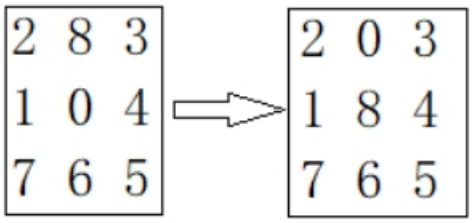
第二次作业：A 星搜索求解八数码问题

1953194 庞鹏瞩

问题引入

在 3×3 的棋盘上，摆有八个棋子，每个棋子上标有 1 至 8 的某一数字。棋盘中留有一个空格，空格用 0 来表示。空格周围的棋子可以移到空格中。要求解的问题是：给出一种初始布局（初始状态）和目标布局（设目标状态为 123456780），找到一种最少步骤的移动方法，实现从初始布局到目标布局的转变。

移动的方法如下图所示：



（这个图中将数字 8 移动到了空格上）

问题分析

初始的布局可以看作是起始状态，目标布局则是目标状态，每一次的移动是一次的状态改变。这样的变化没有太多的规律可循，且由于数据规模小，状态数量有限，所以我们可以考虑用搜索的方法进行求解。

搜索方法

一、普通 BFS

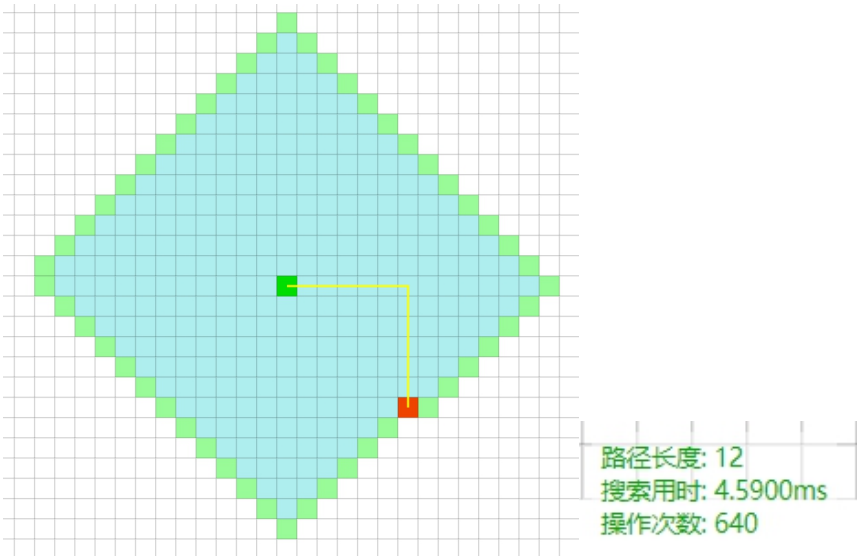
容易想到这个题能够使用 BFS 的方法，当我们在搜索的过程中遇到了目标状态的时候，它所对应的搜索深度即为我们需要的答案数。

这种方法的原理比较简单，实现的难度较低。缺点在于它的算法复杂度较大，需要巧妙的方法来记录状态，否则运行时间较长。而且由于宽搜的特点，它在记录最短路径时候不太方便，这在其它几种基于 BFS 的搜索方法上也有所体现。

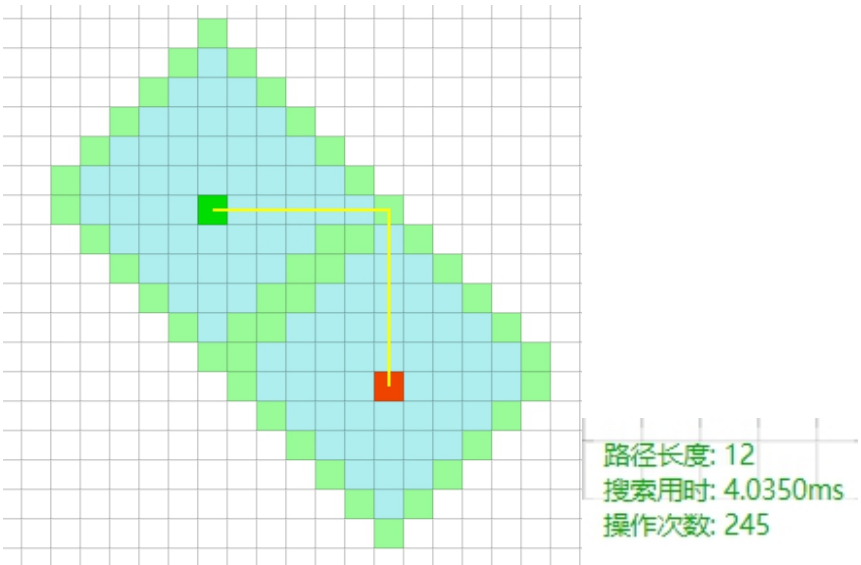
二、双向 BFS

双向 BFS 是对于普通的 BFS 的优化。这个问题的一个特点在于，我们知道初始状态，同时我们也知道目标状态，且步骤是双向的，即从初始状态到目标状态和目标状态到初始状态的最短路径数是一样的。基于这样的特点，我们可以通过双向搜索的方法降低搜索算法的复杂度。

