同济大学

课程名称　人工智能

设计名称　运用 A*算法解决八数码问题

学院(系)　电子与信息工程学院

专　　业　计算机科学与技术

成　　员　蔡乐文　　学号　1353100

成　　员　周宇星　　学号　1352652

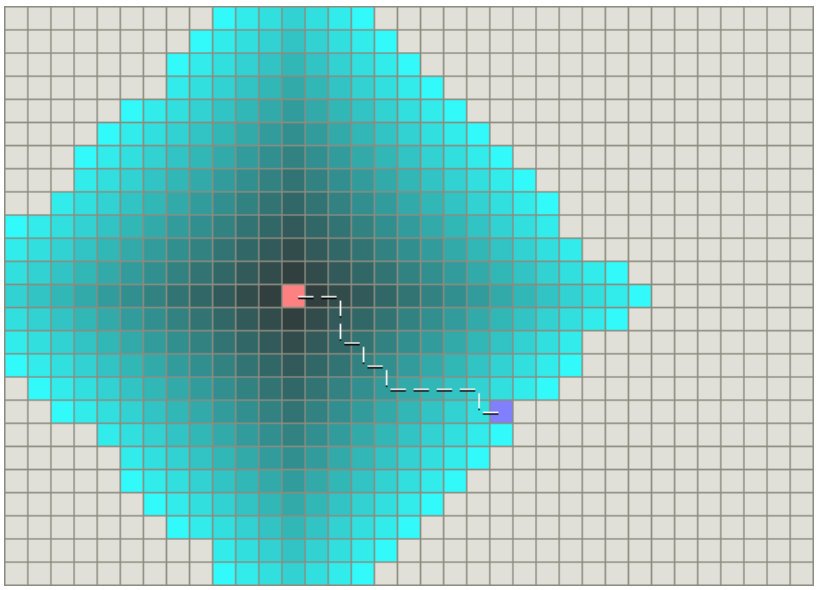
2015-2016 学年 第 2 学期

目录

问题背景 2
A*算法 2
八数码问题 3
状态的存储方式 4
状态的变化 6
具体代码实现流程 7
实验总结 9
字符界面版源代码 9
带图形显示的代码： 14

问题背景

在许多计算机应用中，经常会碰到路径最优化问题，也就是人们常数的最短路问题。最短路问题（short-path problem）是这样定义的，若网络中的每条边都有一个数值（长度、成本、时间等），则找出两节点（通常是源节点和阱节点）之间总权和最小的路径就是最短路问题。对于边权全为 1 的图而言，可以利用 BFS 来解决。对于更一般无负权图而言，可以利用 dijkstra 来解决。无论是 BFS 还是 Dijkstra 都会搜索大量的无用空间。因此，我们需要更加有效的搜索算法。



（利用 BFS 或 Dijkstra 算法解决平面最短路示意图）

A*算法

经过分析，可以发现 BFS 和 Dijkstra 都没有有效的利用环境信息，此时，我们需要更换一种更加有效的，利用了环境信息的搜索算法——A*算法。

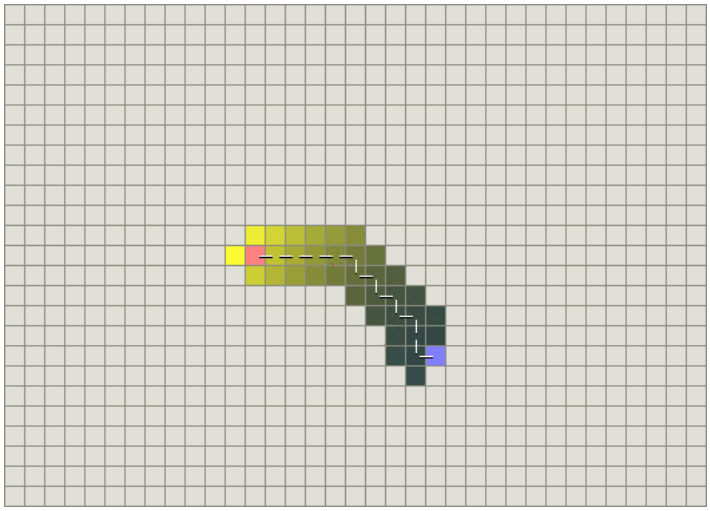
A*算法利用的是维护一个 F(x) = H(x) + G(x) 的 OPEN 列表，每次从 OPEN 列表中选取一个 F(x)最小的点往外扩展，其中：

H(x)为从源状态到 x 状态的最小花费；

G(x)为从 x 状态到目标状态的估计最小花费；

H(x)可以通过每一步操作费用叠加得到，但是 G(x)就不能通过这样的手段获得，因此，我们需要设计一种方法，使得在搜索过程中可以预测状态 X 与目标值的差异。因为不同问题的预测方法截然不同，所以我们只在下面实现八数码问题时，提供八数码问题的 G(X)计算方法。

