浙江工业大学

人工智能及其应用实验报告

(2021 级)

实验二:A* 算法实验

学生姓名：温家伟

学生学号：202103151422

学科专业：大数据分析 2101 班

所在学院：理学院

提交日期：2023 年 11 月 30 日

目录

1 实验目的 **2**

2 实验原理 **2**

3 实验内容 **2**

4 实验结果 **3**

4.1 不同估价函数的比较 3

4.2 搜索空间 4

5 附录 **4**

5.1 game.h 4

5.2 test.cpp 22

1 实验目的

熟悉和掌握启发式搜索的定义、估价函数和算法过程，并利用 A* 算法求解 N 数码难题，理解求解流程和搜索顺序。

2 实验原理

A* 算法是一种启发式图搜索算法，其特点在于对估价函数的定义上。对于一般的启发式图搜索，总是选择估价函数 f 值最小的节点作为扩展节点。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，所以，可考虑每个节点 n 的估价函数值为两个分量：从起始节点到节点 n 的实际代价以及从节点 n 到达目标节点的估价代价。

3 实验内容

- 8 数码问题是一种滑动拼图，玩家需要在一个 3x3 的网格上移动 8 个标有 1 到 8 的方块以及一个空白方块。目标是通过尽量少的移动重新排列方块，使它们按行主序排列。玩家可以将方块沿水平或垂直方向滑动到空白方块的位置。

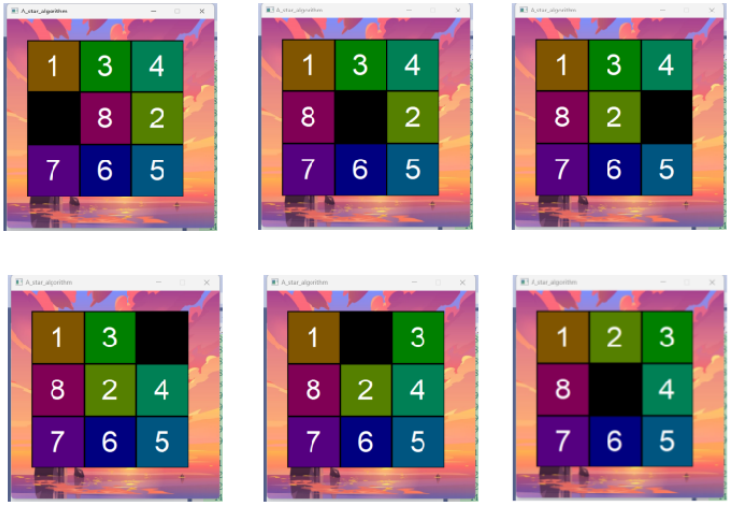

图 1: 移动步骤

- 参考 A* 算法核心代码，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。
- 在求解 8 数码问题的 A* 算法程序中，设置相同的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、生成节点数等。
- 对于 8 数码问题，设置与上述 2 相同的初始状态和目标状态，用宽度优先搜索算法（即令估计代价 h(n) = 0 的 A* 算法）求得问题的解，以及搜索过程中的扩展节点数、生成节点数。

4 实验结果

4.1 不同估价函数的比较

在本次实验中，我们研究了使用不同启发函数以及不采用启发性搜索的广度优先算法来解决八数码问题。我们的目标是比较这些方法在搜索时间和节点数方面的效率，并评估启发函数对搜索效果的影响。

首先，我们选择了两种不同的启发函数：简单错排和曼哈顿距离。通过与不使用启发函数的广度优先算法进行对比，我们发现使用启发函数的算法明显优于不使用启发函数的算法。具体比较结果如表 1 所示。从表中可

表 1: 不同算法的比较结果

	简单错排	曼哈顿距离	BFS
扩展节点数	25	12	263
生成节点数	74	36	695
运行时间	357	144	1737

以观察到，第二种启发函数（曼哈顿距离）相比于第一种启发函数（简单错排）表现更好。另外，使用启发函数的算法在搜索节点数和运行时间方面都明显优于不使用启发函数的广度优先搜索算法。这表明，在解决八数码问题时，选择适当的启发性信息可以提高搜索效率。