举一个例子：假如我们没有使用状态的记录，搜索树的分支因子为 q，且最后的最短路径为 n，那么使用普通的 BFS 的话，拓展的总的节点的数量为 1+q+q2+…+qn；而用双向 BFS，拓展的总的节点的数量为 2(1+q+q2+…+qn/2)，是远小于普通 BFS 的。



（普通 BFS）



（双向 BFS）

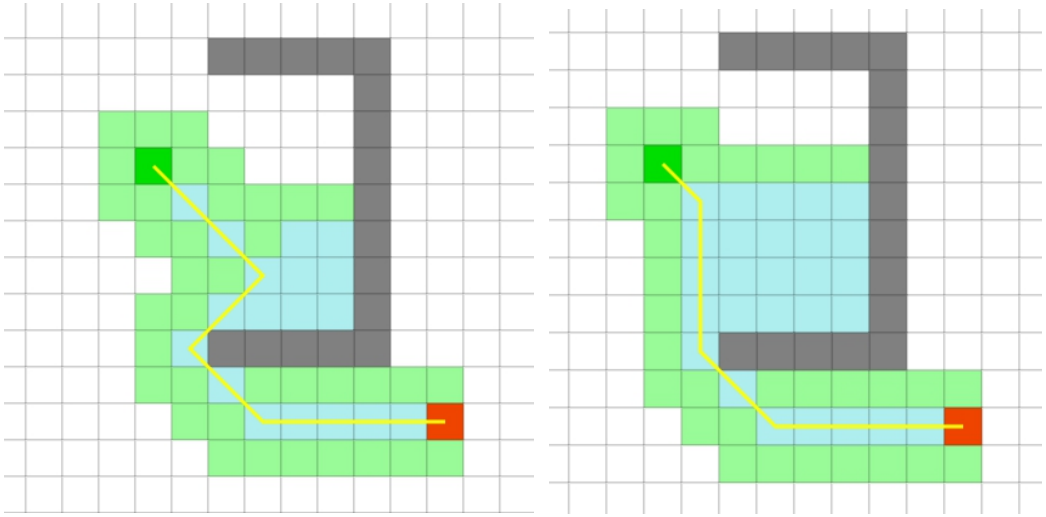
三、A*

A*是一种启发式搜索。启发式搜索与普通的搜索不同的地方在于，它有一个状态的估价函数 f(n)，通过这个函数，我们可以选择优先拓展某一个节点，从而达到拓展更少的节点来搜索到目标状态的效果。

当我们每一次都选择估价函数 f(n)最优秀的节点进行拓展，这样的算法是贪婪最佳优先算法。它可以以最快的速度找到目标状态，但是它的弊端在于它没法求解一个最短的路径，所以它在这个问题当中失去了用武之地。

而 A*，就可以利用估价函数 f(n)，兼顾速度与正确性来找到答案。

A*算法和贪婪最佳优先算法的区别在于，后者是优先拓展 f(n)最好的节点，而 A*则需要记录当前的深度 g(n)，也就是从初始状态到目前状态的步骤数量，然后计算 v(n) = g(n) + f(n)，优先拓展 v(n)最好的节点。这就兼顾了估价函数以及路径最优。当 f(n)为 0 时，即当前的状态已经是目标状态了，那么它的 g(n)即为我们所需的答案。



上图左为贪婪最佳优先搜索，右为 A*搜索，可以看到前者所拓展的节点的个数要少一些，但是最后得到的路径并不是最短的。A*算法则得到的最短的合法路径，且其复杂度相对于盲目搜索优秀太多。

A*搜索算法的核心代码如下：

```
int A_star()
{
    node temp;
    temp.hash = getHash();    //得到当前局面的一个哈希值
    temp.val = dif();         //优先队列以其为参数进行排序
    temp.dep = 0;             //当前节点的深度
```

```
        q.push(temp);              //放入优先队列中
        while (!q.empty())
        {
            node now = q.top(); q.pop();
            m[now.hash] = 1;       //将当前局面放入 map
            hashToArray(now.hash);//由哈希值逆推局面
            if (!dif()) return now.val;//当前局面为目标局面

            //找到 0 所在的位置
            int pos;
            for (int i = 0; i < 9; i ++)
                if (!a[i])
                    pos = i;
            if (pos > 2)//向上移动
            {
                swap (a[pos], a[pos - 3]);  //进行位置改变
                temp.hash = getHash();      //位置改变之后的哈希值
                if (!m[temp.hash])          //初始化新节点，放入 map
                {
                    temp.val = now.dep + 1 + dif();
                    temp.dep = now.dep + 1;
                    q.push(temp);
                }
                swap (a[pos], a[pos - 3]);  //变化为原来状态
            }
            //其余三个方向与向上相同处理，此处不再展示
            if (pos < 6){}
            if (pos % 3){}
            if (pos % 3 != 2){}
        }
    }
```