虽然 A*算法和 dijkstra 算法类似，都是采用最优先搜索策略。但是由于 A*算法不仅采用了过去的信息去搜索，同时也估计了未来的情况，所以 A*搜索可以避免很多无用的搜索。



图为 A*算法解决简单网络的示意图

从上图可以看出，同样的搜索问题，由于 A*算法考虑预测方向因素使得 A*算法搜索量很少

八数码问题

对于状态 X，记其每个数字的位子为X_{p_i}，对于目标状态 T，记其每个数字的位子为T_{P_i}。则其启发函数值为，X_{p_i}与T_{P_i}不同的个数。

但是，我们可以发现上述启发函数没有充分的利用已知信息。

如果九宫格上只有一个数字，那么其移动到目标位置的代价应该是数字所在位置与目标位置的曼哈顿距离。由于，八数码中一次仅移动一个数字，所以我们完全可以令启发函数为：$C\sum_1^8(X_{p_i}-T_{P_i})$，其中 C 为大于等于 1 的参数。如果 C 大于 1，意味着 A*搜索出的解未必为最优解。

由于八数码问题的状态较为特殊，所以，我们需要构造较大的空间去存储 OPEN 表中的状态。

状态的存储方式

九字节状态存储

很显然，我们可以直接存储每个九宫格。但是，这种做法存储代价大，查询效率低。

位串存储方式

考虑到每个九宫格上只有 8 种数字和一个空位置，所以针对每个位置，可以使用 3bit 去存储该位置上的数字是什么，然后在利用 4bit 去记录空位置的编号。

这样下来，每个状态，我们仅仅只需要 31bit 就能存储一个八数码状态了。这样一来，我们就可以使用一个 long int 类型来存储一个 八数码状态了。使得存储和查询的速度加快。

同时我们用 000 表示数字 1, 001 表示数字 2，002 表示数字 3 ，以此类推，这样就可以获得下面这个序列。

空格位置	(2,2)	(2,1)	(2,0)	(1,2)	(1,1)	(1,0)	(0,2)	(0,1)	(0,0)
100	111	110	101	100	000	010	001	011	000

康拓展开法

通过计算，我们可以计算出，八数码的状态数应该是小于 9！的。也就是说，八数码的状态应该是小于等于 362, 880 个。然而，前面却使用了 31bit 来存储其值，同时必须借助高级数据结构才能维护 OPEN 表。

因此，我们需要一种更加强大的方式存储我们的状态——康托展开。利用康拓展开，我们可以计算出每一个八数码状态其按照字典序排列时的序号（小于等于 362, 880 ），这样一来，我们就可以借助数组来存储搜索过的状态了。同时，我们可以利用康拓逆展开来根据序

号还原出八数码的状态。

康托展开公式为：

$$\sum_{1}^{n} a_n * (n-1)!$$

将上述状态转化为一个序列 1,4,2,3,9,5,6,7,8 （空用 9 表示）

利用康托展开，得到：

$$0 * 8! + 2 * 7! + 0 * 6! + 0 * 5! + 4 * 4! + 0 * 3! + 0 * 2! + 0 * 1! + 0 * 0! = 10176$$

对 10176 进行康托逆展开运算：

$$\frac{10176}{8!} = 0 \ldots\ldots 10176$$

$$\frac{96}{4!} = 4 \ldots\ldots 0$$

$$\frac{10176}{7!} = 2 \ldots\ldots 96$$

$$\frac{0}{!3} = 0 \ldots\ldots 0$$

$$\frac{96}{6!} = 0 \ldots\ldots 96$$

$$\frac{0}{!2} = 0 \ldots\ldots 0$$

$$\frac{96}{5!} = 0 \ldots\ldots 96$$

$$\frac{0}{!1} = 0 \ldots\ldots 0$$

$$\frac{0}{!0} = 0 \ldots\ldots 0$$

我们可以得到，0,2,0,0,4,0,0,0,0 的序列

然后对序列进行进一步处理得 1,4,2,3,9,5,6,7,8。也就是我们刚刚拿来计算康托展开的八数码的状态。

由于利用 A*算法解决问题时需要每次从 OPEN 表中取出 F(x)值最小的元素，所以我们还需要一个优先队列，使得我们可以快速的查询出当前 OPEN 表中的最优状态。

状态的变化

三种不同储存方式在状态改变上的操作有着一定的不同。

九字节状态存储

由于九字节方式的状态存储简单明了，所以改变状态时仅需检查是否将空格移出九宫格的边界。

位串存储方式

位串存储方式将空间分布全部压到了一个一维空间里。虽然可以通过将位串还原为九个数字，然后再进行移动，但是这样做就无法体现出位串储存方式的优势了。

我们可以通过位置信息的四 bit 的信息读出去空位置的信息。然后可以快速确定出，哪四个位置是要操作的。但是，相对于九字节状态存储方法而言，较难判断出移动时是否超出边界，但是，哪些位置是可以往哪边移动我们是事先知道的，所以我们仅需设置允许移动状态表即可。

