

# 中山大学计算机学院人工智能本科生实验报告（2022学年春季学期）

课程名称：Artificial Intelligence

教学班级	专业（方向）	学号	姓名
2班	计算机科学与技术	21307174	刘俊杰

## 一、实验题目

使用A\* 与 IDA\*算法解决15-Puzzle问题

## 二、实验内容

使用 A\* 与IDA\* 算法解决15-Puzzle问题，启发式函数可以自己选取，可以多尝试几种不同的启发式函数，完成ppt上的4个测例及压缩包的4个测例

### 结果分析要求

- 1. 对比分析 A\* 和 IDA\* 的性能和结果
- 2. 如果使用了多种启发式函数也可以进行对比和分析
- 3. 加分项（供参考）
  - 算法实现优化分析（使用数据结构、利用性质的剪枝等）
  - 未提及的启发式函数实现、对比和分析

### 1. 算法原理

- A\*算法

估价函数

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

算法描述

从起始状态开始，每次选取未进行拓展的状态中估价函数最小的状态进行拓展，不断搜索新的可达状态并计算它们的估价函数值，或更新待进行拓展的状态的 $g(x)$ ，对已进行拓展的状态进行剪枝，直到找到目标状态。

步骤

1. 从起始状态开始，把其当成待拓展状态存入一个“开启列表”
2. 从“开启列表”中找到估价函数最小的状态 $C$ ，并将它从开启列表中删除，添加到“关闭列表”中（列表中保存所有不需要再次搜索的状态）
3. 检查 $C$ 所有相邻状态，
  - (1)如果状态在关闭列表中，则剪枝；
  - (2)如果状态不在开启列表中，则计算 $h(C), g(C), f(C)$ ，将 $C$ 作为的“父状态”，并加入开启列表
  - (3)如果状态在开启列表中，则使用当前的 $g(C)$ 更新状态的原 $g(C)$ ，成功更新则同时更新 $f(C)$ 与父状态，否则剪枝
4. 重复步骤2, 3，直到找到目标状态（步骤2抽取的是目标状态）或者开启列表为空

- IDA\*算法

估价函数

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

算法描述

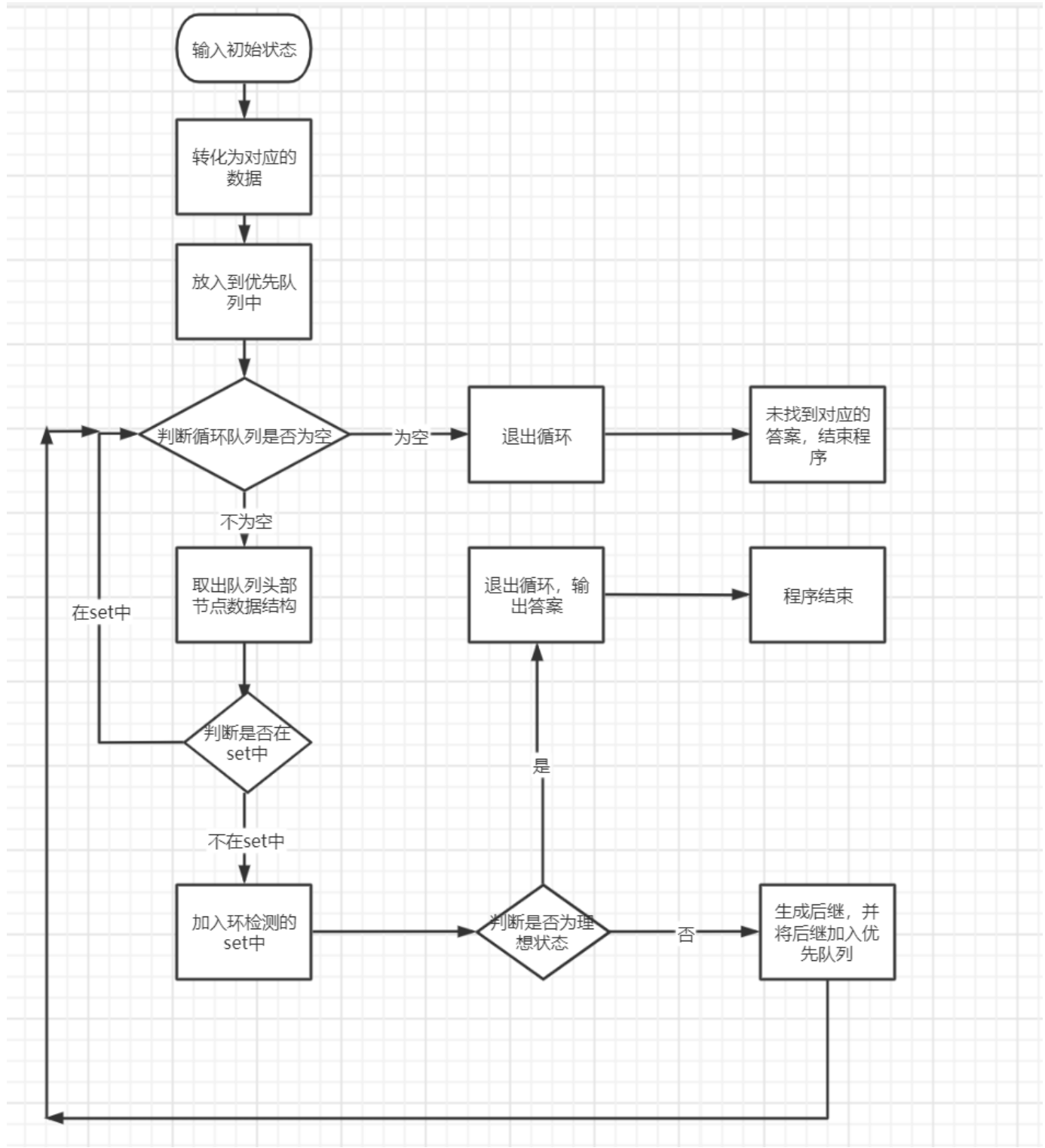
对估价函数值的阈值进行迭代，在迭代的每一步都进行深度优先搜索，优先选择相邻状态中估价函数值最小的状态进行递归，若当前状态的估价函数值大于当前的阈值，则进行剪枝，直到到达目标状态或无解，否则更新阈值继续迭代。