四、IDA*

IDA*是基于迭代加深搜索，结合启发式函数 f(n)的搜索算法。与基于 BFS 的 A*算法不同，IDA*基于迭代加深，也就是基于 DFS，在这个问题中体现了它一些独特的优势，是应对这个问题的最为合适的算法。

首先我们要从迭代加深搜索出发，迭代加深搜索是对于限制深度的 DFS 的一种应用，将限制深度从 0 开始增大到 n，直到可以得到答案为止。当深度 n-1 没有搜索到，n 能搜索到的时候，就代表着 n 即为我们所需要求解的答案。可以说迭代加深搜索兼顾了 DFS 和 BFS 的特点。

最开始学习迭代加深搜索的时候会有一个问题：在搜索到 n 之前，限制深度为 0 到 n-1 的搜索是否造成算法的复杂度很高？事实不是这样。因为搜索的复杂度随着深度的增加是指数上升的，所以前 n-1 层所拓展的节点之和不一定是比单独第 n 层多的。

在此基础上，我们加入估价函数，便是启发式的 IDA*搜索。我的理解是这样的：IDA*是使用估价函数来对于迭代加深搜索进行启发式的剪枝的搜索方法。它的剪枝原理是这样的：加入当前限定的深度为 n，当前状态的深度为 dep，当前状态的估价函数为 f(x)，假如说有 f(x) > n - dep，代表着在接下来的 n-dep 层的搜索中是达到不了目标转态的。所以可以直接进行剪枝。

IDA*相比于其他算法有两个显然的优势，这些都是 DFS 的搜索方式所带来的。一个优势是答案所需要的步骤的记录，在 DFS 当中，每一次操作的记录，直接通过答案的入栈出栈便可以解决（最后要做成动画就需要记录每一步的操作，所以用的是 IDA*的算法实现的），而 BFS 需要每一个结点都记录其父亲结点以及转移的操作；第二个优势是不需要记录状态，只需要添加一个最优化剪枝，记录上一次的操作，不要操作回去就好。

IDA*的运行速度也相当可观，以 luogu p1379 八数码难题为例：

（A*算法）

（IDA*算法）

IDA*搜索算法的核心代码如下：

```
bool IDA_star(int dep, int pre)
{
    //dep 由 n 递减到 0
    if (dep == 0)
    {
        //flag 为 true，表示到达了目标状态
        bool flag = true;
        for (int i = 0; i < 9; i ++)
            if (a[i] != tar[i])
                flag = false;
        //当找到最小方案时，输出方案数
```

```
            if (flag)
            {
                for (int i = ans; i >= 1; i --)
                    printf ("move %d to %d\n", re[i].from + 1, re[i].to +
1);
            }
            return flag;
        }

        //dif 为当前状态的估价函数
        //若大于 dep，则一定没法到达目标状态，进行剪枝
        int diff = dif();
        if (diff > dep) return false;

        //找到 0 所在的位置
        int pos;
        for (int i = 0; i < 9; i ++)
            if (!a[i])
                pos = i;

        //向四个方向操作，上下左右分别编号为 1234
        bool flag = false;
        if (pos > 2 && pre != 2)//向上移动，若上一次操作为向下则跳过
        {
            swap(a[pos], a[pos - 3]);//进行交换
            //记录这一次的操作
            re[dep].from = pos - 3;
            re[dep].to = pos;
            //由于返回函数是一个 bool 值，找到一个 true 就算成功，所以用或运算
            //传入的是限定深度值，所以减小，这一次向上，操作编号为 1
            flag |= IDA_star(dep - 1, 1);
            swap(a[pos], a[pos - 3]);//交换为原来状态
        }
        if (flag) return flag;
        //其余方向与向上处理方式一样，这里忽略
        if (pos < 6 && pre != 1){}//向下移动
        if (flag) return flag;
        if (pos % 3 && pre != 4){}//向左移动
        if (flag) return flag;
        if (pos % 3 < 2 && pre != 3){}//向右移动
        if (flag) return flag;
        return false;
    }
```

状态记录

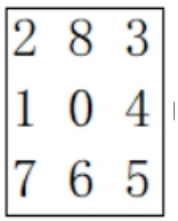
状态记录是对于基于 BFS 算法而言的，也就是除了 IDA*以外的所有的算法都需要把之前搜索得到的状态记录下来，防止重复不断的访问。

