于，使用爬山算法、模拟退火算法或是遗传算法时，可能会遇到当前合法摆放数很多但实际需要修改很多才能得到正确的结果，落入陷阱。这时由于不像数值问题梯度明显，很难找到合适的方式进行调整，需要合理插入再启动机制使其能够避免过长时间困在错误的状态。同时由于需要保留较多的可能状态，其时空复杂度可能都比较差。