EXP 1 实验报告

PB17000209 许睿达

P1 数码问题

源代码编译运行方法见`./digit/src/readme.md`

A* 算法

启发函数

使用每个方块对目标距离的Manhattan距离对代价进行估计。可以证明该启发函数是可采纳、一致的。故可以使用A*算法的图搜索方法进行搜索。

> **可采纳性**：由于目标状态时所有块的Manhattan距离和为0，并且每次移动最多只能使得距离和减少1，故这样的启发式函数不会对代价高估，故为可采纳的。
>
> **一致性**：记该启发式函数为$f(n)$，对任意非最终结点n，他的后继节点n'，总有$f(n') \in \{f(n)-1, f(n), f(n)+1\}$成立（因为最多只能使得距离和改变1）。故$c(n,a,n')+f(n') = 1+f(n') \geq f(n)$，得到该启发式一致。

伪代码

A* 算法的运行过程如下，返回null则证明没找到解，否则返回找到的结点。

```
/* A* Search */

priority_queue = {}
closeSet = {}

while (!priority_queue.empty())
    now = priority_queue.top(); /*取队头*/
    priority_queue.pop();

```

```
    // Check Block 7
    for directions around block 7 do
        if (block 7 move is valid)
            new_node = now.makeNode(direction, position); /*构造新节点*/
            if (new_node in closeSet)
                delete new_node; /*释放空间*/
            else
                priority_queue.emplace(new_node); /* 入队列 */
                closeSet.insert(new_node->key); /*键值入已访问集合*/

    // Single move
    for all positions
        if (position is empty)
            for directions around empty position
                if (position + direction is not empty) /* 某个方向上非空 */
                new_node = now.makeNode(-direction, position + direction)
                /* 这里加减意思是传入附近的非空块，将空格的移动转化为附近非空方格的移动 */
                if (new_node is target)
                    return new_node;
                if (new_node in closeSet)
                    delete new_node;
                else
                    priority_queue.emplace(new_node);
                    closeSet.insert(new_node);

return nullptr; /*未找到解*/
```

> makeNode函数根据方向和位置建立新的结点。

优化过程

> （测试具体结果见下节）

我编写的最初版代码的查重方式是仅仅查找其祖先节点部分查重，但是这种情况下重复节点过多，导致内存迅速爆炸，故初版代码修改为使用线性结构进行查找`std::vector<Node *>`。

但是线性一一对结点进行分析，查重效率很低，为O(n)。从结果可以看到1,2均可以跑的出来，但是3完全不能跑出来（跑了一夜，只跑了五十万结点）。

但是在线性查重的情况下，如果对测试三进行优化，先修改权重使得搜索分为：

- 将1,2,6,7归位。
- 将其他块不变的情况下归位。

可以在20000结点左右找到结果（部分输出见下方）。然而优化后的代码对于测试一的表现极差，并不能在合理的时间下找到解。仅仅可以找到2,3的解。

对其继续进行优化，使用unordered_set中的`std::unordered_set`类型，通过内置的hash表进行查找，使查找时间接近一个常数。为了较为方便的使用STL库中的类型，我在节点内部存储了一个`std::string`类型的State_key，长为25字节，按行和列的顺序每一位存储了一个大小为（'A'+state[i][j]）的值，使得key较为直观，且仅仅需要使用string模板定义无序集合，就可以不用自己写hash函数而直接使用内置的hash和rehash函数。

此时程序是有希望在可以接受的时间内找到解的（借了服务器，跑了10分钟节点的评价函数已经到57步了），但是已经用了128G内存，直接报了std::bad_alloc。

退而求其次，我们可以给每个数字进行加权，从而使得移动这个块到正确位置变得更重要。构造如下：

```
#define STEP_COST 10

const int g_weight[] = {
    0,
    20,  18, 15, 12, 10,
    15,  30, 10, 10, 10,
             10, 10, 10,
    10,  10, 10, 10, 10,
    10,  10, 10
};
```

将代价函数×10，给需要优先恢复的节点加超过10的权值，虽然此时启发函数不再可采纳，但是这时已经可以找到非最优解了（63步，10s左右），同时也可以秒出测试1,2，达到了比较满意的效果。

结果分析

在同样开启-O3优化和-g调试选项的情况下：

最初未优化版本`digitAstar_initial.cpp`：

```
1.txt

Index: 129
Result:
Total Step(s): 24
Detail:
(1, u)
...
(21, l)
Searching takes: 0.00284 seconds

2.txt

Index: 24
Result:
Total Step(s): 12
Detail:
(14, d)
...
(21, l)
Searching takes: 0.00136 seconds

3.txt
// 跑不出来
```