#### 步骤

1. 对估价函数值的阈值进行迭代增加，对于给定的一个阈值，定义递归过程
2. 从起始状态开始，计算所有相邻状态的估价函数值，选取估价函数最小的状态进行递归
  - (1) 如果当前状态的估价函数值大于阈值，则记录当前状态的估价函数值（以确定阈值的下一个迭代值），回溯
  - (2) 如果当前状态是目标状态，则记录答案，结束递归
  - (3) 如果递归找到目标状态，则算法结束
  - (4) 如果递归过程中所有到达状态的估价函数值均小于等于阈值，则目标状态无法到达，算法结束；
3. 根据递归过程中记录的超出阈值的估价函数值更新阈值（如最小值），再次进行步骤1的递归过程，直到找到目标状态，或无解

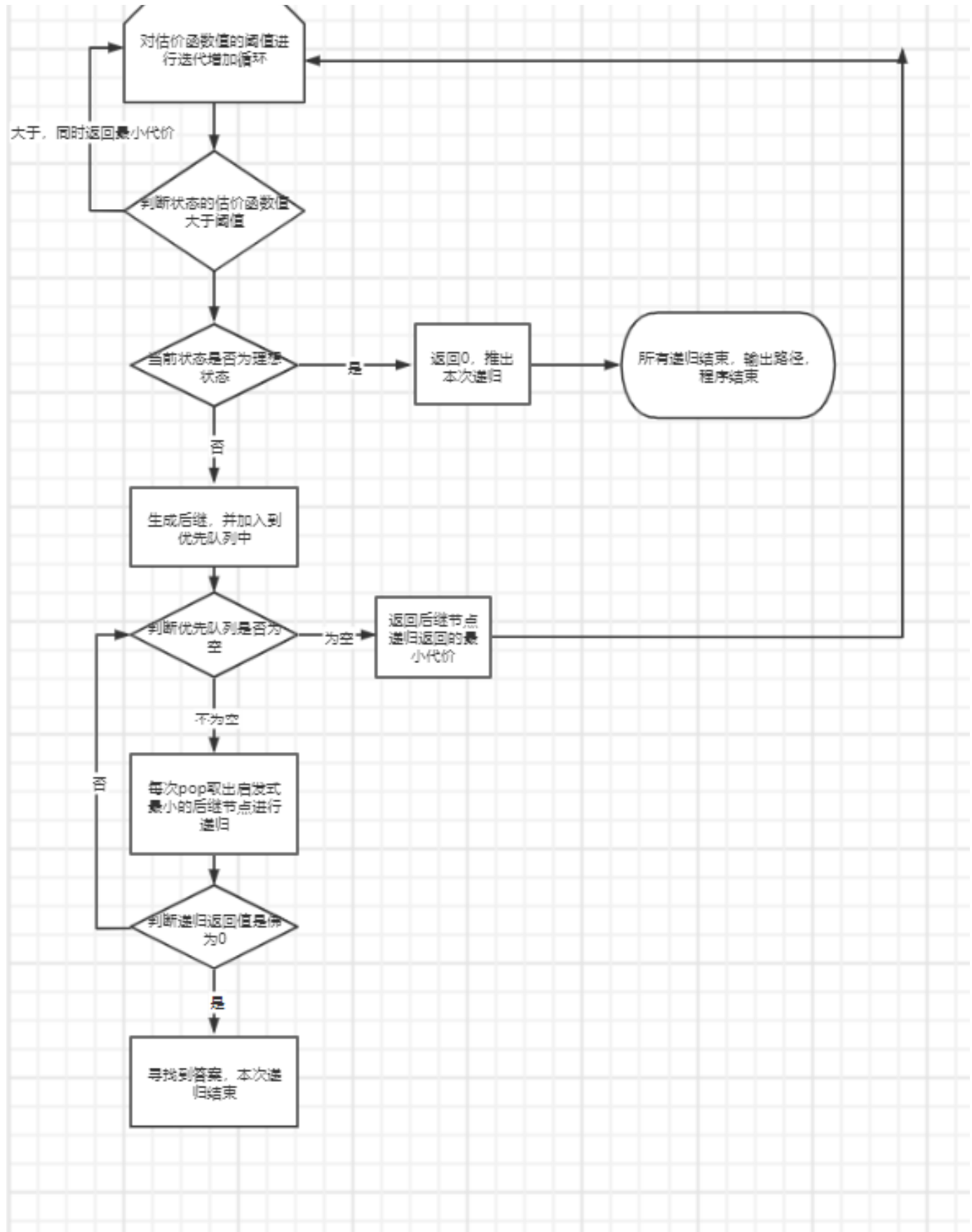
## 2.流程图

### A\*算法



IDA\*算法





3.关键代码展示

1. 创建数码表的类：数码表的状态state用元组储存,path记录移动路径，h、g分别记录h(n)和g(n),zero记录0(空)在元组中的下标，\_lt\_定义小于(代价函数相等时，以h(n)比较)，hn函数计算h(n).

```

class Puzzle:
    def __new__(cls, list):
        return super().__new__(cls)
    def __init__(self, list):
        self.state = tuple(list) # 数码表, 用元组来记录
        self.path = [] # 路径
        self.g = 0 # g
        self.h = 0 # h
        self.zero = self.state.index(0) # 记录元组中0的下标
        self.hn() # 计算h(n)

    def __lt__(self, other): # 优先队列中的小于比较 (代价函数相等时, 以h(n)比较)
        return self.g+self.h < other.g+other.h or (self.g+self.h == other.g+other.h and self.h <
other.h)
    def hn(self): # 求h(n) 曼哈顿距离
        self.h = 0
        for i in range(16):
            if self.state[i] != 0:
                self.h += abs((self.state[i]-1)%4-i%4)+abs(int(((self.state[i]-1)/4)-i//4))

```

**2.A\*算法:**将初始状态放入优先队列中, 每次取出估价函数最小的状态, 如果该状态在环检测的set中则剪枝, 否则加入环检测set中, 判断是否为目标状态, 是则退出输出答案, 否则将其后继加入优先队列中, 重复上述操作.

```

def A_star(p): # 环检测A*
    global sum
    pri = PriorityQueue()
    pri.put(p)
    visited = set() # set去重, 环检测
    while not pri.empty():
        puz = pri.get()
        #tmp.print_mat()
        if puz.state in visited: # 环检测
            continue

        if puz.h == 0: # 寻找到答案, 返回
            return puz
        sum+=1
        visited.add(puz.state) #
        #生成后继
        zero = puz.state.index(0)
        x = zero%4
        y = zero//4
        move = []
        if x > 0: move.append(-1) # 左移
        if y > 0: move.append(-4) # 上移
        if x < 3: move.append(1) # 右移