通过 strategy 来描述每一种移动方式，位置的变化：

```
const int strategy[strategyNum] = {-3, +3, -1, +1};
```

通过 invalid 来描述每一种方式移动，哪些位置是禁止：

```
const uint invalid[strategyNum] = {7, 448, 73, 292};
```

同时，需要一些宏定义来完成诸如取数，移动等操作。

```
//获取空位置的编号
#define getEmpty(x) ((x)>>27)
//将 x 状态的 y 位置设置为空
#define setEmpty(x,y) (((x)&(0xffffffff-(0xf<<27)))|((y)<<27))
//清楚 x 状态的 y 位置
#define clearE(x,y) ((x)&(0xffffffff-(7<<(3*(y)))))
//获得 x 状态的 y 位置的数字
#define getE(x,y) ((x)>>(3*(y))&7)
//将 z 放入 x 状态的 y 位置
#define putE(x,y,z) ((x)|((z)<<((y)*3)))
```

康拓展开法

康拓展开法的数字特征变换毫无规律可言，所以只能通过还原为九字节状态后再进行操作。

具体代码实现流程

本次代码实现，我们使用了位串储存方式来进行操作。

由于我们独特的实现方式，我们首先对读入的数据进行格式转换，将九个数字变成一个整型数字。

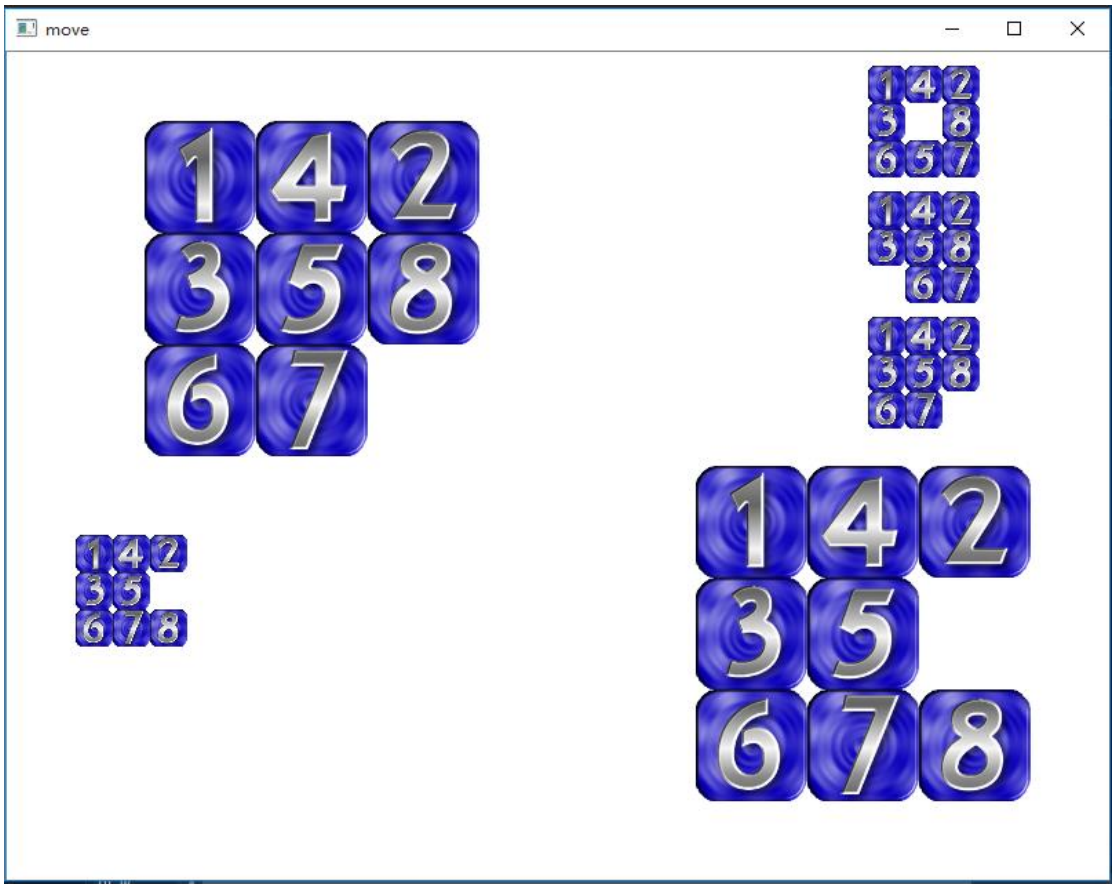
对初始的状态和目标的状态都进行相应的处理，这样我们就能得到两个数字——初始状态和目标状态。