下面是三种在这个问题中用得到的记录状态的方法：

一、平衡树

平衡树是我们上学期所学到的一种特殊的排序二叉树，可以在 O(log n)的复杂度的条件下实现插入和查找操作。

每一次我们将当前的状态转化为数字，例如：



我们便直接将其转化为 283104765，然后在平衡树中查找是否有这样的一个数字。若存在，那么这个状态是被访问过的，我们将其删除。若不存在，我们便将这个数字插入到平衡树中，然后把这个状态放入队列进行拓展。

当然，平衡树的手动代码实现有一定的复杂度，所以我们可以直接用 c++ stl 中的 set 或者 map，这样会更加方便。当然它们的底层实现实质上是 RBtree，实际上也是平衡树的思想，复杂度为 O(log n)。

二、哈希表

哈希表也是我们在上学期数据结构课上学到的一种数据结构。按照前面的方法，将这个九位数(0 开头的话是八位数)按照散列函数映射到对应的存储位置上，注意冲突的处理。查找的时候按照构造散列函数和处理冲突的办法进行查找即可。

c++ stl 中的 unsorted map 底层就是用的哈希表的方式实现，比普通的 map 更快。

三、康托展开

康托展开是一个全排列到一个自然数的双射，是计算当前排列在所有由小到

大全排列中的顺序，因此是可逆的。

在八数码问题中，如果我们将空格的位置看作为 9，那么实际上每一个状态都可以用一个全排列数来表示，一共就有：9!个状态，这个数值等于 362800。也就是说如果我们对于遇到的每一个状态进行康托展开，就可以得到一个小于 362800 的数，这代表着我们可以直接用数组 O(1)求得当前状态是否访问过。相比于平衡树的方法会更快，比哈希表的方法更直观。

康托展开的公式：

$$X = a_n(n-1)! + a_{n-1}(n-2)! + \cdots + a_1 \cdot 0!$$

其中，a_i 表示原数的第 i 位在当前未出现的元素中是排在第几个。

这个公式的推导如下：

假设我们要计算在五个数的全排列中，34152 的康托展开值，也就是所以比 34152 小的全排列数的个数。从头开始计算，第一位为 3，那么以 1, 2 为第一位的全排列一定都在它的前面，$a_5 = 2$，答案数加上 2 * 4!；第二位是 4，由于 3 已经出现过了，那么 $a_4 = 2$，答案数加上 2 * 3!；以此类推，a3 = 0, a2 = 1,所以最后的答案数为 2 * 24 + 2 * 6 + 1 * 1 = 61。

用这个构造的方法，也可以从对应的康托展开值得到原有的排列。同样是从前往后，用 X 整除(i - 1)!得到 a_i 的值，然后 X 对(i - 1)!取模后用相同方法求解 a_{i-1} 的值。以此来还原原来的排列。

康托展开的核心代码如下：

```
const int factorial[] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880};
//预处理阶乘的值
inline int cantor(int *p)//康托展开
{
    int ans = 0;
    for (int i = 0; i < 9; i ++)
    {
        int cnt = 0;//cnt 即为 a[9 - i]的值
        for (int j = i + 1; j < 9; j ++)
            if (*(p + i) > *(p + j))
                ++ cnt;
        ans += cnt * factorial[8 - i];
    }
    return ans;
}
```

```
//逆康托展开
inline void decantor(int x)
{
    //判断前面是否出现过
    bool vis[10];
    for (int i = 1; i <= 9; i ++)
        vis[i] = 0;
    //还原操作
    for (int i = 0; i < 9; i ++)
    {
        a[i] = x / factorial[8 - i];
        //这里的a[i]不是真的a[i]，只是用来暂存整除的值
        for (int j = 1; j <= 9; j ++)
        {
            //从小到大枚举，如果前面出现过就跳过
            if (vis[j]) continue;
            if (!a[i])
            {
                a[i] = j;      //这才是真的a[i]的值
                vis[j] = 1;    //标记出现
                break;
            }
            a[i] --;
        }
        //x进行取模操作
        x %= factorial[8 - i];
    }
    return;
}
```

估价函数

估价函数是A*和IDA*以及其它的启发式搜索算法当中很重要的一环。一个好的估价函数可以让搜索得到很大的优化。

在这个问题中，容易想到两种估价函数：一是以当前与目标位置不同的数码的个数；二是每一个数码与其目标位置的曼哈顿距离之和。

总体来说，第二种估价函数比第一种估价函数更加优秀。虽然在A*算法中，这两种估价函数在实际的数据验证中没有太大的区别，但是在IDA*算法中，第二种估价函数在理论上就比第一种估价函数优秀。因为IDA*是根据深度和估价