然而，理论上来讲，如果选用的启发性信息过强，可能会导致无法找到最优解。因此，在实际应用中，我们需要根据问题的特点和需求来选择适当的启发函数。

综上所述，本次实验结果表明，在解决八数码问题时，采用启发式搜索算法（如 A* 算法）结合适当的启发函数可以显著提高搜索效率。未来的研究可以进一步探索更多启发函数的设计和应用，以及在其他问题领域中的应用。

4.2 搜索空间

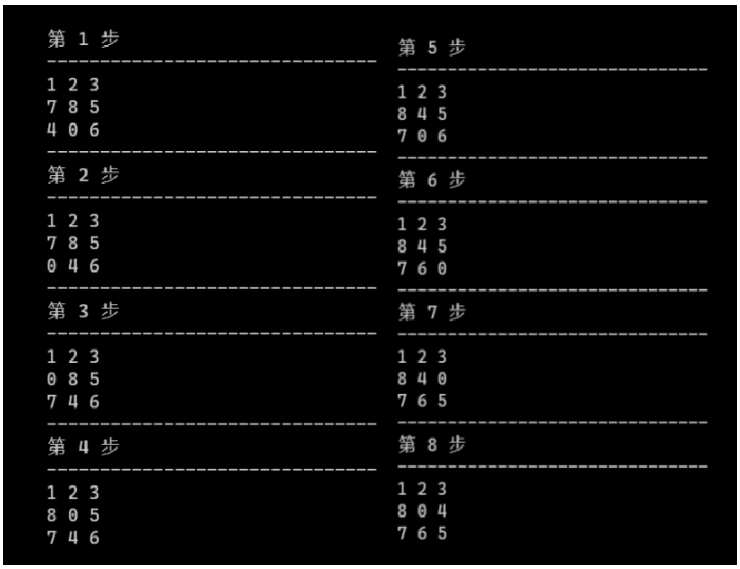

图 2: 实验结果

5 附录

5.1 game.h

```
1 #pragma once
2 #include <graphics.h>
3 #include <time.h>
4 #include <conio.h>
5
6 #define BACKGROUND_IMAGE L"background.jpg"
```

```
7  #define maxState 9999
8  #define N 3
9
10 IMAGE   g_Block[N * N];
11 byte    g_Map[N][N];
12 byte    g_EmptyX, g_EmptyY;
13
14 using namespace std;
15
16 void Draw()
17 {
18     for (int y = 0; y < N; y++)
19         for (int x = 0; x < N; x++)
20         {
21             if (g_Map[x][y] != 0)
22                 putimage(x * 100 + 40, y * 100 + 40,
                       &g_Block[g_Map[x][y]]);
23             else
24             {
25                 // 最后一片拼图暂时不显示
26                 setfillcolor(BLACK);
27                 solidrectangle(x * 100 + 40, y * 100
                       + 40, x * 100 + 139, y * 100 +
                       139);
28             }
29         }
30
31 }
32
33 bool isEqual(int a[N][N][maxState], int b[N][N], int
       n)
34 {
35     for (int i = 0; i < N; i++)
```

```
36      {
37          for (int j = 0; j < N; j++)
38          {
39              if (a[i][j][n] != b[i][j])
40                  return false;
41          }
42      }
43      return true;
44  }
45
46  bool isEqual(int a[N][N], int b[N][N])
        //检查两矩阵是否完全一致
47  {
48      for (int i = 0; i < N; i++)
49      {
50          for (int j = 0; j < N; j++)
51          {
52              if (a[i][j] != b[i][j])
53                  return false;
54          }
55      }
56      return true;
57  }
58
59  int evalute1(int state[N][N], int target[N][N])
        //估价函数h-计算不在位的将牌个数
60  {
61      int num = 0;
            //num表示当前矩阵state中不在目标位置上的将牌个数
62      for (int i = 0; i < N; i++)
63      {
64          for (int j = 0; j < N; j++)
65              if (state[i][j] != target[i][j])
```

```
66                  num++; //统计num
67      }
68      return num; //返回估价
69  }
70
71  int evalute2(int state[N][N], int target[N][N]) {
72      int manhattan = 0; // 曼哈顿距离累加
73
74      for (int i = 0; i < N; i++) {
75          for (int j = 0; j < N; j++) {
76              if (state[i][j] != target[i][j]) {
77                  // 寻找当前元素在目标矩阵中的位置
78                  for (int x = 0; x < N; x++) {
79                      for (int y = 0; y < N; y++) {
80                          if (state[i][j] ==
                                target[x][y]) {
81                              // 计算曼哈顿距离并累加
82                              manhattan += abs(i - x) +
                                    abs(j - y);
83                              break;
84                          }
85                      }
86                  }
87              }
88          }
89      }
90
91      return manhattan; // 返回估价
92  }
93
94  void findBrack(int a[N][N], int x, int y)
95  {
96      for (int i = 0; i < N; i++)
```

```
97      {
98          for (int j = 0; j < N; j++)
99          {
100             if (a[i][j] == 0)
101             {
102                 x = i;
103                 y = j;
104                 return;
105             }
106         }
107     }
108 }
109
110 bool move(int a[N][N], int b[N][N], int dir)
111 {
112     //1 up 2 down 3 left 4 right
113     int x = 0, y = 0;
114     for (int i = 0; i < N; i++)
115     {
116         for (int j = 0; j < N; j++)
117         {
118             b[i][j] = a[i][j];
                    //将原矩阵复制以进行移动操作
119             if (a[i][j] == 0)
120             {