下一步，我们将这个初始状态放入优先队列中，同时将初始状态与目标状态都映射到一个小的递增的唯一的整数上去（后面的所有状态都这么处理）。

接着不断取出优先队列中值最优的状态进行扩展（具体见代码注释），直到满足停止条件。

如果可以搜索到目标状态，则输出操作步骤，否则输出无解。

实验结果展示



（搜索过程界面）

（结果输出界面）

实验总结

本次实验由多人组队完成，非常讲究组员与组员之间的合作与沟通的能力。同时，本次实验是第一次采用采用到了图形界面的编程，对于自学能力有一定的要求。

因为一开始的时候并不知道如何使用图形界面，所以在不得已的情况下，才能先编写核心的部分。因为设计得当，程序在将字符结果输出转为图形结果输出的过程中没有遇到大的障碍。

由于使用了 A*算法进行搜索，所以 A*算法的估计函数的选取对于程序运行结果极为重要。我们进行了多次的比较，测试，才能最终确定使用说明样的估计函数，极大的考验了我们的耐心与毅力。有助于锻炼我们解决一些难题时的恒心与决心。

本次实验，由蔡乐文（1353100）和周宇星（1352652）完成。其中，蔡乐文设计并完成位串存储方式的研究与实现，周宇星提供了康托展开法的设计思路，蔡乐文与周宇星共同实现了 A*搜索的算法与其估计函数的设计。蔡乐文设计并完成了搜素过程的显示，周宇星则相应的实现了最终结果的显示模块。

字符界面版源代码

```
/*
启发函数的选择对结果的影响
*/
#include <cstdio>
#include <iostream>
#include <vector>
#include <cstring>
#include <set>
#include <windows.h> //取系统时间
#include <iomanip>
#include <map>
#include <queue>

//获取空位置的编号
#define getEmpty(x) ((x)>>27)
//将 x 状态的 y 位置设置为空
#define setEmpty(x,y) (((x)&(0xffffffff-(0xf<<27)))|((y)<<27))
//清楚 x 状态的 y 位置
#define clearE(x,y) ((x)&(0xffffffff-(7<<(3*(y)))))
//获得 x 状态的 y 位置的数字
#define getE(x,y) ((x)>>(3*(y))&7)
//将 z 放入 x 状态的 y 位置
#define putE(x,y,z) ((x)|((z)<<((y)*3)))
```

```
using namespace std;

typedef unsigned int uint;

const int DNUM = 9;
const int strategyNum = 4;

//移动策略表
const int strategy[strategyNum] = {-3, +3, -1, +1};
//禁止移动表
const uint invalid[strategyNum] = {7, 448, 73, 292};

//将 9 字节的数字转为一个 int 数
int getOpt(const int p[DNUM]){
	int b[DNUM];
	memset(b, 0, sizeof(b));
	for (int i = 0; i < DNUM; ++i) {
		if (p[i] < -1 || p[i] > 7) return -1;
		if (b[p[i] + 1]) return -1;
		b[p[i] + 1] = 1;
	}
	int res = 0;
	for (int i = 0; i < DNUM; ++i) {
		if (p[i] != -1) {
			res |= (p[i] << (3*i));
		} else {
			res |= (i << 27);
		}
	}
	return res;
}

//将一个 int 数转为九宫格状态输出
void showOpt(const int opt){
	int emptyPlace = getEmpty(opt) & 0xf;
	puts("+-+-+-+");
	for (int i = 0; i < 9; ++i) {
		if (i != emptyPlace) {
			printf("|%d", 1 + ((opt & (7<<(i*3))) >> (i*3)));
		} else {
			printf("| ");
		}
		if (i%3==2) puts("|\n+-+-+-+");
	}
```

```
}

//设置最大状态数
const int MAXOPT = 362881;

//Open 表
priority_queue<pair<int, int> > que;

//前驱
int pre[MAXOPT];

//已经消耗的花费
int cost[MAXOPT];

//已经搜索到的状态的编号
map<int, int> searchSet;

int ans = 0;
//进行答案的回溯
void showAns(int mark){
	if (mark == -1) return;
	showAns(pre[mark]);
	printf("第%d 步状态为：\n", ans);
	++ans;
	showOpt(optR[mark]);
}

//通过状态 opt 获得九个状态的位置
int getArr(int opt, int x[9], int y[9]){
	int k = getEmpty(opt);
	for (int i = 0; i < 3; ++i) {
		for (int j = 0; j < 3; ++j) {
			if (i * 3 + j == k) continue;
			int n = getE(opt, i*3+j);
			x[n+1] = i;
			y[n+1] = j;
		}
	}
	return k;
}

//计算两个状态之间的差异值
int dif(const int opt_, const int targe_){
	int res = 0;
```

```
    int initx[9], inity[9];
    int tarx[9], tary[9];

    getArr(opt_, initx, inity);
    getArr(targe_, tarx, tary);
    for (int i = 0; i < 9; ++i) {
        res += abs(initx[i] - tarx[i]) + abs(inity[i] - tary[i]);
    }
    return res;
}

//增加新状态的操作
int dealNewNode(const int & opt_, const int & targe_, const int & preN, const int & cost_){