为第三个测试集合定向优化版本`digitAstar_stage.cpp`：

```
1.txt
// 跑不出来
2.txt
```

```
Index: 17
Searching takes: 0.00086 seconds

Result:
Total Step(s): 12
Detail:
(14, d)
...
(21, l)

3.txt

Index: 19072
Searching takes: 16.24249 seconds

Result:
Total Step(s): 91
Detail:
(21, r)
(17, r)
...
(13, u)
(18, u)
```

使用`unordered_set`后的结果:

```
1.txt
****************Find Result!****************
Index:49
Searching takes: 0.00234 seconds

Result:
Total Step(s): 24
Detail:
(1, u)      (1, u)      (6, u)      (19, l)
...
(17, u)     (21, l)     (20, l)     (21, l)

2.txt

****************Find Result!****************

Index:20
Searching takes: 0.00126 seconds

Result:
Total Step(s): 12
Detail:
(14, d)     (6, d)      (15, d)     (7, l)
(8, l)      (9, l)      (11, u)     (16, u)
(10, u)     (13, u)     (18, u)     (21, l)
```

```
3.txt

***************Find Result!***************
Index:2509429
Searching takes: 10.74370 seconds

Result:
Total Step(s): 63
Detail:
(6, l)      (9, u)      (2, u)      (3, l)
(13, d)     (15, d)     (21, r)     (12, d)
...
(16, l)     (12, u)     (12, u)     (17, r)
(17, u)     (21, l)     (21, l)
```

> 在服务器上，仅仅给7块权重翻倍可以得到：
>
> ```
> Index:66336067
> Searching takes: 340.01378 seconds
>
> Result:
> Total Step(s): 59
> Detail:
> (6, l)          (15, l)         (21, r)         (17, r)
> ...
> (17, u)         (21, l)         (21, l)
> ```
>
> 这个参数吃了60G内存（显然不太行.jpg）

	测试集	解步数	扫描结点数	时间
未优化	1	24	129	2.84ms
	2	12	24	1.36ms
	3	-	-	-
阶段优化	1	-	-	-
	2	12	17	<1ms
	3	91	19072	16242ms
无序集合	1	24	49	2.34ms
	2	12	20	1.26ms
	3(对1,2,6,7等加权)	63	2509429	10743ms
	3(仅对6加权)	59	66336067	340013ms

可以看出，小测试集合1，2下复杂度很低，优化作用并不是很大，但是在大测试数据下，还是的优化效果还是很显著的。

IDA* 算法

按照实验指导PDF的伪代码，重构一下复用重构一下代码即可。。但是还是需要使用类似BFS的方式维护一个队列，仍然需要很大的内存开销，可以在本机上运行到1500w节点以上，然而内存还是很大，可以正常完成1,2，在搜索3的过程中虽然节省了一定的内存，跑了更多的节点，但是可惜并没有给出最终的结果。

测试集	步数	结点数	时间
1	24	160	3.51ms
2	12	23	1.40ms
3	-	-	-

P2 X数独问题

问题只需要递归进行回溯搜索即可。

伪代码如下：

```
// backtracking
bool backtracking_search(int matrix[][9], int x, int y)
    bo = false;
    if (x > 9)
        return true;
    for tryNum = 1 to 9 do
        if tryNum in position(x, y) is valid
            matrix[x][y] = tryNum
            find_next(matrix, x, y);
            if (backtracking(matrix, x, y)) return true;
            matrix[x][y] = 0;
     return false;
```

其中find_next若依次按照顺序，则是未使用度启发式优化的版本，如果对矩阵进行遍历找到度最小的位置，则是采用MRV优化的版本，具体信息见`./sudoku/src/readme.md`

优化结果比较

```
.\sudoku.exe sudoku01.txt
Index: 111
Searching takes: 0.00100 seconds

.\sudoku.exe sudoku02.txt
Index: 14853
Searching takes: 0.00700 seconds

.\sudoku.exe sudoku03.txt
Index: 4233934
```

```
Searching takes: 0.95300 seconds

================================

.\sudoku_opt.exe sudoku01.txt
Index: 47
Searching takes: 0.00200 seconds

.\sudoku_opt.exe sudoku02.txt
Index: 107
Searching takes: 0.00300 seconds

.\sudoku_opt.exe sudoku03.txt
Index: 3793
Searching takes: 0.04200 seconds
```

可以发现，优化后的节点数显著减少，成功提升了大量的效率。

思考题

数独问题可以尝试使用这几类局部搜索算法去解决，例如，使用不合法的数字个数作为评价函数，目标则是让这个评价函数为0。然而，使用爬山算法很有可能会使得问题搜索进入瓶颈，从而无法找到解。模拟退火和遗传算法则分别通过概率和解杂交的方式让陷入瓶颈的概率变小。所以我认为是可行的，但是搜索的效率可能并不像求解CSP问题的回溯算法那样简单高效。因为实际上在问题域很大，但其实限制条件很多，并且可行域很小的情况下，如果局部搜索算法的随机取值很可能使得问题的求解变得缓慢。