121                 x = i;
122                 y = j; //标记空格位置
123             }
124         }
125     }
126     if (x == 0 && dir == 1)
127         return false;
            //四条if语句排除四种不可能的移动方向
```

```
128     if (x == N - 1 && dir == 2)
129         return false;
                //返回false意指此种移动方式不可行，回到调用函数重新选择移动方向
130     if (y == 0 && dir == 3)
131         return false;
132     if (y == N - 1 && dir == 4)
133         return false;
134     if (dir == 1)
            //按照传入的dir将空格往相应的方向移动
135     {
136         b[x - 1][y] = 0;
137         b[x][y] = a[x - 1][y];
138     }
139     else if (dir == 2)
140     {
141         b[x + 1][y] = 0;
142         b[x][y] = a[x + 1][y];
143     }
144     else if (dir == 3)
145     {
146         b[x][y - 1] = 0;
147         b[x][y] = a[x][y - 1];
148     }
149     else if (dir == 4)
150     {
151         b[x][y + 1] = 0;
152         b[x][y] = a[x][y + 1];
153     }
154     else
155         return false;
156     return true; //移动空格成功返回true
157 }
158
```

```
159 void statecpy(int a[N][N][maxState], int b[N][N], int
       n)
160 {
161     for (int i = 0; i < N; i++)
162     {
163         for (int j = 0; j < N; j++)
164         {
165             a[i][j][n] = b[i][j];
                   //将移动完的新矩阵复制到close表中，n可以表示第n步搜索结果
166         }
167     }
168 }
169
170 void getState(int a[N][N][maxState], int b[N][N], int
       n)
171 {
172     for (int i = 0; i < N; i++)
173     {
174         for (int j = 0; j < N; j++)
175         {
176             b[i][j] = a[i][j][n];
177         }
178     }
179 }
180
181 void statecpy(int a[N][N], int b[N][N])
182 {
183     for (int i = 0; i < N; i++)
184     {
185         for (int j = 0; j < N; j++)
186             a[i][j] = b[i][j]; //复制当前矩阵start
187     }
188 }
```

```
189
190 int checkAdd(int a[N][N][maxState], int b[N][N], int
    n)
191 {
192     for (int i = 0; i < n; i++)
193     {
194         if (isEqual(a, b, i))
195             return i;
                //若两矩阵相同则返回对应矩阵的编号
196     }
197     return -1;
198 }
199
200 int Astar1(int a[N][N][maxState], int start[N][N],
    int target[N][N], int path[maxState])
201 {
202     bool visited[maxState] = { false }; //
        true表示矩阵已被遍历
203     int fitness[maxState] = { 0 };
204     int passLen[maxState] = { 0 };
205     int curpos[N][N];
206     statecpy(curpos, start);
207     int id = 0, Curid = 0;
208     int expandedNodes = 0; // 扩展节点数
209     int generatedNodes = 1; //
        生成节点数，初始为1，因为起始节点已生成
210     fitness[id] = evalute1(curpos, target);
211     statecpy(a, start, id++);
212     while (!isEqual(curpos, target)) //
        只要当前矩阵序列curpos与目标矩阵target不相同即执行while循环
213     {
214         for (int i = 1; i < 5; i++) // 向四周找方向
215         {
```

```
216             int tmp[N][N] = { 0 };
217             if (move(curpos, tmp, i)) //
                    依次按照上下左右顺序尝试移动空格
218             {
219                 int state = checkAdd(a, tmp, id);
220                 generatedNodes++; // 生成节点数加1
221
222                 if (state == -1) // 不添加到close表中
223                 {
224                     path[id] = Curid; //
                            走到当前第id个节点实际已经走过的路径的花费
225                     passLen[id] = passLen[Curid] + 1;
226                     fitness[id] = evalute1(tmp,
                            target) + passLen[id]; //
                            总花费估价
227                     statecpy(a, tmp, id++);
                                                    //
                            将处理得到的新矩阵编号为id复制到open表中a中保存
228                 }
229                 else // 添加到close表中
230                 {