        if y < 3: move.append(4) # 下移
        for i in range(len(move)):
            l = list(puz.state)
            l[zero] = l[zero+move[i]]
            l[zero+move[i]] = 0
            puz_child = Puzzle(l)
            puz_child.g = puz.g+1
            puz_child.path = copy.deepcopy(puz.path)
            puz_child.path.append(l[zero])
            pri.put(puz_child)

```

**3.IDA\*算法:**对估价函数值的阈值进行迭代,在迭代的每一步都进行深度优先搜索,优先选择相邻状态中估价函数值最小的状态进行递归,若当前状态的估价函数值大于当前的阈值,则进行剪枝,直到到达目标状态或无解,否则更新阈值继续迭代,同时要路径检测。

```
def IDA_start(path,puz):#列表环检测的IDA_star
    max_limit = 10000
    i = 0
    while i < max_limit:#对阈值进行迭代
        visited=set()
        ans= dfs(puz,visited,i,path)
        if ans == 0:#找到目标状态退出
            return True
        elif ans > i:#若返回最小代价大于当前阈值,就对阈值进行更新
            i = ans
        else:
            i+=1

def dfs(p,visited,limit,path):#深度优先搜索
    res = 9999#最小代价
    visited.add(p.state)#路径检测
    Pri=PriorityQueue()
    if(p.h + p.g > limit):
        return p.h + p.g#返回代价
    if p.h == 0:
        return 0#找到目标, 返回
    #生成后继
    zero = p.zero
    x = zero%4
    y = zero//4
    move = []
    if x > 0:move.append(-1)#左移
    if y > 0:move.append(-4)#上移
    if x < 3:move.append(1)#右移
    if y < 3:move.append(4)#下移
    for i in range(len(move)):
        l = list(p.state)
        l[zero] = l[zero+move[i]]
        l[zero+move[i]] = 0
        puz_child = Puzzle(l)
        puz_child.g = p.g+1
        if puz_child.state in visited:#路径检测
            continue
        Pri.put(puz_child)#

    while not Pri.empty():#每次取估计函数最小的状态
        puz = Pri.get()
        path.append(puz.state[zero])
        ans = dfs(puz,visited,limit,path)
        if ans > 0 and ans < res:#计算所有相邻状态的估价函数值, 选取估价函数最小的状态进行递归
            res = ans
        if ans == 0:
            return 0 #找到答案,返回
        else:
            visited.remove(puz.state)
            path.pop()
        if res > ans:
            res = ans
    return res#返回最小代价
```