    static int keyCount = 0;
    que.push(make_pair(-(cost_ + dif(opt_, targe_)), opt_));
    pre[keyCount] = preN;
    cost[keyCount] = cost_;
    optR[keyCount] = opt_;
    searchSet[opt_] = keyCount;
    return keyCount++;
}

//A*算法搜索
void search(const int opt_, const int targe_){
    dealNewNode(opt_, targe_, -1, 0);
    while (!que.empty()){
        pair<int, int> ele = que.top(); //获得 Open 表中的第一项元素
        int opt = ele.second;      //获得元素的编号
        que.pop();      //删除 Open 表中的第一项元素
        int preN = searchSet[opt];     //通过编号，获得其状态
        int emptyPlace = getEmpty(opt);   //获得其空位置
        for (int i = 0; i < 4; ++i) {
            //如果本状态不允许有进行本移动操作，则跳过
            if (((1<<emptyPlace) & invalid[i]) > 0) continue;

            //计算新状态的位置
            int newEmptyPlace = emptyPlace + strategy[i];

            //获得相应位置的元素
            int element = getE(opt, newEmptyPlace);

            //将相应状态的元素清空
            int optT = clearE(opt, newEmptyPlace);
```

```
                //将原来的空位置放置元素
                int newOpt = setEmpty(putE(optT, emptyPlace, element), newEmptyPlace);
                //如果该元素被搜寻过
                if (searchSet.find(newOpt) != searchSet.end()) {
                    int COST = cost[searchSet[opt]] + 1;
                    int p = searchSet[newOpt];
                    if (cost[p] > COST + 1) {
                        cost[p] = COST + 1;
                        que.push(make_pair(-(cost[p] + dif(opt_, targe_)), opt_));
                    }
                    continue;
                }

                //添加新元素
                int key = dealNewNode(newOpt, targe_, preN, cost[searchSet[opt]] + 1);

                //如果等于目标状态，则退出
                if (newOpt == targe_) {
                    showAns(key);
                    cout<<key<<endl;
                    return ;
                }
            }
        }
        cout << "NO SOLUTION" << endl;
}

int eight_digit_code(const int p[DNUM], const int t[DNUM]){
        int opt = getOpt(p);
        if (opt == -1) return -1;
        int targe = getOpt(t);
        if (targe == -1) return -1;
        search(opt, targe);
        return 0;
}

int main(){
        LARGE_INTEGER tick, fc_begin, fc_end;
        QueryPerformanceFrequency(&tick);    //获得时钟频率
        QueryPerformanceCounter(&fc_begin);       //获得初始硬件定时器计数

        int p[9] ={7,2,4,0,1,6,3,5,-1};//{3, 0, 1, -1, 4, 6, 2, 5, 7};
        int targe[9] = {0,1,2,3,4,5,6,7,-1};//{7, 3, 1, 5, -1, 6, 4, 2, 0};
        eight_digit_code(p, targe);
```

```
    QueryPerformanceCounter(&fc_end);//获得终止硬件定时器计数
    cout << setiosflags(ios::fixed) << setprecision(3);
    cout << "时钟频率：" << double(tick.QuadPart) / 1024 / 1024 << "GHz" << endl;
    cout << setprecision(0);
    cout << "时钟计数：" << double(fc_end.QuadPart - fc_begin.QuadPart) << endl;
    cout << setprecision(6) << double(fc_end.QuadPart - fc_begin.QuadPart) / double(tick.QuadPart) << "秒" << endl;
    return 0;
}
```