231                     int len = passLen[Curid] + 1, fit
                            = evalute1(tmp, target) + len;
                            // 修改估价函数
232                     if (fit < fitness[state])
233                     {
234                         path[state] = Curid;
235                         passLen[state] = len;
236                         fitness[state] = fit;
237                         visited[state] = false;
238                     } //
                            若所得结果小于预期花费则修改多余部分的估价值，并将未
239                 }
```

```
240             }
241         }
242         visited[Curid] = true; //
                第curid个矩阵已被遍历过
243         expandedNodes++; // 扩展节点数加1
244
245         int minCur = -1;
246         for (int i = 0; i < id; i++) //
                从open表中(visited值为false)寻找总估价函数fitness值最小的矩阵作为
247             if (!visited[i] && (minCur == -1 ||
                    fitness[i] < fitness[minCur]))
248                 minCur = i;
249         Curid = minCur;                 //
                Curid现在表示被选作下一个扩展节点的矩阵的编号
250         getState(a, curpos, Curid); //
                将被选中的矩阵复制给curpos
251         if (id == maxState)
252             return -1; //
                    如果已经搜索节点数达到设定的maxState，则认为目标矩阵不可达，
253     }
254     std::cout << "Expanded nodes: " << expandedNodes
            << std::endl;
255     std::cout << "Generated nodes: " <<
            generatedNodes << std::endl;
256     return Curid; //
            已求得目标矩阵，返回最终矩阵的编号
257 }
258
259 int Astar2(int a[N][N][maxState], int start[N][N],
        int target[N][N], int path[maxState])
260 {
261     bool visited[maxState] = { false }; //
            true表示矩阵已被遍历
```

```
262     int fitness[maxState] = { 0 };
263     int passLen[maxState] = { 0 };
264     int curpos[N][N];
265     statecpy(curpos, start);
266     int id = 0, Curid = 0;
267     int expandedNodes = 0; // 扩展节点数
268     int generatedNodes = 1; //
            生成节点数，初始为1，因为起始节点已生成
269     fitness[id] = evalute2(curpos, target);
270     statecpy(a, start, id++);
271     while (!isEqual(curpos, target)) //
            只要当前矩阵序列curpos与目标矩阵target不相同即执行while循环
272     {
273         for (int i = 1; i < 5; i++) // 向四周找方向
274         {
275             int tmp[N][N] = { 0 };
276             if (move(curpos, tmp, i)) //
                    依次按照上下左右顺序尝试移动空格
277             {
278                 int state = checkAdd(a, tmp, id);
279                 generatedNodes++; // 生成节点数加1
280
281                 if (state == -1) // 不添加到close表中
282                 {
283                     path[id] = Curid; //
                            走到当前第id个节点实际已经走过的路径的花费
284                     passLen[id] = passLen[Curid] + 1;
285                     fitness[id] = evalute2(tmp,
                            target) + passLen[id]; //
                            总花费估价
286                     statecpy(a, tmp, id++);
                                                    //
                            将处理得到的新矩阵编号为id复制到open表中a中保存
```

```
287                 }
288                 else // 添加到close表中
289                 {
290                     int len = passLen[Curid] + 1, fit
                        = evalute2(tmp, target) + len;
                        // 修改估价函数
291                     if (fit < fitness[state])
292                     {
293                         path[state] = Curid;
294                         passLen[state] = len;
295                         fitness[state] = fit;
296                         visited[state] = false;
297                     } //
                        若所得结果小于预期花费则修改多余部分的估价值，并将未