函数的大小比较来进行剪枝的，而每一次操作最多使这两种估价函数值改变 1。在第一种方法中，估价函数的最大值只能为 8，而第二种里可以比 8 更大，可以完成 IDA*中更加优秀的剪枝，因此效率也会更高。

下面是两种不同的估价函数的实现方法：

第一种：当前与目标位置不同的数码的个数

```
inline int dif()
{
    int cnt = 0;//记录与目标位置不同的数码个数
    for (int i = 0; i < 9; i ++)
        if (tar[i] && tar[i] != a[i])
            ++ cnt;
    return cnt;
}
```

第二种：每一个数码与其目标位置的曼哈顿距离之和

```
void init_dis()//初始化两个位置的 dis
{
    const int inf = 1e9;
    for (int i = 0; i < 16; i ++)
        for (int j = 0; j < 16; j ++)
            if (i != j)
                dis[i][j] = inf;//初始化距离为 inf
    //相邻的点建长度为 1 的边
    for (int i = 0; i < 16; i ++)
        if (i % 4 < 3)
            dis[i][i + 1] = dis[i + 1][i] = 1;
    for (int i = 0; i < 12; i ++)
        dis[i][i + 4] = dis[i + 4][i] = 1;
    //floyd 算法求多源最短路
    for (int k = 0; k < 16; k ++)
        for (int i = 0; i < 16; i ++)
            for (int j = 0; j < 16; j ++)
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
    return;
}

inline int dif()//估价函数
{
    int cnt = 0;
    for (int i = 0; i < 16; i ++)
        if (a[i])
            cnt += dis[i][a[i] - 1];//哈密顿距离之和
```

```
    return cnt;
}
```

接下来我们测试一下，这两种不同的估价函数对于性能的影响。8 数码问题的数据范围比较小，区分度不是十分明显，所以我们增大问题的规模，将每一条边的长度扩大为 4，这样就一共有 15 个数码。总共的状态数 16!，约等于 2.1*(10^13)，用其它的普通搜索方法无法在短时间内得到正确的答案。

测试的时候，我使用的是 IDA*算法，用两个不同的估价函数的结果：

如果使用的是前一种估价函数，那么无法在短时间内得到答案。我们看一下它在迭代到哪一个深度的时候会卡住：

```
0 1 2 3
15 12 13 4
11 14 8 10
9 7 6 5
In which depth the time is longer than 10s: 36
```

可以看到，在第 36 层的时候，这种方法已经很难短时间得到结果。

而我们使用第二种估价函数：

```
0 1 2 3
15 13 12 4
11 14 8 10
9 7 6 5
46
time: 876ms
--------------------------------

Process exited after 12.25 seconds with return value 0
请按任意键继续. . .
```

在不到 1s 的时间内得到了答案。那么我们同样测试一下它的搜索极限（为得到极限的情况，这一组数据是没有合法解的）：

```
15 14 13 12
11 10 9 8
7 6 5 4
3 2 1 0
In which depth the time is longer than 10s: 64
```

可以看到，只是更换了估价函数，这个程度的可以搜索的深度增加了近 30 层，快到第一种方法的两倍了。且考虑到随着深度的增加，复杂度是指数级的升高，所以可见一个优美的估价函数对于问题求解复杂度的影响有多大。

在这个问题中，造成这样情况的原因是什么？我们从估价函数的值来进行分析：

使用测试数据的时候，我们看到，在第一个估价函数中，得到的估价函数值

为 15，这代表着每一个位置的数都与其目标位置不一样，并且这也已经是这个估价函数的极限了。而在第二个估价函数中，这个初始的估价函数值为 32，这意味着，在迭代加深的过程中，前 32 层都可以直接通过启发式剪枝来跳过。而第一种方法的搜索极限才仅仅 36 层，这就是一个正确且优美从估价函数带来的影响。

无解情况

在开始对于算法进行验证的时候，常常会遇到一种情况：随机输入一组数据，然后看似没有问题的代码却一直没法得到正确的结果，或者是一直在运行。这是因为不是所有的起始状态都可以转移到目标状态。

以 123405678 为例，在这个序列中，若我们将 0 去掉，逆序对的个数为 0。这个序列可以进行几种变换：0 与 4 交换，与 5 交换，与 2 交换，与 7 交换。其中 4 和 5 与 0 在同一行，在它们进行交换后，将 0 去掉的序列没有变化，逆序对的个数同样没有变化。而对于 2 和 7 这样在同一列的数，在进行交换后，相当于 2 往后移动了两位，或者 7 往前移动了两位。这样的情况下，逆序对的个数增加了 2。由此可见，当我们对于这个八数码进行操作的时候，在去掉 0 之后，这个序列的逆序对个数的奇偶性不会发生变化。

因此我们可以得到结论，在起始状态和目标状态去掉 0 之后，若它们的逆序对数对于 2 同余，则这个问题可解，否则这个问题没有解。例如 012345678 就永远不可能转化为 021345678 的状态。所以在求解前，我们应当先对于无解情况进行判断。