#### 4.创新点&优化

## 1. 环检测方面：使用了set() 数码表状态用元组存储

一开始在做环检测时,我使用的是字典和用列表存储数码表状态,但元组在存储空间上显得更加轻量级一些。对于元组这样的静态变量,如果它不被使用并且占用空间不大时,Python 会暂时缓存这部分内存。这样,下次我们再创建同样大小的元组时,Python 就可以不用再向操作系统发出请求,去寻找内存,而是可以直接分配之前缓存的内存空间,这样就能大大加快程序的运行速度。

```
#生成后继
zero = puz.state.index(0)
x = zero%4
y = zero//4
move = []
if x > 0:move.append(-1)#左移
if y > 0:move.append(-4)#上移
if x < 3:move.append(1)#右移
if y < 3:move.append(4)#下移
for i in range(len(move)):
    l = list(puz.state)
    l[zero] = l[zero+move[i]]
    l[zero+move[i]] = 0
    puz_child = Puzzle(l)
    puz_child.g = puz.g+1
```

## 2. 使用了内嵌函数：使用了tuple()、index()等内嵌函数

使用第一点后运行样例优化效果不大,与同学讨论后发现自己手动实现列表到元组的转换和寻找0的下标耗费时间,故使用内置函数完成,的确时间缩短了很多。

## 3. 环检测细节：

优化前,A\*算法的环检测中扩展状态和后继状态加入优先队列时都进行了环检测,其实只要在扩展时进行就好了,否则对保存状态多的环检测将耗费过多的时间。

## 4. 优先队列细节：\_\_lt\_\_定义小于(代价函数相等时,以h(n)比较), hn函数计算h(n).

这样在优先队列中代价函数相等时, h(n)较小的先计算,这对步数较多的到后面的比较有优势。

```
def __lt__(self, other):#优先队列中的小于比较
    return self.g+self.h < other.g+other.h or (self.g+self.h == other.g+other.h and self.h < other.h)
    def hn(self):#求h(n)
```

## 5.用时间换空间



对PPT中最长的两个样例,运行 A\* 算法时会出现内存超限的问题

```
Traceback (most recent call last):
  File "c:\Users\刘俊杰\Desktop\code\ai\E4\e4_tuple.py", line 175, in <module>
    main()
  File "c:\Users\刘俊杰\Desktop\code\ai\E4\e4_tuple.py", line 133, in main
    ans = A_star(a)
  File "c:\Users\刘俊杰\Desktop\code\ai\E4\e4_tuple.py", line 62, in A_star
    pri.put(puz_child)
  File "C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\lib\queue.py", line 150, in put
    self._put(item)
  File "C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\lib\queue.py", line 236, in _put
    heappush(self.queue, item)
MemoryError
```

所以对于56步的例子我们避免优先队列中有重复的状态，所以在加入优先队列时先使用环去重，检测这会增加环检测的次数，造成时间的损耗，但最后使56步可以正确执行 A\* 算法得出正确结果。

```
if puz_child.state in visited:
    continue
Pri.put(puz_child)
```

但是尽管使用了这种优化也需要许多的内存，还要进行读写磁盘操作，这使得运行时间变慢。

名称	状态	33% CPU	97% 内存	27% 磁盘	0% 网络
应用 (5)					
>  Microsoft Edge (2)		0.2%	52.9 MB	1.9 MB/秒	0 Mbps
>  Visual Studio Code (19)		8.2%	10,991.6 ...	60.2 MB/...	0 Mbps

所以需要下面一步的优化

6.使用改善后的启发式函数

对于最长的62步的样例使用 A\* 算法内存不够 所以与同学讨论后，我们使用改善后的启发式函数 $f=g+1.15h$ ,这样的启发式函数，这样在后面搜索时h对f的影响不会过小。

```
def __lt__(self, other):#优先队列中的小于比较
    return self.g+1.15*self.h < other.g+1.15*other.h or (self.g+1.15*self.h ==
other.g+1.15*other.h and self.h < other.h)
def hn(self):#求h(n)
```