298                 }
299             }
300         }
301         visited[Curid] = true; //
            第curid个矩阵已被遍历过
302         expandedNodes++; // 扩展节点数加1
303
304         int minCur = -1;
305         for (int i = 0; i < id; i++) //
            从open表中(visited值为false)寻找总估价函数fitness值最小的矩阵作为
306             if (!visited[i] && (minCur == -1 ||
                fitness[i] < fitness[minCur]))
307                 minCur = i;
308         Curid = minCur;                 //
            Curid现在表示被选作下一个扩展节点的矩阵的编号
309         getState(a, curpos, Curid); //
            将被选中的矩阵复制给curpos
310         if (id == maxState)
```

```
311             return -1; //
                    如果已经搜索节点数达到设定的maxState，则认为目标矩阵不可达，这
312     }
313     std::cout << "Expanded nodes: " << expandedNodes
            << std::endl;
314     std::cout << "Generated nodes: " <<
            generatedNodes << std::endl;
315     return Curid; //
            已求得目标矩阵，返回最终矩阵的编号
316 }
317
318 int BFS(int a[N][N][maxState], int start[N][N], int
        target[N][N], int path[maxState])
319 {
320     bool visited[maxState] = { false }; //
            true表示矩阵已被遍历
321     int fitness[maxState] = { 0 };
322     int passLen[maxState] = { 0 };
323     int curpos[N][N];
324     statecpy(curpos, start);
325     int id = 0, Curid = 0;
326     int expandedNodes = 0; // 扩展节点数
327     int generatedNodes = 1; //
            生成节点数，初始为1，因为起始节点已生成
328     statecpy(a, start, id++);
329     while (!isEqual(curpos, target)) //
            只要当前矩阵序列curpos与目标矩阵target不相同即执行while循环
330     {
331         for (int i = 1; i < 5; i++) // 向四周找方向
332         {
333             int tmp[N][N] = { 0 };
334             if (move(curpos, tmp, i)) //
                    依次按照上下左右顺序尝试移动空格
```

```
335             {
336                 int state = checkAdd(a, tmp, id);
337                 generatedNodes++; // 生成节点数加1
338 
339                 if (state == -1) // 不添加到close表中
340                 {
341                     path[id] = Curid; //
                        走到当前第id个节点实际已经走过的路径的花费
342                     passLen[id] = passLen[Curid] + 1;
343                     fitness[id] = passLen[id]; //
                        总花费估价
344                     statecpy(a, tmp, id++);     //
                        将处理得到的新矩阵编号为id复制到open表中a中保存
345                 }
346                 else // 添加到close表中
347                 {
348                     int len = passLen[Curid] + 1, fit
                        = len; // 修改估价函数
349                     if (fit < fitness[state])
350                     {
351                         path[state] = Curid;
352                         passLen[state] = len;
353                         fitness[state] = fit;
354                         visited[state] = false;
355                     } //
                        若所得结果小于预期花费则修改多余部分的估价值，并将未
356                 }
357             }
358         }
359         visited[Curid] = true; //
            第curid个矩阵已被遍历过
360         expandedNodes++; // 扩展节点数加1
361 
```

```
362         int minCur = -1;
363         for (int i = 0; i < id; i++) //
                从open表中(visited值为false)寻找总估价函数fitness值最小的矩阵作为
364             if (!visited[i] && (minCur == -1 ||
                    fitness[i] < fitness[minCur]))
365                 minCur = i;
366         Curid = minCur;                 //
                Curid现在表示被选作下一个扩展节点的矩阵的编号
367         getState(a, curpos, Curid); //
                将被选中的矩阵复制给curpos
368         if (id == maxState)
369             return -1; //
                    如果已经搜索节点数达到设定的maxState，则认为目标矩阵不可达，
370     }
