装 订 线

八数码实验

实验报告

小组成员： 2053410 胡孝博
2052147 雒俊为
1952241 张家瑞
指导老师：武妍
实验时间：2022.4.13

目录

1 实验概述

1.1 实验目的

（1）熟悉人工智能原理中的搜索问题求解过程；

（2）熟悉状态空间的启发式搜索算法的应用，掌握启发式搜索的定义、估价函数和算法过程；

（3）熟悉对八数码问题的建模、求解及编程语言的应用，利用 A*算法求解 8 数码难题，理解求解流程和搜索顺序。

1.2 实验内容及要求

(1)以 8 数码问题为例实现 A*算法的求解程序（编程语言不限），要求设计两种不同的估价函数。

(2)设置相同初始状态和目标状态，针对不同的估价函数，求得问题的解，并比较它们对搜索算法性能的影响，包括扩展节点数、生成节点数和运行时间。画出不同启发函数 h(n)求解 8 数码问题的结果比较表，进行性能分析。

(3)要求界面显示初始状态，目标状态和中间搜索步骤。

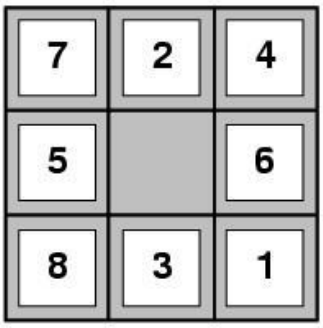
(4)画出图示所示的搜索生成的树，在每个节点显示对应节点的 f(n)值，以显示搜索过程，以红色标注出最终结果所选用的路线。

(5)撰写实验报告，提交源代码（进行注释）、实验报告、汇报 PPT。

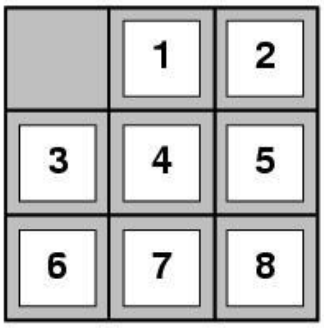
2 实验方案设计

2.1 问题描述

八数码游戏描述为：在 3×3 组成的九宫格棋盘上，摆有八个将牌，每一个将牌都刻有 1-8 八个数码中的某一个数码。棋盘中留有一个空格，允许其周围的某一个将牌向空格移动，这样通过移动将牌就可以不断改变将牌的布局。



Start State



Goal State

该游戏抽象的问题是：将给定一种初始的布局或结构（初始状态）和一个目标的布局（目标状态），如何移动将牌，实现从初始状态到目标状态的转变。

从“问题的形式化”的角度而言，可以将八数码问题形式化为：

（1）状态：状态描述指明 8 个棋子以及空格在棋盘 9 个方格上的分布。

（2）初始状态；任何状态都可能是初始状态。注意要到达任何一个给定的目标，可能的初始状态中恰好只有一半可以作为开始（后续加以讨论）。

（3）后继函数：用来产生通过四个行动(把空位向 Left、Right、Up 或 Down 移动)能够达到的合法状态。

（4）目标测试：用来检测状态是否能匹配目标状态。

（5）路径耗散：每一步的耗散值为 1，因此整个路径的耗散值是路径中的步数。

2.2 总体设计思路

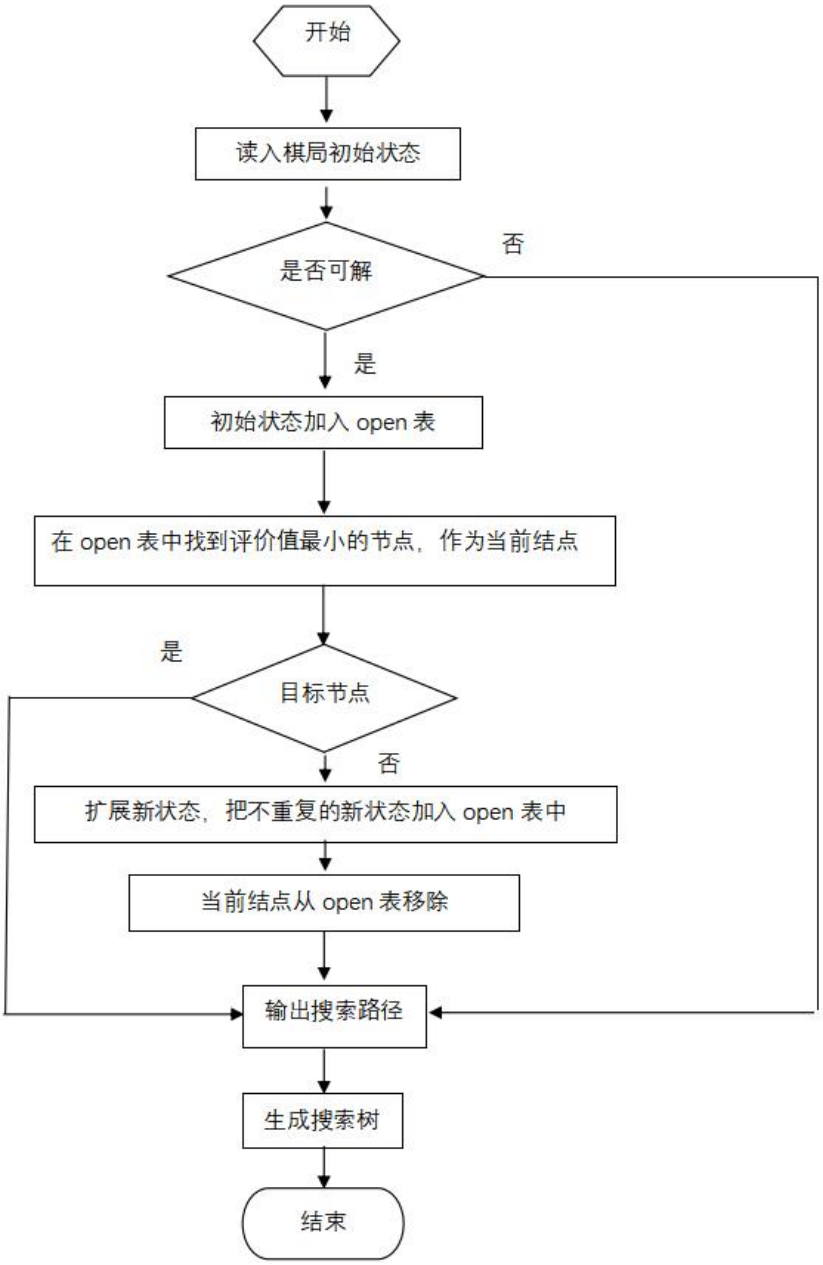
以八数码为代表的滑块问题为 NP 完全问题，因此不存在在最坏情况下明显好于已知搜索算法的方法。本题的最容易想到的搜索解法是暴力搜索，只要尝试将空格周围的数码填入空格尝试解决逆序即可，但这样时间复杂度指数上升，同时，盲目的搜索并不能帮助判断是否有解。对于八数码问题，共有 9!/2=181440 个可达到的状态，考虑到现有计算机的算力，求解时间或许可以接受。但若讲该问题推广，如 15 数码问题有大约 1.3 万亿个状态，用最好的搜索算法求解一个随机的实例的最优解需要几毫秒。24 数码问题的状态数可达 10^{25} 个，求解随机实例的最优解可能需要几个小时。此时，无信息或暴力搜索并不能有效求解。