使用这样的启发式函数，搜索的状态数目减少，可以运行出62步的正确结果 缺陷是证明这样的启发式函数的可采纳性和单调性很困难。

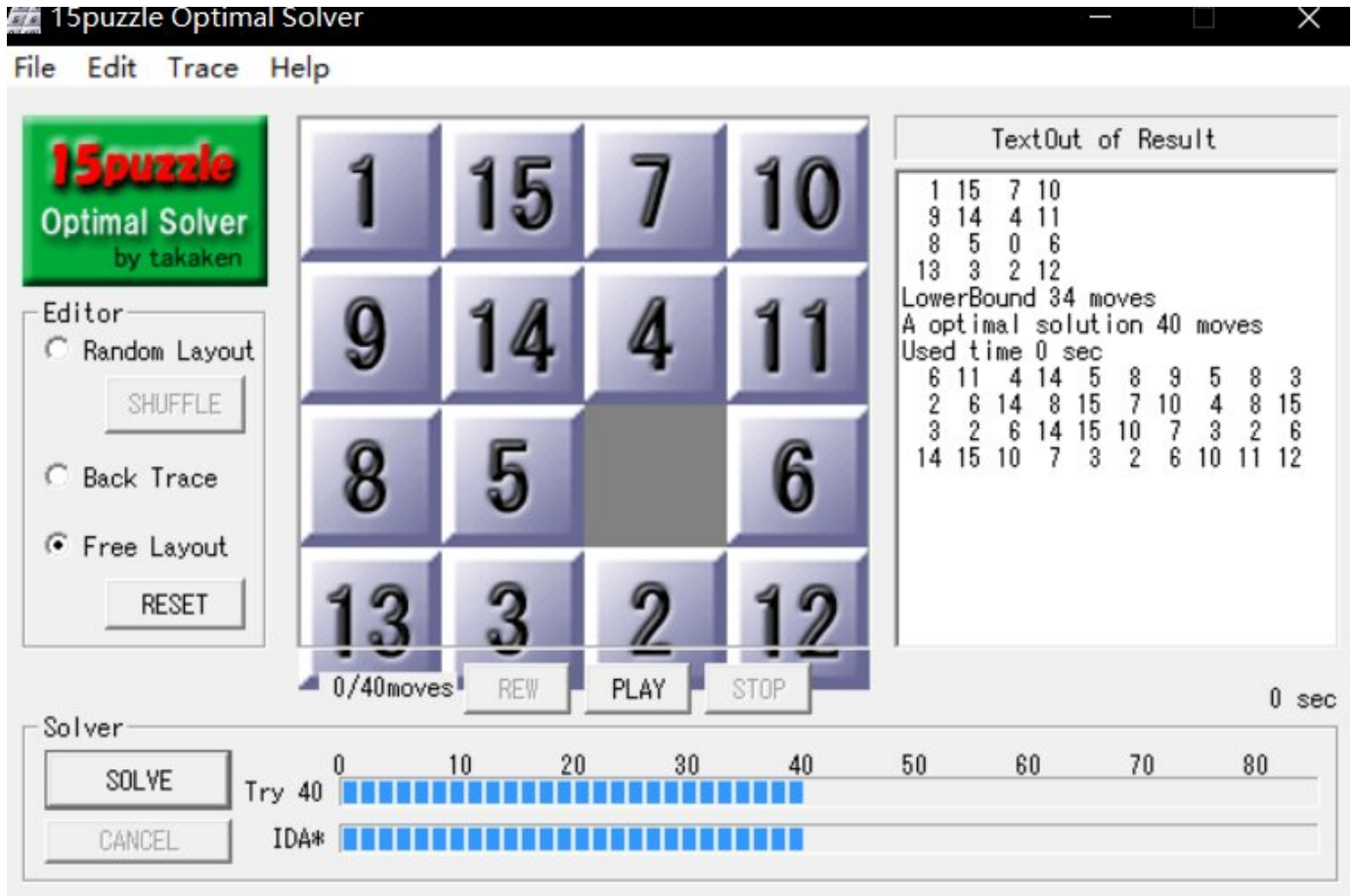
三、实验结果及分析

1. 实验结果展示示例(优化后的)

(实验报告中展示时截图，结果展示在result文件夹中)

- 1. 压缩包样例一





```
PS C:\Users\刘俊杰\Desktop\code\ai> C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\python.exe "c:\Users\刘俊杰\Desktop\code\ai\E4\e4_tuple.py"
Finding 3480 nodes
[6, 11, 4, 14, 5, 8, 9, 5, 8, 3, 2, 6, 14, 8, 15, 7, 10, 4, 8, 15, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 10, 11, 12]
40
A*: time cost 0.47314953804016113 s
[6, 11, 4, 14, 5, 8, 9, 5, 8, 3, 2, 6, 14, 8, 15, 7, 10, 4, 8, 15, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 14, 15, 10, 7, 3, 2, 6, 10, 11, 12]
40
IDA*: time cost 0.778179407119751 s
```