371     std::cout << "Expanded nodes: " << expandedNodes
            << std::endl;
372     std::cout << "Generated nodes: " <<
            generatedNodes << std::endl;
373     return Curid; //
            已求得目标矩阵，返回最终矩阵的编号
374 }
375
376 void show(int a[N][N][maxState], int n)
377 {
378     cout << "------------------------------\n";
379     for (int i = 0; i < N; i++)
380     {
381         for (int j = 0; j < N; j++)
382         {
383             g_Map[j][i] = a[i][j][n];
384
385             cout << a[i][j][n] << " ";
386         }
```

```
387         cout << endl;
388     }
389     Draw();
390     Sleep(800);
391     cout << "------------------------------\n";
392 }
393
394 int calDe(int a[N][N])
395 {
396     int sum = 0;
397     for (int i = 0; i < N * N; i++)
398     {
399         for (int j = i + 1; j < N * N; j++)
400         {
401             int m, n, c, d;
402             m = i / N;
403             n = i % N;
404             c = j / N;
405             d = j % N;
406             if (a[c][d] == 0)
407                 continue;
408             if (a[m][n] > a[c][d])
409                 sum++;
410         }
411     }
412     return sum;
413 }
414
415 void autoGenerate(int a[N][N])
416 {
417     int maxMove = 50;              //设置步数上限
418     srand((unsigned)time(NULL)); //生成随机数种子
419     int tmp[N][N];
```

```
420     while (maxMove--)
421     {
            //随机移动空格五十步可以保证初始状态的矩阵序列不相同
422         int dir = rand() % 4 + 1;
                //dir取值范围1~4, 代表空格移动的四个方向
423         if (move(a, tmp, dir))
424             statecpy(a, tmp);
                    //打乱原目标矩阵的顺序以构造初始矩阵
425     }
426 }
427
428 void results_op(int res, int path[maxState], int
        a[N][N][maxState])
429 {
430
431     int shortest[maxState] = { 0 }, j = 0;
432     while (res != 0)
433     {
434         shortest[j++] = res;
435         res = path[res];
436     }
437     for (int i = j - 1; i >= 0; i--)
438     {
439         cout << "第 " << j - i << " 步\n";
440         show(a, shortest[i]);
441     }
442 }
443
444 void InitBlock()
445 {
446         // 初始化拼图碎片
447         wchar_t s[3];
448         for (int i = 0; i < N * N; i++)
```

```
449         {
450                 g_Block[i].Resize(100, 100);
451                 SetWorkingImage(&g_Block[i]);
452
453                 // 背景
454                 cleardevice();
455                 setfillcolor(HSVtoRGB(float(360.0 * i
                        / 9), 1, 0.5));
456                 solidrectangle(2, 2, 98, 98);
457                 // 文字
458                 settextstyle(64, 0, _T("Arial"), 0,
                        0, 400, false, false, false,
                        DEFAULT_CHARSET,
                        OUT_DEFAULT_PRECIS,
                        CLIP_DEFAULT_PRECIS,
                        PROOF_QUALITY, DEFAULT_PITCH);
459                 setbkmode(TRANSPARENT);
460                 settextcolor(WHITE);
461                 _itow_s(i, s, 10);
462                 outtextxy((100 - textwidth(s)) / 2,
                        18, s);
463         }
464         // 恢复绘图目标
465         SetWorkingImage(NULL);
466 }
467
468 void MoveTo(int newx, int newy)
469 {
470         g_Map[g_EmptyX][g_EmptyY] = g_Map[newx][newy];
471         g_Map[newx][newy] = N * N - 1;
472         g_EmptyX = newx;
473         g_EmptyY = newy;
474 }
```

```
475
476 void RandMap(int(*start)[N])
477 {
478     for (int i = 0; i < N; i++)
479     {
480         for (int j = 0; j < N; j++)
481         {
482             g_Map[i][j] = start[i][j];
483         }
484     }
485         g_EmptyX = N - 1;
486         g_EmptyY = N - 1;
487 }
```