因此，我们对于八数码问题的实验方案可以概括为：

（1）首先利用数学方法，判断初始矩阵的状态字符串的逆序数与目标矩阵的状态字符串的逆序数是否同奇或者同偶，如果同是奇数或同是偶数就有解，否则无解。

（2）在此基础上我们尝试利用队列进行广度优先搜索，但是盲目搜索会浪费很多时间和空间，所以我们在路径搜索时，会首先选择最有希望的节点，这种搜索称之为 "启发式搜索(Heuristic Search)"我们需要通过启发函数 (Heuristic Function) 计算得到，利用 A*算法进行动态加权，从而尝试寻找到综合最优的解。

2.3 总体架构



2.4 核心算法及基本原理

2.4.1 A*搜索

A*是一种，斯坦福研究所（现为 SRI 国际）的彼得·哈特、尼尔斯·尼尔森和伯特伦·拉斐尔于 1968 年首次发布了该算法。它是 Edsger Dijkstra 1959 年算法的扩展。A*通过使用启发式来指导其搜索，从而获得更好的性能。广泛用于寻路和图形遍历的计算机算法，这是在多个节点之间绘制有效定向路径的过程。由于其性能和准确性，它得到了广泛的应用。

算法是一种静态路网中求解最短路径最有效的直接搜索方法，公式表示为：

$$f(n) = g(n) + h(n)$$

其中：

f(n) 是从初始状态经由状态 n 到目标状态的代价估计，称作估计函数

g(n) 是在状态空间从初始状态到状态 n 的实际代价，即节点深度

h(n) 是从状态 n 到目标状态的最佳路径的估计代价，称作启发函数

当 g(n) 退化为 0, 只计算 h(n) 时,，A* 即退化为 BFS。

当 h(n) 退化为 0, 只计算 g(n) 时,，A* 即退化为 Dijkstra 算法。

A*算法步骤：

（1）建立一个队列，计算初始结点的估价函数 f，并将初始结点入队，设置队列头和尾指针。

（2）取出队列头（队列头指针所指）的结点，如果该结点是目标结点，则输出路径，程序结束。否则对结点进行扩展。

（3）检查扩展出的新结点是否与队列中的结点重复，若与不能再扩展的结点重复（位于队列头指针之前），则将它抛弃；若新结点与待扩展的结点重复（位于队列头指针之后），则比较两个结点的估价函数中 g 的大小，保留较小 g 值的结点。跳至第五步。

（4）如果扩展出的新结点与队列中的结点不重复，则按照它的估价函数 f 大小将它插入队列中的头结点后待扩展结点的适当位置，使它们按从小到大的顺序排列，最后更新队列尾指针。

（5）如果队列头的结点还可以扩展，直接返回第二步。否则将队列头指针指向下一结点，再返回第二步。

A*算法伪代码描述：

```
heuristic_search:
    open = [root]；closed = [ ]；
    define:   f(n) = g(n)+h(n)
    While !open.empty():
        从 open 表中取第一个状态；
        If n =  目的状态
            Then Return（success）；
        生成 n 的所有子状态；
```

If n 没有任何子状态
Then Continue；
For n 的每个子状态 Do
Case 子状态 is not already on open 表 or closed 表:
计算该子状态的估价函数值；将该子状态加到 open 表中
Case 子状态 is already on open 表:
If 该子状态是沿着一条比在 open 表已有的更短路径而到达
Then 记录更短路径走向及其估价函数值；
Case 子状态 is already on closed 表:
If 该子状态是沿着一条比在 closed 表已有的更短路径而到达
Then 将该子状态从 closed 表移到 open 表中
记录更短路径走向及其估价函数值
将 n 放入 closed 表中；根据估价函数值，从小到大重新排列 open 表
Return(failure)；
END

2.4.2 八数码 A*求解的具体分析

（1）估值函数的选择

g(n)：已经移动的步数

h(n)：此状态与目标状态矩阵中相异数字的个数(number of displaced tiles)、此状态与目标状态矩阵中相异数字间的曼哈顿距离(total Manhattan distance)或 0(Dijkstra)。

这样估价函数 f(n)在 g(n)一定的情况下，会或多或少的受距离估计值 h(n)的制约，节点距目标点近，h 值小，f 值相对就小，能保证最短路的搜索向终点的方向进行， 因此 f 是根据需要找到一条最小代价路径的观点来估算节点的。

在启发函数 h(n)的具体选择中，需要考虑哪种 h(n)能够更好地提供趋近目标状态的“方向”，这也是有信息搜索的关键。显然，number of displaced tiles 更大可能为搜索提供一些“误导性信息”，而 total Manhattan distance 比 number of displaced tiles 提供的信息更加明确。这种理论上的思考会在后续 3.4 模块加以验证。

（2）open-closed 表维护

传统的 A*算法在实现时，常通过维护一个递增链表作为 open 表，在每次从表中读数时，可以直接从头部取值。这种方法实现简单，但在生成的新结点加入到 open 表中时，需要遍历原表为结点寻找合适的插入位置。因此在结点插入时，可以根据关键字进行二分查找，时间复杂度 O(logn)。也可以直接以“最小堆”来维护 open 表，时间复杂度 O(logn)，简化了 A*算法中“对 open 表中的全部节点按从小到大的顺序重新进行排序”的维护过程，后续将在 2.6 中进行详细阐释。