带图形显示的代码：

```
#include "opencv/cv.hpp"
#include <queue>
#include <iostream>
#include <cstdio>
#include <vector>
#include <set>
#include <utility>
#include <cstring>
#include <list>
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <windows.h> //取系统时间
#include <cmath>
#include <map>

const int delay_t = 50;
const int image_size = 81;

//获取空位置的编号
#define getEmpty(x) ((x)>>27)
//将 x 状态的 y 位置设置为空
#define setEmpty(x,y) (((x)&(0xffffffff-(0xf<<27)))|((y)<<27))
//清楚 x 状态的 y 位置
#define clearE(x,y) ((x)&(0xffffffff-(7<<(3*(y)))))
//获得 x 状态的 y 位置的数字
#define getE(x,y) ((x)>>(3*(y))&7)
```

```
//将 z 放入 x 状态的 y 位置
#define putE(x,y,z) ((x)|((z)<<((y)*3)))

using namespace std;

using namespace cv;

//该命名域负责实现画面显示
namespace draw {
//定义彩色背景图
	Mat background(image_size * 3, image_size * 3, CV_8UC3);
	Mat num[9];
	Mat paintImage(600, 800, CV_8UC3);
//输出初始化
	void OptInit(const char * file_name) {
		for (int i = 1; i <= 9; ++i) {
			char fName[256];
			sprintf_s(fName, "%s%d.bmp", file_name, i);
			Mat tmp = imread(fName);
			resize(tmp, num[i - 1], Size(image_size, image_size), 0, 0, CV_INTER_LINEAR);
		}
		memset(background.data, 0xff, background.step*background.rows);
	}
//将一个图像复制到另一个图像中
	int ImageCopy(Mat & src, Mat & dis, int x = 0, int y = 0) {
		if (src.channels() != dis.channels()) {
			return -1;
		}

		int src_r = src.rows;
		int src_c = src.cols * src.channels();
		uchar * src_data = src.data;
		int src_step = src.step;

		int dis_r = dis.rows;
		int dis_c = dis.cols * dis.channels();
		uchar * dis_data = dis.data;
		int dis_step = dis.step;

		y *= dis.channels();
		for (int i = 0; i < src_r; ++i) {
			//	cout << i << ' ' << (x + i) << endl;
			memcpy(dis_data + (x + i) * dis_step + y,
				src_data + i * src_step,
```

```
                src_c);
        }
        return 0;
    }
//利用一个 int 数字获得一个图像
    Mat getImage(int opt) {
        Mat res(image_size * 3, image_size * 3, CV_8UC3);
        ImageCopy(background, res);
        int emptyPlace = getEmpty(opt) & 0xf;

        int x = 0;
        for (int i = 0; i < 9; ++i) {
            if (i != emptyPlace) {
                ImageCopy(num[((opt & (7 << (i * 3))) >> (i * 3))], res, i / 3 * image_size, i % 3
* image_size);
            }
        }
        return res;
    }

//显示数字的移动
    void showMove(int opt, int from, int x, int y) {
        int emptyPlace = getEmpty(opt) & 0xf;
        int sx = from / 3 * image_size, sy = from % 3 * image_size;
        int dx = emptyPlace / 3 * image_size, dy = emptyPlace % 3 * image_size;

        int cx = 0, cy = 0;
        if (sx < dx) cx = -9;
        else if (sx > dx) cx = 9;
        if (sy < dy) cy = -9;
        else if (sy > dy) cy = 9;

        for (int bx = dx, by = dy; bx != sx || by != sy; bx += cx, by += cy) {
            Mat res = getImage(opt);
            ImageCopy(num[8], res, sx, sy);
            ImageCopy(num[((opt & (7 << (from * 3))) >> (from * 3))], res, bx, by);
            ImageCopy(res, paintImage, x, y);
            imshow("move", paintImage);
            waitKey(delay_t);
        }
        ImageCopy(draw::getImage(opt), paintImage, x, y);
        imshow("move", paintImage);
    }
```

```
//显示答案的界面
    void showAnsMove(int opt, int from) {
        int emptyPlace = getEmpty(opt) & 0xf;
        int sx = from / 3 * image_size, sy = from % 3 * image_size;
        int dx = emptyPlace / 3 * image_size, dy = emptyPlace % 3 * image_size;

        int cx = 0, cy = 0;
        if (sx < dx) cx = -9;
        else if (sx > dx) cx = 9;
        if (sy < dy) cy = -9;
        else if (sy > dy) cy = 9;
        Mat res;
        for (int bx = dx, by = dy; bx != sx || by != sy; bx += cx, by += cy) {
            res = getImage(opt);
            ImageCopy(num[8], res, sx, sy);
            ImageCopy(num[((opt & (7 << (from * 3))) >> (from * 3))], res, bx, by);
            imshow("RESULT", res);
            waitKey(delay_t);
        }
    }

//将背景图进行初始化
    Mat paintImageInit() {
        int row = paintImage.rows;
        int col = paintImage.cols * paintImage.channels();
        uchar * data = paintImage.data;
        int step = paintImage.step;

        for (int i = 0; i < row; ++i) {
            memset(data, 0xff, col);
            data += step;
        }

        return paintImage;
    }



//画一个八数码状态
    void drawOpt(int opt) {
        imshow("drawOpt", draw::getImage(opt));
        waitKey(1000);
    }
}
```

```
namespace search {

	typedef unsigned int uint;

	const int DNUM = 9;
	const int strategyNum = 4;

//定义搜索的方向
	const int strategy[strategyNum] = { -3, +3, -1, +1 };