假如我们把问题的规模扩大，到 4*4 的问题，5*5 的问题甚至更大，我们可以用同样的方法进行无解情况判断吗？

在 4*4 的情况中，向左和向右对于去掉 0 之后的序列的逆序对数没有一点影响，而向上和向下，假设 0 的位置在第 i 行第 j 列，则是把 i - 1 行第 j 列或者 i + 1 行第 j 列的数移动到这个位置。这就相当于把一个数向前或者向后移动了三位。这样的话，与 3*3 不同，在去掉 0 之后的奇偶性有了改变。那么这代表对于所有情况都有解？

事实并不是这样。虽然奇偶性改变了，但是要从初始状态变化到目标状态的

话，对于单个数码，它左右移动不会改变奇偶性，它上下移动一次会改变奇偶性。对于初始状态的每一个数码，我们计算它从当前行数到目标位置的行数之差，便是这个数码的移动对于逆序对奇偶性的贡献。再将所有的数码对应的贡献进行求和，加上初始状态的逆序对，便可以判断是否有合法解。

x*x 的情况，若 x 为奇数，则按照 3*3 方法处理，否则是 4*4 的方法。

4*4 情况的判断代码如下：

```
bool judge()//判断是否合法
{
    for(int i = 1; i <= 4; i ++)
        for(int j = 1; j <= 4; j ++)
            b[(i - 1) * 4 + j] = a[i][j];
    int cnt = 0;
    //统计逆序对数
    for(int i = 1; i <= 16; i ++)
        for(int j = i + 1; j <= 16; j ++)
            if(b[i] && b[j] && b[j] < b[i])
                ++ cnt;
    //数码上下移动对奇偶性的贡献
    for(int i = 1; i <= 4; i ++)
        for(int j = 1; j <= 4; j ++)
            if(a[i][j])
                cnt += (a[i][j] - 1) / 4 + 1 + i;
    return (cnt & 1);
}
```

图形化界面

由于作业要求，所以需要做一个动态演示。由于自己的前端能力很弱，加上为了高效率求解，所以用了 c++的 easyX 库来进行绘制。运行前需要安装 easyX 图形库，并且似乎其只支持 VS 系列的产品，所以在 VS2019 的环境下完成代码的编写和调试。

首先我们以下图为目标状态进行正确性的验证：

1	2	3
8		4
7	6	5

```
283104765
move 2 to 5
move 1 to 2
move 4 to 1
move 5 to 4
4
--------------------------------
Process exited after 16.33 seconds with return value 0
请按任意键继续. . .
```

```
273645801
move 9 to 8
move 6 to 9
move 5 to 6
move 8 to 5
move 7 to 8
move 4 to 7
move 5 to 4
move 2 to 5
move 1 to 2
move 4 to 1
move 5 to 4
move 8 to 5
move 7 to 8
move 4 to 7
move 5 to 4
15
--------------------------------
Process exited after 1.23 seconds with return value 0
请按任意键继续. . .
```

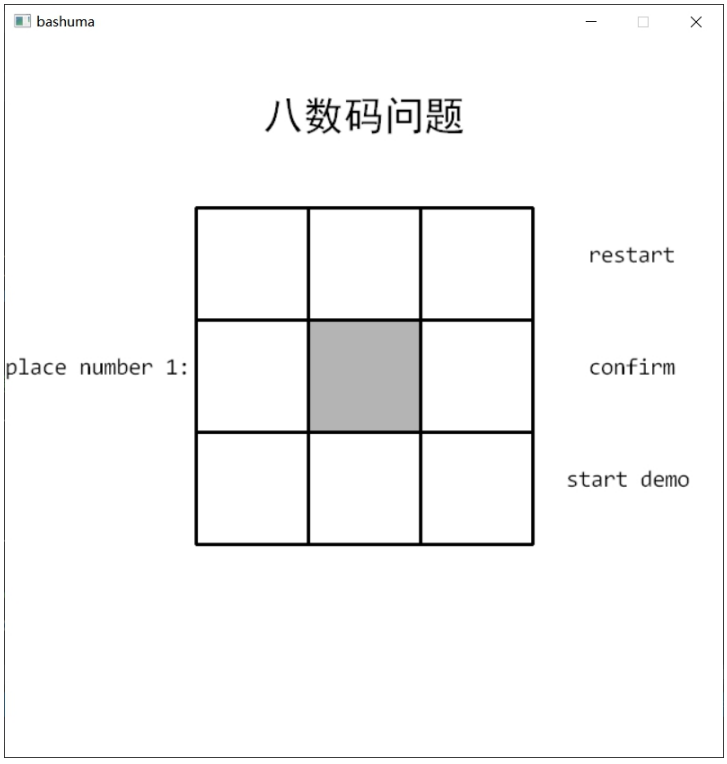
```
603712458
move 5 to 2
move 8 to 5
move 7 to 8
move 4 to 7
move 5 to 4
move 6 to 5
move 9 to 6
move 8 to 9
move 7 to 8
move 4 to 7
move 1 to 4
move 2 to 1
move 5 to 2
move 8 to 5
move 7 to 8
move 4 to 7
move 5 to 4
move 6 to 5
move 9 to 6
move 8 to 9
move 7 to 8
move 4 to 7
move 5 to 4
23
--------------------------------
Process exited after 1.248 seconds with return value 0
请按任意键继续. . .
```