（3）无解情况的判断

将矩阵按先行后列展开成线性后，计算初始状态和目标状态的奇偶性（即逆序数）是否一致。

求解逆序数可以通过归并排序来实现，时间负责度 O(nlogn)。因逆序对的数目可能存在平方数个逆序对，因此要想将逆序对数目求解的复杂度降低到 O(nlogn)，就不能对每一个逆序对进行计算。根据归并排序思想，使用归并求解逆序对时，将会在将两个有序数组合并成一个有序数组的过程中，记录逆序对的数目，其他如数组划分和排序过程同归并排序。

2.5 模块设计

2.5.1 状态结点与 open-closed 表

存储结点所有信息的结构体，结点以树形结构组织成搜索树，以链表结构组织成搜索路径。

```
struct Node {
    // Used for currently expanded node in the graph/tree.
    int hn;              // Estimated minimum path cost from "n" to "t"
    int gn;              // Actual path cost from "s" to "n"
    int fn;              // The value of the valuation function at this node: fn = gn + hn
    MATRIX matrix;   // current status
    // Generate a linklist to save a search path
    Node *next;
    Node *prior;
    // Search Tree Structure
    QVector<Node*> child;
    Node *parent;
    int layer;
    bool path;
    int no;
};
```

线性表形式存储的 open-closed 表，两个指针分别指向链表头与链表尾。

```
struct NodeList{
    NodePtr head;
    NodePtr tail;
};
```

2.5.2 八数码求解类

以类的形式，包含求解一个八数码问题所需的全部信息——初始状态、目标状态、open-closed 表、搜索树根节点、启发函数的指针变量、Qt 的 UI 变量，以及求解所涉及的函数声明——初始化、open-closed 表维护、搜索（生成结点与扩展结点）、启发函数、UI 界面中动画演示与结果输出等。

```
class PuzzleSolver : public QWidget {
private:
```

```
    // Termination status (saved as a matrix
    int target_matrix[Matrix_N][Matrix_N];
    // The open table saves all nodes that have been generated but not investigated
    // The closed table records the visited nodes
    NodeListPtr open = NULL, closed = NULL;
    // Root of a search tree
    STree root = NULL;
    // Pointer to different heuristic functions
    int (PuzzleSolver::*h_ptr)(Node *);
    // Pointer to ui
    Ui::PuzzleSolver *ui;

public:
    explicit PuzzleSolver(QWidget *parent = nullptr);
    ~PuzzleSolver();

    // Initilization and check
    NodePtr init();
    NodeListPtr init_open();
    NodeListPtr init_closed();
    int check_target_matrix(NodeListPtr);

    // Comparison between two state (save as node or matrix)
    bool matrix_compare(MATRIX, MATRIX);
    bool node_compare    (NodePtr, NodePtr);

    // Maintain a open table and a closed table
    NodePtr get_open_head(NodeListPtr open);
    bool search_closed(NodeListPtr closed, NodePtr expand_node);
    void add_to_open(NodeListPtr open, NodePtr node);
    void add_to_closed(NodeListPtr closed, NodePtr node);

    // Differnet heuristic functions and unified interface
    int h1_misplaced_tiles(NodePtr node);
    int h2_manhattan_distance(NodePtr node);
    int hval(NodePtr node, int (__thiscall PuzzleSolver::*fptr)(Node *));
```

```
    // Searching process
    void move(int row, int col, direction dir, NodePtr node, NodeListPtr open, NodeListPtr closed);
    void search(int row, int col, NodePtr node, NodeListPtr open, NodeListPtr closed);
    void expand(NodeListPtr open, NodeListPtr closed, NodePtr to_expand);

    // Entrance function!
    NodePtr solve();

    // Functions for UI exhibition
    int ui_move_blocks(NodePtr node);
    int ui_output_path(NodePtr node);
    QLabel *find_Qlabel(int i);
    int find_zero(NodePtr node);

    // Automatic generate and solution
    void shuffle_matrix(int *matrix, inversion inv = Even);

private slots:
    // slot function for buttons
    void on_Solve_clicked();
    void on_Test_clicked();
    void on_showtree_clicked();
};
```

2.5.3 启发函数

以函数指针的形式，为不同的启发函数提供了一致的调用接口。

即在类构造函数或初始化阶段，例如指定：

```
this->h_ptr = &PuzzleSolver::h2_manhattan_distance
```

启发函数的设计直接影响了估计函数的效率，有几种定义方法：

- 当前节点与目标节点差异的度量 => 当前结点与目标节点相比，位置不符的数字个数
- 当前节点与目标节点距离的度量 => 当前结点与目标节点格局位置不符的数字移动到目标节点中对应位置的最短距离之和
- 每一对逆序数字的某倍数
- 位置不符的数字个数的总和+逆序数的三倍

```
int PuzzleSolver::hval(NodePtr node, int (__thiscall PuzzleSolver::*fptr)(Node *)){
    return (*this.*fptr)(node);
```

```
}

int PuzzleSolver::h1_misplaced_tiles(NodePtr node){
    int misplaced_counter = 0;
    for (int i = 0; i < Matrix_N; ++i) {
        for (int j = 0; j < Matrix_N; ++j) {
            if (node->matrix[i][j] != target_matrix[i][j]) {
                ++misplaced_counter;
            }
        }
    }
    return misplaced_counter;
}
int PuzzleSolver::h2_manhattan_distance(NodePtr node){
    int distance_counter = 0;
    // for each position in current matrix
    for (int row = 0; row < Matrix_N; ++row) {
        for (int col = 0; col < Matrix_N; ++col) {
            int key = node->matrix[row][col];
            int correct_row = -1, correct_col = -1;
            // for each position in target matrix
            // calculate the norm between "right posotion" and "current position"
            bool find = 0;
            for (int i = 0; i < Matrix_N && !find; ++i) {
                for (int j = 0; j < Matrix_N && !find; ++j) {
                    if(key == target_matrix[i][j]){
                        correct_row = i;
                        correct_col = j;
                        find = true; // stop loop
                    }
                }
            }
            distance_counter += abs(row - correct_row) + abs(col - correct_col);
        }
    }
    return distance_counter;
}
```