//定义每种搜索方式不能进行搜索的位置
	const uint invalid[strategyNum] = { 7, 448, 73, 292 };
//获得状态的整形表示
	int getOpt(const int p[DNUM]) {
		int b[DNUM];
		memset(b, 0, sizeof(b));
		for (int i = 0; i < DNUM; ++i) {
			if (p[i] < -1 || p[i] > 7) return -1;
			if (b[p[i] + 1]) return -1;
			b[p[i] + 1] = 1;
		}
		int res = 0;
		for (int i = 0; i < DNUM; ++i) {
			if (p[i] != -1) {
				res |= (p[i] << (3 * i));
			}
			else {
				res |= (i << 27);
			}
		}
		return res;
	}
//字符方式显示一个状态
	void showOpt(const int opt) {
		int emptyPlace = getEmpty(opt) & 0xf;
		puts("+-+-+-+");
		for (int i = 0; i < 9; ++i) {
			if (i != emptyPlace) {
				printf("|%d", 1 + ((opt & (7 << (i * 3))) >> (i * 3)));
			}
			else {
				printf("| ");
			}
			if (i % 3 == 2) puts("|\n+-+-+-+");
```

```
            }
        }

        const int MAXOPT = 362881;

    //A*搜索需要使用的数组以及数据结构
        priority_queue<pair<int, int> > que;
        int pre[MAXOPT];
        int cost[MAXOPT];
        int optR[MAXOPT];
        int black[MAXOPT];
        int TA;
        map<int, int> searchSet;
    //获得整形数字所表示的八数码的各个位置
        int getArr(int opt, int x[9], int y[9]) {
            int k = getEmpty(opt);
            x[0] = 0;
            y[0] = 0;
            for (int i = 0; i < 3; ++i) {
                for (int j = 0; j < 3; ++j) {
                    if (i * 3 + j == k) continue;
                    int n = getE(opt, i * 3 + j);
                    x[n + 1] = i;
                    y[n + 1] = j;
                }
            }
            return k;
        }
    //计算当前状态与目标状态的差异
        int dif(const int opt_, const int targe_) {
            //return 0;
            int res = 0;
            int initx[9], inity[9];
            int tarx[9], tary[9];

            getArr(opt_, initx, inity);
            getArr(targe_, tarx, tary);
            for (int i = 0; i < 9; ++i) {
                //res += (initx[i] == tarx[i]) + (inity[i] == tary[i]);
                res += abs(initx[i] - tarx[i]) + abs(inity[i] - tary[i]);
            }
            return res;
        }
    //增加一个新状态
```

```
    int dealNewNode(const int & opt_, const int & targe_, const int & preN, const int & cost_) {
        static int keyCount = 0;
        cout << dif(opt_, targe_) << endl;
        que.push(make_pair(-(cost_ + dif(opt_, targe_)), opt_));
        pre[keyCount] = preN;
        cost[keyCount] = cost_;
        optR[keyCount] = opt_;
        searchSet[opt_] = keyCount;
        return keyCount++;
    }

    int ans = 0;
//显示图像移动界面
    void showAns(int mark) {
        if (mark == -1) return;
        showAns(pre[mark]);
        printf("第%d 步状态为：\n", ans);
        ++ans;
        int preEmpty = getEmpty(optR[pre[mark]]) & 0xf;
        //cout << preEmpty << endl;
        if (pre[mark] != -1) {
            draw::showAnsMove(optR[mark], preEmpty);
            waitKey(delay_t);
        }
        imshow("RESULT", draw::getImage(optR[mark]));
    }
//实现列表浏览画面
    void showSearchList(list<int> L, int stdOpt) {
        vector<int> table;
        for (list<int>::iterator i = L.begin(); i != L.end(); ++i)
            table.push_back(*i);

        Mat LImage(81 * 5 + 40, 81, CV_8UC3);

        int l = 0;
        int p = 1, q = 81;
        if (table.size() <= 10) {
            p = 9;
            q = 9;
        }
        else if (table.size() <= 100) {
            p = 3;
            q = 27;
        }
```

```
else {
	p = 1;
	q = 81;
}
if (table.size() == 0) return;
int minId = table[0];
int minV = 100;// (cost[searchSet[minId]] + dif(table[0], TA));
for (int i = 0; i < table.size(); ++i) {
	if (l < 4) ++l;
	int rows = LImage.rows, cols = LImage.cols, ch = LImage.channels();
	for (int i = 0; i < rows; ++i)
		memset(LImage.data + i * cols * ch, 0xff, cols * ch);

	int id = table[i];
	int v = (cost[searchSet[id]] + dif(table[i], TA));
	if (v <= minV) {
		minV = v;
		minId = id;
		cout << v << ' ' << minV << endl;
	}
	if (id == stdOpt) {
		minId = stdOpt;
		minV = 0;
	}

	for (int k = i - l + 1; k <= i; ++k) {
		Mat pic = draw::getImage(table[k]), tmp;
		resize(pic, tmp, Size(81, 81), 0, 0, CV_INTER_LINEAR);
		draw::ImageCopy(tmp, LImage, (l - (i - k) - 1) * 91, 0);
	}
	Mat image_roi(81 * 3 + 20, 81, CV_8UC3);
	uchar * data = LImage.data;
	int size = LImage.cols * LImage.channels();
	for (int k = 0; k < p; ++k) {
		for (int j = 5 * 81 + 39; j >= q; --j) {
			memcpy(data + (j)* size, data + (j - q) * size, size);
		}

		//imshow("LIMAGE", LImage);

		for (int r = 0; r < 81 * 3 + 20; ++r) {
			memcpy(image_roi.data + r * (size), LImage.data + (r + 81) * (size), size);