其中用到的算法为 IDA*，经验证，该算法的最短步数和完成最短步数的步骤均具有正确性，且复杂度较低。

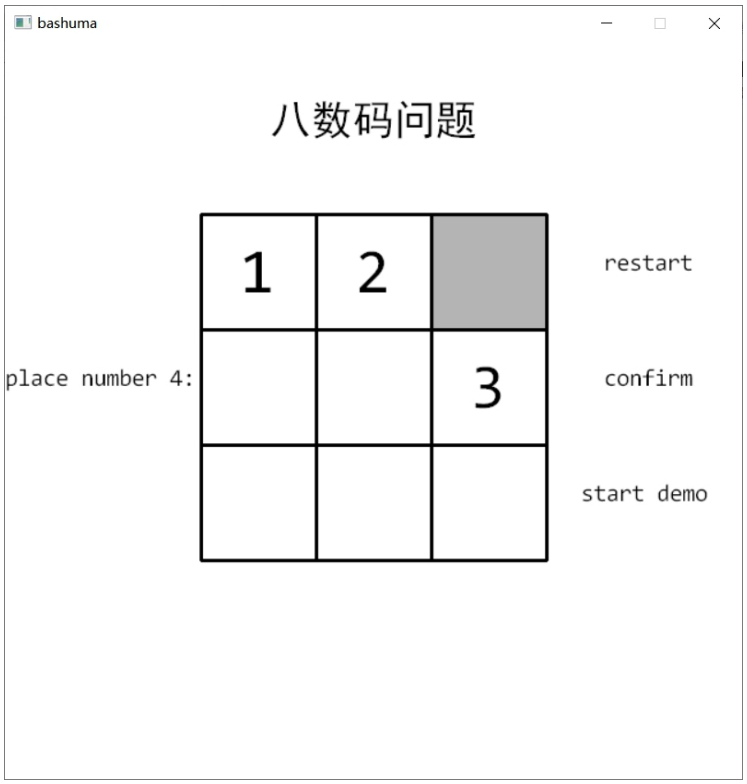
之后我们把目标状态设置如下，进行图形化：

1	2	3
4	5	6
7	8	

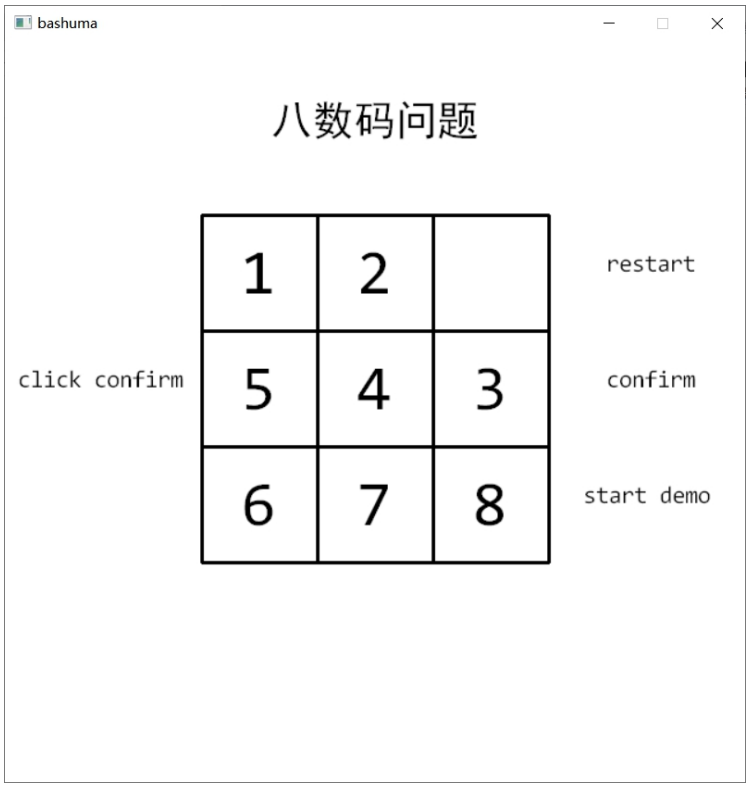
首先初始化界面：



为了防止输入的个数不够，或者输入的数字不合法，这个程序会让使用者按照 1-8 的顺序，依次点击这些方格，实现数字的填入。



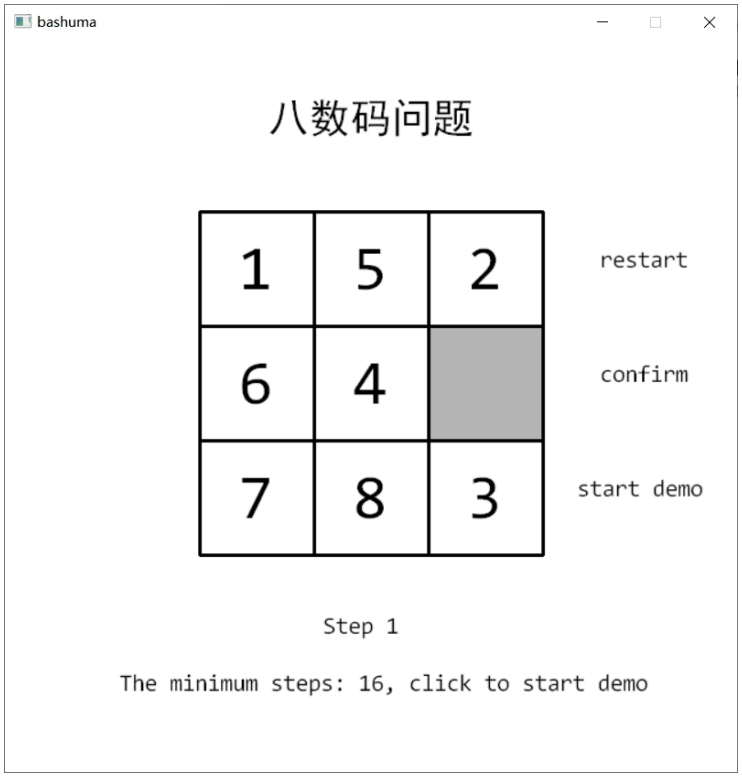
若是发现自己填错了数字，可以随时点击 restart 按钮，再次初始化页面，重新填入数字。