2.5.4 搜索与生成结点

```
/**
 * Try a move of elem"0" in a designated direction
 * @param row,col : position of 0 in current matrix
 */
void PuzzleSolver::move(int row, int col, direction dir, NodePtr node, NodeListPtr open, NodeListPtr
closed){
    NodePtr new_node = new Node;
    if(!new_node){
        QMessageBox::warning(NULL, "warning", "Memory allocate failed!", QMessageBox::Yes,
            QMessageBox::Yes);
        return;
    }

    for (int i = 0; i < Matrix_N; ++i) {
        for (int j = 0; j <Matrix_N; ++j) {
            new_node->matrix[i][j] = node->matrix[i][j];
        }
    }
    switch (dir) {
        case Up:
            new_node->matrix[row][col] = new_node->matrix[row+1][col];
            new_node->matrix[row+1][col] = 0;
            break;
        case Down、Left、Right:
            ......
    }
    new_node->gn = node->gn + 1; // step++
    new_node->prior = node;          // save the current node as a prior
    new_node->next = nullptr;
    new_node->hn = hval(new_node,this->h_ptr); // calculate h(n) for new node
    new_node->fn = new_node->hn + new_node->gn;
    new_node->parent = node;
    new_node->layer = node->layer+1;
    new_node->path = false;
    if (!search_closed(closed, new_node)){ // unvisted
```

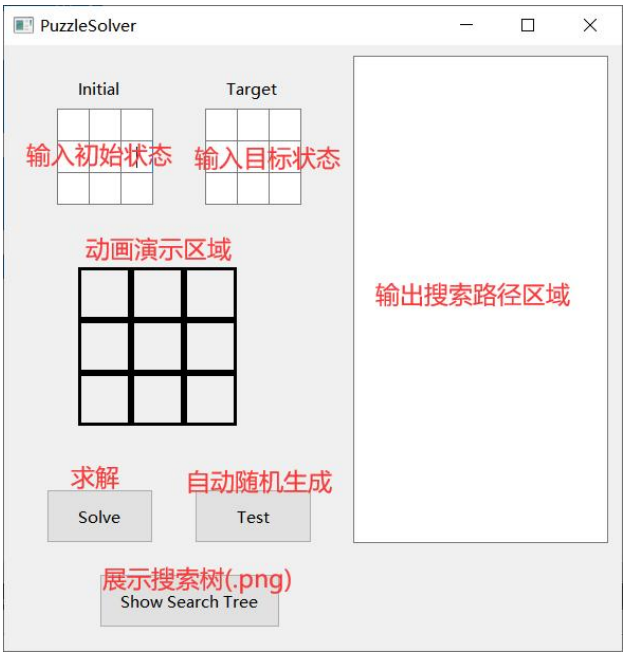
```
            node->child.push_back(new_node);
            add_to_open(open, new_node);
        }
        else { // visited
            delete new_node;
        }
    }

    /**
     * @brief
     * Move and probe according to the position of 0.
     * Perform weight judgment, put the better one or the only one into open table.
     * @param row,col : position of 0 in current matrix
     */
    void PuzzleSolver::search(int row, int col, NodePtr node, NodeListPtr open, NodeListPtr closed){
        if(row!=Matrix_N-1)
            move(row, col, Up, node, open, closed);
        if(row!=0)
            move(row, col, Down, node, open, closed);
        if(col!=Matrix_N-1)
            move(row, col, Left, node, open, closed);
        if(col!=0)
            move(row, col, Right, node, open, closed);
    }

    /**
     * Expand the next state of the current node
     */
    void PuzzleSolver::expand(NodeListPtr open, NodeListPtr closed, NodePtr to_expand){
        int pos, row, col;
        pos = find_zero(to_expand);
        row = pos / Matrix_N;
        col = pos % Matrix_N;
        search(row, col, to_expand, open, closed);
    }
```

2.5.5 求解

循环执行搜索、生成节点、维护 open-closed 表、检查是否到达目标结点这一流程。

```
NodePtr PuzzleSolver::solve(){
    bool succ = 0;    // flag for testing target
    NodePtr tmp = nullptr;
    while (!succ && open->head) {
        tmp = get_open_head(open);
        if (matrix_compare(tmp->matrix,target_matrix)) { // whether extend to target state
            succ = true;
            break;
        }
        add_to_closed(closed, tmp);
        expand(open, closed, tmp);
    }
    if (!succ) { // unsolvable question( won't happen if inversion check has been done
        QMessageBox::critical(this, "Error", "Unsolved for unknown reasons.", QMessageBox::Yes,
                QMessageBox::Yes);
        return nullptr;
    }
    return tmp;
}
```

2.5.5 UI 界面与动画演示



UI 界面的设计基于 Qt，主要为用户提供了初始与目标状态的输入接口、动画演示区域、模式 1 求解、模式 2 随机生成与自动求解、求解完成后的路径打印和搜索树自动生成。

功能的实现主要依赖于信号与槽函数的设计以及递归输出。

递归实现动画演示与路径输出，自根节点向下读取路径，直至叶节点（依赖于之前求解的完成与结果的保存）。

```
int PuzzleSolver::ui_move_blocks(NodePtr node){
    if (node) {
        int pre = ui_move_blocks(node->prior);
        if (pre == -1) { // reaching the final state and get return value as -1
            // initialize blocks
            ui->block1->setText(QString::number(node->matrix[0][0]));
            ......
            ui->block9->setText(QString::number(node->matrix[2][2]));
            int pos0 = find_zero(node);
            find_Qlabel(pos0)->setText(""); // empty the position of 0 (for better demonstration
            return pos0;
        }
        else {
            int pos0 = find_zero(node);
            ......
            QLabel* label_pre = find_Qlabel(pre);
            QLabel* label_cur = find_Qlabel(pos0);

            QElapsedTimer t; // set delay
            t.start();
            while(t.elapsed() < 1000)
                QCoreApplication::processEvents();
            label_pre->setText(QString::number(num)); // fill number
            label_cur->setText("");                   // empty
            return pos0;
        }
    }
    else
        return -1;
}
```

```
int PuzzleSolver::ui_output_path(NodePtr node){
    if (node) {
        int rec = ui_output_path(node->prior);
        QString str;
        for (int i = 0; i < Matrix_N; ++i) {
            for (int j = 0; j < Matrix_N; ++j) {
            str += QString::number(node->matrix[i][j]) + " ";
            if(j == Matrix_N-1)
                str += "\n";
            }
        }
        ui->result->append(str);
        if(rec == -1) // flag of the end of recursion
            ui->result->setText(str);
        node->path = true;
        return 1;
    }
    else
        return -1;
}
```