2. 压缩包样例二

15puzzle

Optimal Solver

by takaken

Editor

☐ Random Layout

SHUFFLE

☐ Back Trace

☒ Free Layout

RESET

17810

691514

1334

115122

0/40moves

REWPLAYSTOP

TextOut of Result

1 7 8 10  
6 9 15 14  
13 3 0 4  
11 5 12 2  
LowerBound 38 moves  
A optimal solution 40 moves  
Used time 0 sec  
15 14 4 2 12 15 14 8 10 4  
8 9 3 5 11 13 5 14 2 12  
15 11 14 2 9 10 7 3 2 9  
10 7 3 2 6 5 9 10 11 15

Solver

SOLVE

CANCEL

Try 40

IDA\*

01020304050607080

Finding 1390 nodes

[15, 14, 4, 2, 12, 15, 14, 8, 10, 4, 8, 9, 3, 5, 11, 13, 5, 14, 2, 12, 15, 11, 14, 2, 9, 10, 7, 3, 2, 9, 10, 7, 3, 2, 6, 5, 9, 10, 11, 15]

40

A\*: time cost 0.16642498970031738 s

[15, 14, 4, 2, 12, 15, 14, 8, 10, 4, 8, 9, 3, 5, 11, 13, 5, 14, 2, 12, 15, 11, 14, 2, 9, 10, 7, 3, 2, 9, 10, 7, 3, 2, 6, 5, 9, 10, 11, 15]

40

IDA\*: time cost 0.6845748424530029 s

3. 压缩包样例三

15puzzle  
Optimal Solver  
by takaken

Editor

☐ Random Layout

SHUFFLE

☐ Back Trace

☒ Free Layout

RESET

56412

111491

107213

0/40moves

REW

PLAY

STOP

Solver

SOLVE

CANCEL

Try 40

IDA\*

01020304050607080

TextOut of Result

5 6 4 12  
11 14 9 1  
0 3 8 15  
10 7 2 13  
LowerBound 38 moves  
A optimal solution 40 moves  
Used time 0 sec  
11 14 9 1 12 4 1 8 2 13  
15 12 8 2 3 11 14 9 6 1  
2 3 11 7 13 11 7 14 10 13  
14 10 9 5 1 2 3 7 11 15

```
PS C:\Users\刘俊杰\Desktop\code\ai> C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\python.exe "c:\Users\刘俊杰\Desktop\code\ai\E4\e4_tuple.py"
Finding 557 nodes
[11, 14, 9, 1, 12, 4, 1, 8, 2, 13, 15, 12, 8, 2, 3, 11, 14, 9, 6, 1, 2, 3, 11, 7, 13, 11, 7, 14, 10, 13, 14, 10, 9, 5, 1, 2, 3, 7, 11, 15]
40
A*: time cost 0.06250381469726562 s
[11, 14, 9, 1, 12, 4, 1, 8, 2, 13, 15, 12, 8, 2, 3, 11, 14, 9, 6, 1, 2, 3, 11, 7, 13, 11, 7, 14, 10, 13, 14, 10, 9, 5, 1, 2, 3, 7, 11, 15]
40
IDA*: time cost 0.10638952255249023 s
```

4. 压缩包样例四

15puzzle  
Optimal Solver  
by takaken

Editor

Random Layout

SHUFFLE

Back Trace

RESET

Free Layout

RESET

14281

7104

615115

931312

0/40moves

REW

PLAY

STOP

TextOut of Result

14 2 8 1

7 10 4 0

6 15 11 5

9 3 13 12

LowerBound 34 moves

A optimal solution 40 moves

Used time 0 sec

1 8 4 1 5 11 15 3 13 15

3 10 1 5 8 4 2 1 5 3

10 5 7 14 1 2 3 7 5 6

14 5 6 14 9 13 14 10 11 12

Solver

SOLVE

CANCEL

Try 40

IDA\*

01020304050607080

0 sec

PS C:\Users\刘俊杰\Desktop\code\ai> C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\python.exe "c:\Users\刘俊杰\Desktop\code\ai\E4\e4\_tuple.py"

Finding 11829 nodes

[1, 8, 4, 1, 5, 11, 15, 3, 13, 15, 3, 10, 1, 5, 8, 4, 2, 1, 7, 14, 1, 7, 5, 3, 10, 6, 14, 5, 7, 2, 3, 7, 6,

14, 9, 13, 14, 10, 11, 12]

40

A\*: time cost 1.389331340789795 s

[1, 8, 4, 1, 5, 11, 15, 3, 13, 15, 3, 10, 1, 5, 8, 4, 2, 1, 7, 14, 1, 7, 5, 3, 10, 6, 14, 5, 7, 2, 3, 7, 6,