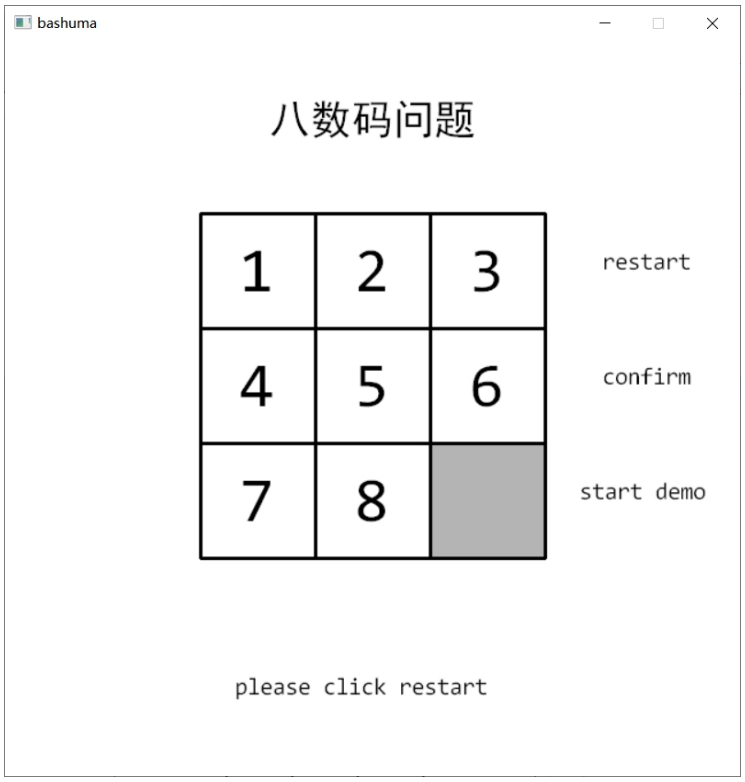
在 8 个数字都填好了之后，会提醒使用者点击 confirm 键来确认自己的输入。

在点击了 confirm 键之后，若是发现这个输入是不合法的，则会提示没有合法答案，并提示点击 restart 重新开始。

若这个输入是合法的，那么会显示这个问题求解所需要的最少步数。并且提醒点击 start demo，开始动态演示。



之后程序会开始动态演示，一秒一步，将八数码变为目标状态。



至此，程序结束。

附：

最后实现图形化界面的 cpp 文件为 graphic.cpp。在压缩包中，还另有几种不同的方式进行八数码问题的求解（不是同一个项目，分别有一个 main 函数）。均在 luogu p1379 八数码难题上进行了正确性和复杂度的验证。