2.5.6 搜索树的绘制

画出图示所示的搜索生成的树，在每个节点显示对应节点的 f(n)值，以显示搜索过程，以红色标注出最终结果所选用的路线。

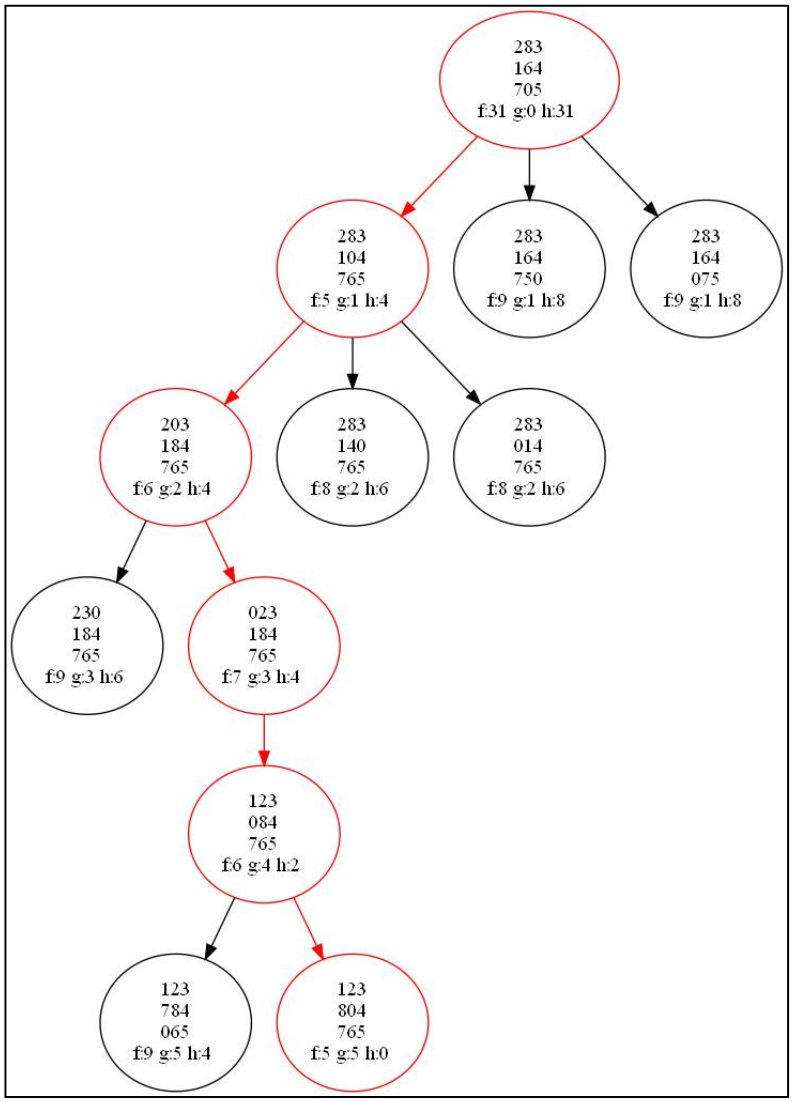
本模块的设计与实现依赖于贝尔实验室设计的一个开源的画图工具 graphviz，通过 dot 语言编写绘图脚本，让 graphviz 解析输入的脚本，分析出其中的点，边以及子图，然后根据属性进行绘制。在具体实现中，我们通过 C++编写了 DrawGraph 类，提供 create_node, create_relation 等函数将上述模块的输出信息转化为相应的 dot 格式，然后通过相应命令转化为 png 格式，最终连接 UI 界面的相应接口(pushbutton)，展现给用户。

类声明及相关函数：

```
class DrawGraph{
public:
    void Start(int mode = 0);
    void Create_Node(int no, std::string lable, bool color = false);
    void Create_Relationship(int, int, int mode = 1, bool color = false);
    void End();
};
```

在实际测试中，graphviz 可以输出大规模的树形图。可参考附件 test.png

例如，对于课堂 PPT 中的八数码案例，可以有如下结果：



2.6 其他创新或优化算法

（1）open-closed 表维护

在实现搜索函数的过程中我们发现，open-close 表的功能主要是在每次循环时确定新生成的结点，和在待扩展的结点中选择一个 f(n)最小的结点。传统的 A*算法在实现时，常通过维护一个递增链表作为 open 表，在每次从表中读数时，可以直接从头部取值。这种方法实现简单，但在生成的新结点加入到 open 表中时，需要遍历原表为结点寻找合适的插入位置。因此在结点插入时，可以根据关键字进行二分查找，时间复杂度 O(logn)。

这一点可以通过小根堆数据结构进行优化。堆排序是利用堆积树（堆）这种数据结构所设计的一种排序算法，它是选择排序的一种，与链表相比，可以利用数组的特点快速定位指定索引的元素。在算法优化阶段，我们以 C++STL 中的 priority_queue 内部堆结构实现了一个优先队列。

在存储时可以通过将 f(n)作为键值保存每个结点，来维护 open 表，时间复杂度 O(logn)；从堆中取值，复杂度 O(1)，简化了 A*算法中“对 open 表中的全部节点按从小到大的顺序重新进行排序”的维护过程。在 closed 表的维护中，由于针对每一个新生成的结点，都需要遍历 closed 表查找是否已存在。考虑到查找频率的需求，我们将基本的线性遍历优化为 hash_map，以键值对映射的形式用于存储键值对（<key,value>）的集合类，将结点的 matrix 信息以及 fn 值糅合作为关键字，避免哈希冲突。

总体来说，我们在保证策略完备性的情况下，利用 STL 容器的 priority_queue，将 open 表的时间复杂度压缩为 O(1)，提高维护速度。利用 hash_map 代替顺序存储，提高索引效率。从而在时间和空间上优化 open-closed 表维护代价。