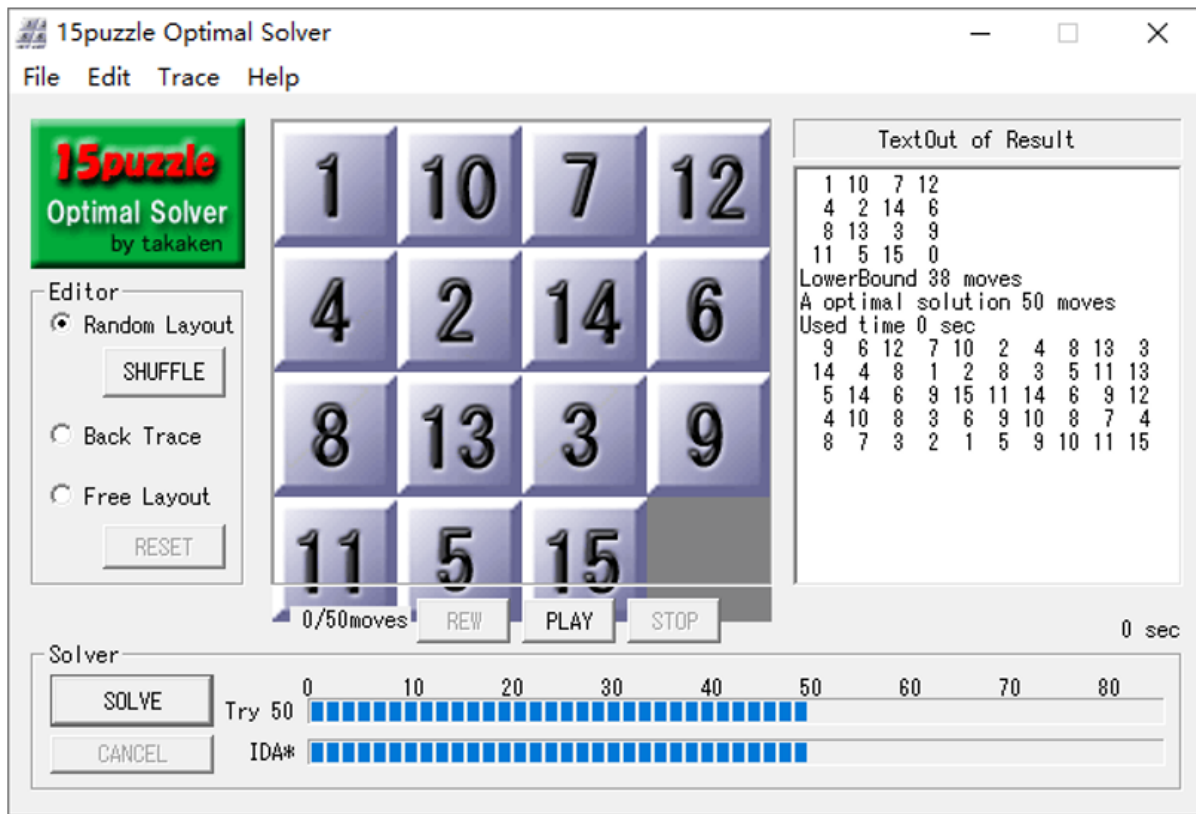
14, 9, 13, 14, 10, 11, 12]

40

IDA\*: time cost 8.177341938018799 s

5. PPT开头样例

12 / 17



```
Finding 799732 nodes
[15, 3, 9, 6, 12, 7, 10, 2, 4, 8, 13, 5, 11, 13, 5, 9, 14, 4, 8, 1, 2, 10, 4, 8, 10, 2, 1, 5, 9, 14, 6, 12,
8, 6, 3, 11, 14, 10, 6, 3, 11, 15, 12, 8, 7, 4, 3, 7, 8, 12]
50
A*: time cost 78.19264936447144 s
[15, 3, 14, 6, 12, 7, 10, 2, 4, 8, 13, 5, 11, 13, 5, 14, 3, 11, 14, 3, 6, 4, 8, 1, 2, 8, 3, 6, 9, 12, 4, 10,
, 8, 3, 6, 9, 10, 8, 7, 4, 8, 7, 3, 2, 1, 5, 9, 10, 11, 15]
50
IDA*: time cost 243.28965044021606 s
```

运行结果与PPT上不同，但验证后发现动作序列同样是正确的 验证程序

```
import math
a = [1,10,7,12,4,2,14,6,8,13,3,9,11,5,15,0]
path=[15, 3, 9, 6, 12, 7, 10, 2, 4, 8, 13, 5, 11, 13, 5, 9, 14, 4, 8, 1, 2, 10, 4, 8, 10, 2, 1, 5,
9, 14, 6, 12, 8, 6, 3, 11, 14, 10, 6, 3, 11, 15, 12, 8, 7, 4, 3, 7, 8, 12]
for move in path:
    zero_index=a.index(0)
    index=a.index(move)
    if abs(index-zero_index)!=1 and abs(index-zero_index)!=4:
        print("False")
        break
    a[zero_index]=a[index]
    a[index]=0
print(a)
```