		}
```

```
                //cout << k << endl;
                draw::ImageCopy(image_roi, draw::paintImage, 10, 625);
                imshow("move", draw::paintImage);

                waitKey(1);
            }
            //Mat pic = draw::getImage(*i);
            //Mat tmp;
            //resize(pic, tmp, Size(81, 81), 0, 0, CV_INTER_LINEAR);
            //draw::ImageCopy(tmp, draw::paintImage, i * tot++, 550);
            //waitKey(100);

            draw::ImageCopy(draw::getImage(minId), draw::paintImage, 50, 100);
            imshow("move", draw::paintImage);

            waitKey(1);
        }
    }

//核心算法部分，介绍与字符界面版的相同
    void search(const int opt_, const int targe_) {
        dealNewNode(opt_, targe_, -1, 0);
        list<int> waitList;
        waitList.push_back(opt_);
        while (!que.empty()) {
            pair<int, int> ele = que.top();
            int opt = ele.second;
            cout << -ele.first << endl;
            draw::paintImageInit();
            showSearchList(waitList, opt);
            waitList.remove(opt);
            draw::ImageCopy(draw::getImage(opt), draw::paintImage, 50, 100);
            imshow("move", draw::paintImage);
            waitKey(delay_t);
            cout << waitList.size() << endl;
            que.pop();
            int preN = searchSet[opt];
            int emptyPlace = getEmpty(opt);
            for (int i = 0; i < 4; ++i) {
                if (((1 << emptyPlace) & invalid[i]) > 0) continue;
                int newEmptyPlace = emptyPlace + strategy[i];
                int element = getE(opt, newEmptyPlace);
                int optT = clearE(opt, newEmptyPlace);
                int newOpt = setEmpty(putE(optT, emptyPlace, element), newEmptyPlace);
```

```
                    draw::showMove(newOpt, emptyPlace, 300, 500);
                    imshow("move", draw::paintImage);
                    int key = dealNewNode(newOpt, targe_, preN, cost[searchSet[opt]] + 1);
                    Mat tmp;
                    resize(draw::getImage(newOpt), tmp, Size(image_size * 1, image_size * 1), 0,
0, CV_INTER_LINEAR);
                    draw::ImageCopy(tmp, draw::paintImage, 350, 50 + i * 95);
                    imshow("move", draw::paintImage);
                    waitList.push_back(newOpt);
                    waitKey(delay_t * 10);

                    if (searchSet.find(newOpt) != searchSet.end()) {
                        int COST = cost[searchSet[opt]] + 1;
                        int p = searchSet[newOpt];
                        if (cost[p] > COST + 1) {
                            cost[p] = COST + 1;
                            waitList.push_back(newOpt);
                            que.push(make_pair(-(cost[p] + dif(opt_, targe_)), opt_));
                        }
                        if (newOpt == targe_) {
                            showAns(key);
                            std::cout << key << endl;
                            return;
                        }

                        continue;
                    }
                }
            }
            std::cout << "NO SOLUTION" << endl;
        }

        int eight_digit_code(const int p[DNUM], const int t[DNUM]) {
            int opt = getOpt(p);
            if (opt == -1) return -1;
            int targe = getOpt(t);
            if (targe == -1) return -1;
            search(opt, TA = targe);
            return 0;
        }
}

int main() {
```

```
	draw::OptInit("");
	LARGE_INTEGER tick, fc_begin, fc_end;
	QueryPerformanceFrequency(&tick);    //获得时钟频率
	QueryPerformanceCounter(&fc_begin);        //获得初始硬件定时器计数

                                                //{2, 1, 3, 5, 4, 6, 0, 7, -1}
                                                //{3, 0, 1, -1, 4, 6, 2, 5, 7};
                                                //     int p[9] ={0,-1,1,2,3,4,5,6,7};
                                                //     int targe[9] = {1,0,-1,2,3,4,5,6,7};

	int p[9] = { 0, 3, 1, 2 , 4, 7, 5, -1, 6};//初始状态
	int targe[9] = { -1, 0, 1, 2, 3, 4, 5, 6, 7};//目标状态
	search::eight_digit_code(p, targe);
	waitKey(0);
	QueryPerformanceCounter(&fc_end);//获得终止硬件定时器计数
	std::cout << setiosflags(ios::fixed) << setprecision(3);
	std::cout << "时钟频率：" << double(tick.QuadPart) / 1024 / 1024 << "GHz" << endl;
	std::cout << setprecision(0);
	std::cout << "时钟计数：" << double(fc_end.QuadPart - fc_begin.QuadPart) << endl;
	std::cout << setprecision(6) << double(fc_end.QuadPart - fc_begin.QuadPart) /
double(tick.QuadPart) << "秒" << endl;
	return 0;
}
```