3 实验过程

3.1 环境说明

操作系统：Win10

开发语言：C++

开发环境：MinGw 7.3.0 64-bit for C++

Qt 5.14.2

Qt Creator 4.11.1

Cmake 3.22.3

核心使用库：Qt 的 Core 模块以及 GUI 模块

Graphviz 3.0.0

3.2 源代码文件清单

8puzzle.pro 项目文件：提供 qmake 关于为应用程序创建 makefile 所需要的细节。

puzzlesolver.h 头文件：与八数码问题求解相关的宏定义、类定义与函数声明。

drawgraph.h 头文件：搜索树的绘制。

utils.h 头文件：搜索树的定义，以及项目中用到的一些通用函数（如求逆序数）的声明。

main.cpp 源文件：main 函数，程序入口。

heuristic_fun.cpp 源文件：实现两种启发函数（错位数与曼哈顿距离），以及调用的统一接口。

puzzlesolver.cpp 源文件：八数码问题求解的函数实现（如初始化、表维护、搜索、扩展等）。

test.cpp 源文件：自动生成与自动求解模式。随机生成初始状态与目标状态，然后求解。

ui.cpp 源文件：与 UI 界面中数据输入、动画演示、路径生成等相关函数的实现。

drawgraph.cpp 源文件：搜索树的绘制函数实现。

puzzlesolver.ui UI 文件：程序运行的整个 UI 界面。

3.3 主要函数清单

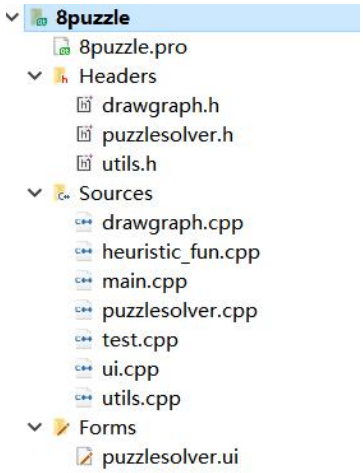
```
// Initilization
    NodePtr init();
    NodeListPtr init_open();
    NodeListPtr init_closed();
    int check_target_matrix(NodeListPtr);
// Comparison between two states (save as node or matrix)
    bool matrix_compare(MATRIX, MATRIX);
    bool node_compare   (NodePtr, NodePtr);
// Maintain a open table and a closed table
    NodePtr get_open_head(NodeListPtr open);
```

```
        bool search_closed(NodeListPtr closed, NodePtr expand_node);
        void add_to_open(NodeListPtr open, NodePtr node);
        void add_to_closed(NodeListPtr closed, NodePtr node);
// Differnet heuristic functions and unified interface
        int h1_misplaced_tiles(NodePtr node);
        int h2_manhattan_distance(NodePtr node);
        int hval(NodePtr node, int (__thiscall PuzzleSolver::*fptr)(Node *));
// Searching process
        void move(int row, int col, direction dir, NodePtr node, NodeListPtr open, NodeListPtr closed);
        void search(int row, int col, NodePtr node, NodeListPtr open, NodeListPtr closed);
        void expand(NodeListPtr open, NodeListPtr closed, NodePtr to_expand);
// Entrance function!
        NodePtr solve();
// Functions for UI exhibition
        int ui_move_blocks(NodePtr node);
        int ui_output_path(NodePtr node);
        QLabel *find_Qlabel(int i);
        int find_zero(NodePtr node);
// Automatic generate and solution
        void shuffle_matrix(int *matrix, inversion inv = Even);
// get inversion number (based on merge sort)
        int get_inversion_number(const int * const arr,const int len);
// mermory recovery
        void delete_linklist(NodeListPtr list);
// debug
        void print_matrix(MATRIX matrix);
        std::string print_matrix_tostr(MATRIX matrix);
        void print_closed(NodeListPtr closed);
// draw the search tree
        void bfs(NodePtr);
```

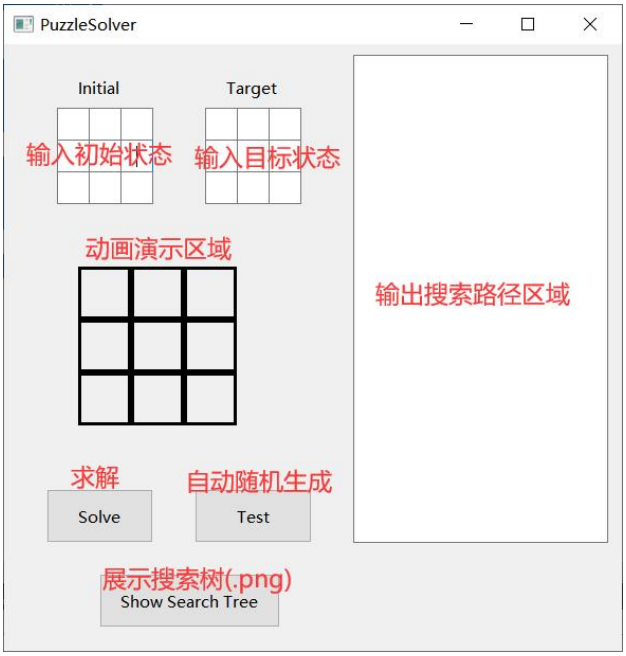
3.3 实验结果展示

（1）双击 code 目录下 8puzzle.pro 文件，进入 Qt Ctreator 配置界面。

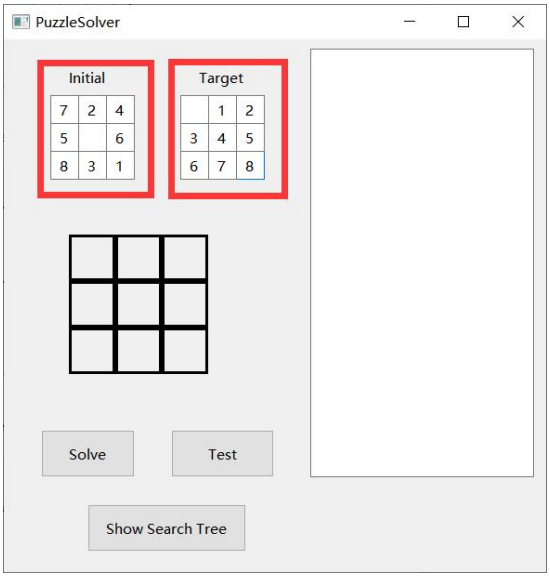
（2）目录结构



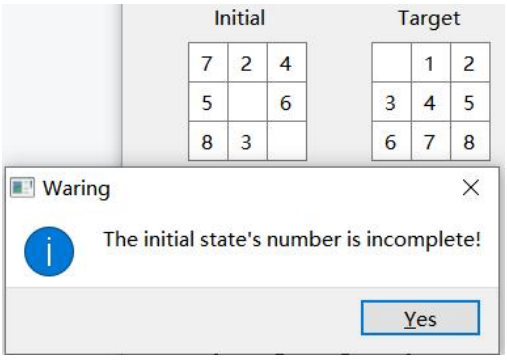
（3）运行，呈现 UI 界面如下



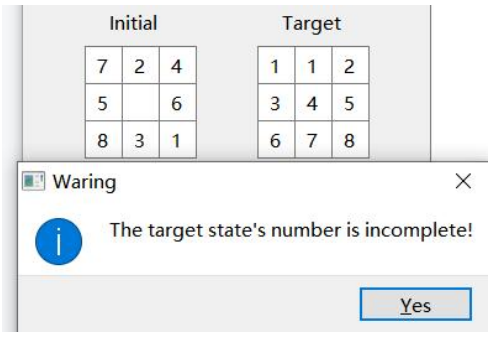
（4）求解：在两个框中分别随机输入数字 1-8（空出数字 0 的位置即可），例如（课本 P91 例子）