5.2 test.cpp

```
1  #include <iostream>
2  #include <ctime>
3  #include <cstdlib>
4  #include "game.h"
5
6  int main()
7  {
8      HWND wnd = initgraph(400, 400);
9      InitBlock();
10     loadimage(NULL, BACKGROUND_IMAGE, 400, 400);
11     int a[N][N][maxState] = { 0 };
12     int start[N][N] = { 1, 2, 3, 8, 0, 4, 7, 6, 5 };
13     autoGenerate(start);
14     RandMap(start);
15     cout << "原始矩阵为: " << endl;
16     for (int i = 0; i < 3; i++)
17     {
```

```
18          for (int j = 0; j < 3; j++)
19          {
20              cout << start[i][j] << ' ';
21          }
22          cout << endl;
23      }
24      int target[N][N] = { 1, 2, 3, 8, 0, 4, 7, 6, 5 };
25      int start_demo[N][N] = { 0 }, target_demo[N][N] =
            { 0 };
26      statecpy(start_demo, start);
27      statecpy(target_demo, target);
28      if (!(calDe(start) % 2 == calDe(target) % 2))
29      {
30          cout <<
                "在此初始状态及目标序列情况下, 无解\n";
31          system("pause");
32          return 0;
33      }
34      int path[maxState] = { 0 };
35      // double begin = GetTickCount();
36      cout << "使用简单错排的A*算法处理如下: " << endl;
37      int res = Astar1(a, start, target, path);
38      if (res == -1)
39      {
40          cout << "此次搜索已经搜索超过" << maxState <<
                "个节点, 认为目标矩阵不可达\n";
41          system("pause");
42          return 0;
43      }
44      results_op(res, path, a);
45      Sleep(3000);
46      // cout << "运行时间" << GetTickCount() - begin
            << endl;
```

```
47
48      // begin = GetTickCount();
49      cout << "使用曼哈顿距离的A*算法处理如下：" <<
            endl;
50      for (int i = 0; i < maxState; i++)
51      {
52          path[0] = 0;
53      }
54      res = Astar2(a, start, target, path);
55      if (res == -1)
56      {
57          cout << "此次搜索已经搜索超过" << maxState <<
                "个节点，认为目标矩阵不可达\n";
58          system("pause");
59          return 0;
60      }
61      results_op(res, path, a);
62      Sleep(3000);
63      for (int i = 0; i < maxState; i++)
64      {
65          path[0] = 0;
66      }
67      // cout << "运行时间" << GetTickCount() - begin
            << endl;
68
69      // begin = GetTickCount();
70      cout <<
            "\n仅使用路径花费作为启发函数的A*算法(BFS)处理结果如下："
            << endl;
71      res = BFS(a, start_demo, target_demo, path);
72      if (res == -1)
73      {
```

```
74          cout << "此次搜索已经搜索超过" << maxState <<
                "个节点，认为目标矩阵不可达\n";
75
76          system("pause");
77          exit(0);
78      }
79      results_op(res, path, a);
80      // cout << "运行时间" << GetTickCount() - begin
            << endl;
81      system("pause");
82      return 0;
83  }
```