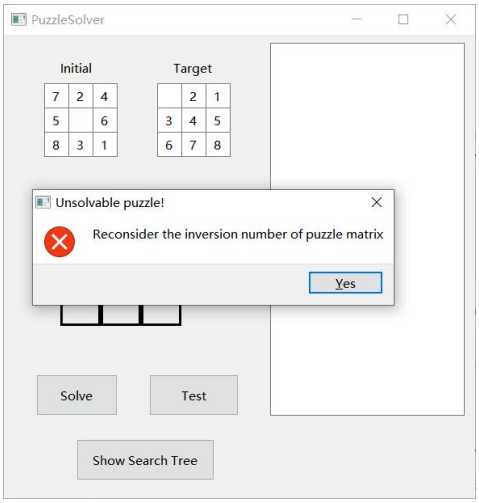
程序自动进行初始状态的检查



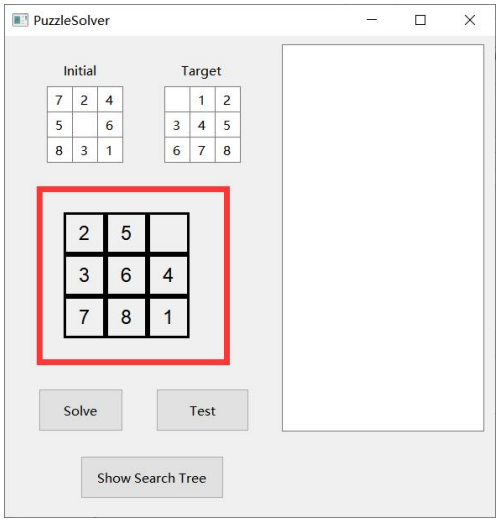
然后进行目标状态的检查



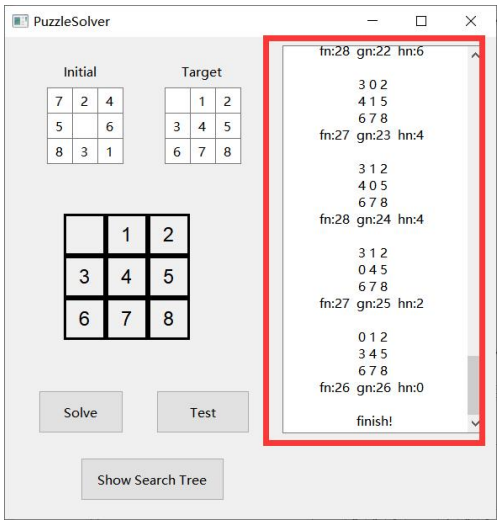
点击 Solve 按钮，程序会检查是否有解（基于逆序数），若无解，给出提示，要求用户调整。



若有解，自动求解，然后在下框内以动画演示求解过程。

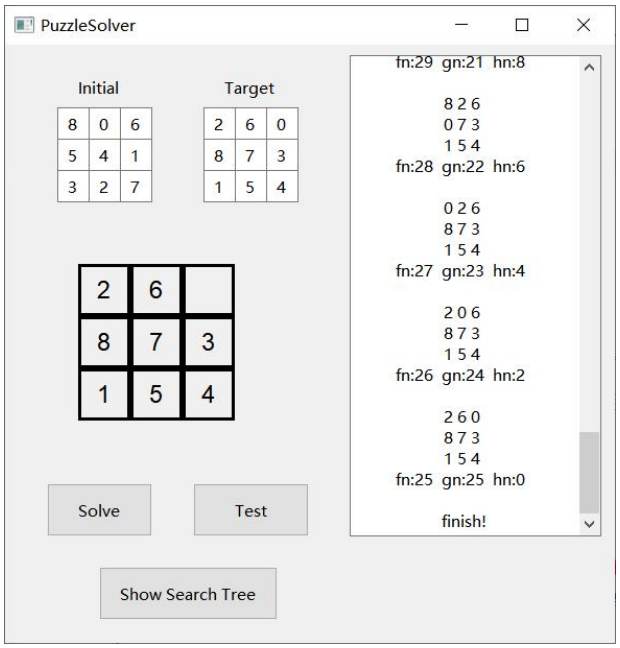


演示结束，右侧框内展示搜索的路径



装 订 线

（5）随机生成：点击 Test 按钮，随机生成一组测试的初始状态与目标转换，然后自动求解，演示流程同上。



3.4 实验结论

实验中使用了错误置位的个数（h1）和曼哈顿距离（h2）两种估价函数。单次测试结果如下:

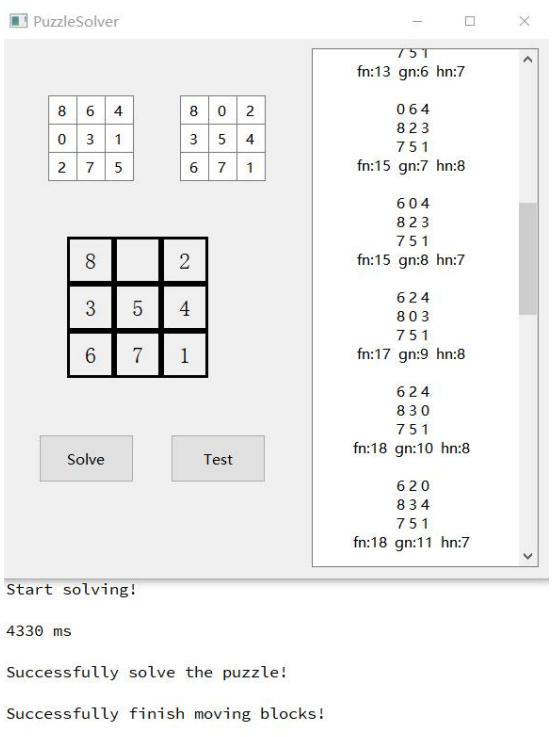


图 1 以错误置位的格数做估价函数

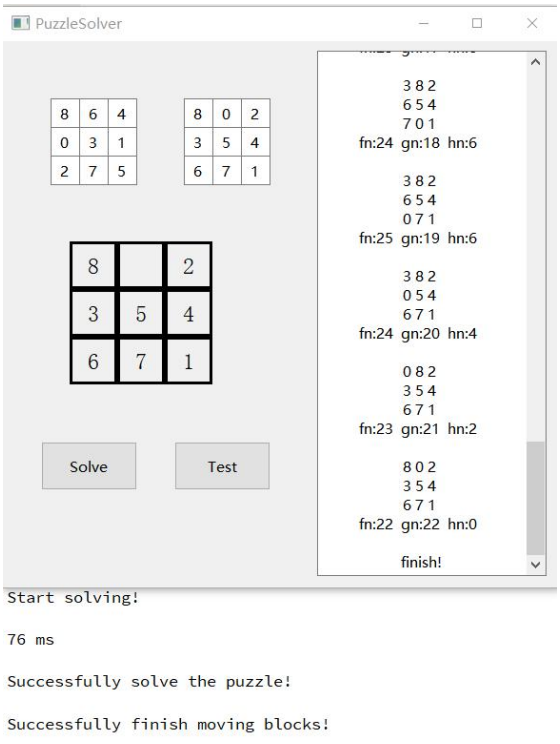


图 2 以曼哈顿距离为估价函数

从运行结果可以看出，对于同一个 Puzzle，采用 h2 作为估价函数时，运行速度和搜索的状态数远小于 h1，效率更高。

	启发函数 f(n)	
	h1	h2
初始状态		
目标状态		
最优解	1 8 2 4 0 7 3 5 6 fn:6 gn:0 hn:6 1 8 2 4 5 7 3 0 6 fn:7 gn:1 hn:6 1 8 2 4 5 7 3 6 0 fn:8 gn:2 hn:6 ..... 5 8 2 3 0 7 6 4 1 fn:23 gn:20 hn:3 5 8 2 0 3 7 6 4 1 fn:23 gn:21 hn:2 5 8 2 6 3 7 0 4 1 fn:22 gn:22 hn:0	1 8 2 4 0 7 3 5 6 fn:33 gn:0 hn:33 1 8 2 4 5 7 3 0 6 fn:15 gn:1 hn:14 1 8 2 4 5 7 3 6 0 fn:16 gn:2 hn:14 ..... 5 8 2 3 0 7 6 4 1 fn:24 gn:20 hn:4 5 8 2 0 3 7 6 4 1 fn:23 gn:21 hn:2 5 8 2 6 3 7 0 4 1 fn:22 gn:22 hn:0
扩展结点数	22	22
生成结点数	18285	1602
运行时间	2818 ms	11ms

4 总结

4.1 实验中存在的问题及解决方案

初次学习使用桌面应用端 Qt 实现图形化界面，小组成员间欠缺配合。在之后的合作中积极沟通配合，分享经验，解决环境配置和运行等实验基础问题。代码编写过程中，对代码的理解占用时间较多。在规范注释的书写后，代码编写速度明显加快。

4.2 心得体会

本次实验让我们对基于搜索的智能体有了一定的了解，通过实际训练对 A*算法有了进一步的认识，也在查阅资料的过程中学习了更多该算法的历史和发展过程。

八数码问题作为曾风靡一时的游戏，学习时已经对于规则比较熟悉，非常适合作为第一个练手的项目，在之后的学习中也应尽量将知识和已经掌握的技能联系起来。

4.3 后续改进方向

实验中为了便与比较不同估价函数的效率，统计求解函数的运行时间，没有做到实时输出打印搜索过程，在之后的项目中是一个改进的方向。除此之外，输入谜题的方式较为单一和麻烦，可以增添文件读入等模式来批量对更多的谜题统一求解。

4.2 总结

本次实验中使用 C++编程语言实现了优化后的 A*算法求解八数码问题，通过 Qt 框架实现图形化界面，打印问题解和运行时间，借助 graphviz 打印搜索树等结果展示内容，加深了对 A*算法、估价函数的理解。了解了更多 A*的背景知识和优化问题，加深了对启发式搜索的理解，锻炼了团队协作能力。

参考文献

[1] 余博文.基于 A*算法的最短路径搜索的优化与研究[J].数码世界,2019(08):35-38.

[2] 孙玉昕,章瑾.利用堆排序优化路径搜索效率的分析[J].武汉工程大学学报,2013,35(06):50-54.

[3] Korf R E, Reid M. Complexity analysis of admissible heuristic search[C]//AAAI/IAAI. 1998: 305-310.

[4] Grapgviz documentation . https://www.graphviz.org/documentation/

[5] Qt Reference Pages. https://doc.qt.io/qt-6/reference-overview.html

装 订 线

成员分工及自评

胡孝博：负责本次实验的代码实现、总体架构设计、技术路线制定等部分。包括数据结构的设计、A*搜索算法的 C++实现、Qt 界面设计与搜索过程可视化、搜索路径输出、graphviz 模块与搜索树绘制等。根据代码实现与模块设计，相应地完成了实验报告中 2.2，2.3，2.4，2.5 以及 3 的撰写。通过本次实验，我与队友分工协作，加深了对启发式搜索、A*算法、估价函数的理解，提高了编程实践能力与算法基础，锻炼了自主探究能力。

雒俊为：在本次实验中我主要负责了算法优化、启发函数比较和报告撰写部分(主要包括 1，2.6，3.4，4 等)，实验总体设计效果符合要求，结果正确，解决问题时做出了许多探索，将知识很好地运用到了实践中。

张家瑞：本次八数码实验的解决过程中，主要负责算法创新、报告撰写(主要包括 1, 2.1，2.2, 2.4, 2.6 等)、PPT 制作工作，学习了一些 Qt 操作及应用，在队友的帮助下，理解学习 8 数码问题的解决原理和优化角度，对问题涉及的知识有了更全面的了解和思考。