验证结果：



```
PS C:\Users\刘俊杰\Desktop\code\ai> C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\python.exe "c:\Users\刘俊杰\Desktop\code\ai\test.py"
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]
PS C:\Users\刘俊杰\Desktop\code\ai>
```

## 6. PPT样例一

11 3 1 7  
4 6 8 2  
15 9 10 13  
14 12 5 0

TextOut of Result			
11	3	1	7
4	6	8	2
15	9	10	13
14	12	5	0
LowerBound 36 moves			
A optimal solution 56 moves			
Used time 3 sec			
13	10	8	6
12	15	14	5
14	9	4	11
1	6	4	2
7	4	2	11
6	2	3	7
5	13	10	8
12	15	14	5
13	10	8	6
14	9	4	11
1	6	4	2
7	4	2	11
6	2	3	7

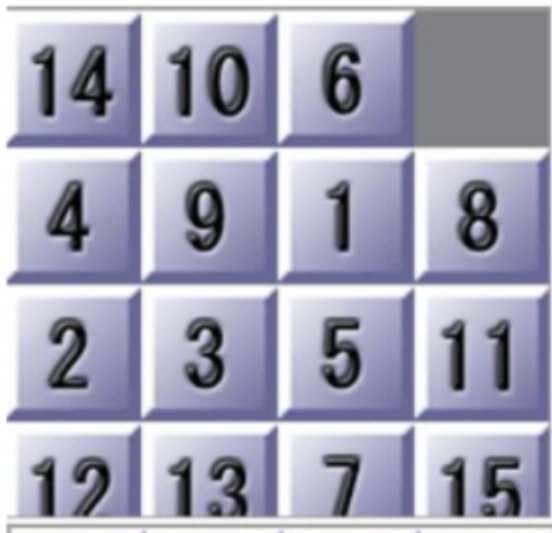
A\*算法:

```
PS C:\Users\刘俊杰\Desktop\code\ai> C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\python.exe "c:\Users\刘俊杰\Desktop\code\ai\E4\e4_tuple.py"
[13, 10, 8, 6, 4, 11, 3, 1, 6, 4, 9, 12, 5, 13, 10, 8, 12, 15, 14, 5, 13, 12, 15, 14, 5, 13, 14, 9, 11, 3, 1, 6, 4, 2, 8, 10, 12, 15, 10, 8, 7, 4, 2, 11, 3, 5, 9, 10, 11, 3, 6, 2, 3, 7, 8, 12]
56
IDA*: time cost 20110.355590343475 s
PS C:\Users\刘俊杰\Desktop\code\ai>
```

IDA\*算法:

```
PS C:\Users\刘俊杰\Desktop\code\ai> C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\python.exe "c:\Users\刘俊杰\Desktop\code\ai\E4\e4_tuple.py"
[13, 10, 8, 6, 4, 11, 3, 1, 6, 4, 9, 12, 5, 13, 10, 8, 12, 15, 14, 5, 13, 12, 15, 14, 5, 13, 14, 9, 11, 3, 1, 6, 4, 2, 8, 10, 12, 15, 10, 8, 7, 4, 2, 11, 3, 5, 9, 10, 11, 3, 6, 2, 3, 7, 8, 12]
56
IDA*: time cost 20110.355590343475 s
PS C:\Users\刘俊杰\Desktop\code\ai>
```

7. PPT样例二



TextOut of Result


14 10 6 0  
4 9 1 8  
2 3 5 11  
12 13 7 15  
LowerBound 37 moves  
A optimal solution 49 moves  
Used time 0 sec  
6 10 9 4 14 9 4 1 10 4  
1 3 2 14 9 1 3 2 5 11  
8 6 4 3 2 5 13 12 14 13  
12 7 11 12 7 14 13 9 5 10  
6 8 12 7 10 6 7 11 15

14 10 6 0  
4 9 1 8  
2 3 5 11  
12 13 7 15

```
PS C:\Users\刘俊杰\Desktop\code\ai> C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\python.exe "c:\Users\刘俊杰\Desktop\code\ai\E4\e4_tuple.py"
Finding 435953 nodes
[6, 10, 9, 4, 14, 9, 4, 1, 10, 4, 1, 3, 2, 14, 9, 1, 3, 2, 5, 11, 8, 6, 4, 3, 2, 5, 13, 12, 14, 13, 12, 7, 11, 12, 7, 14, 13, 9, 5, 10, 6, 8, 12, 7, 10, 6, 7, 11, 15]
49
A*: time cost 43.0419979095459 s
[6, 10, 9, 4, 14, 9, 4, 1, 10, 4, 1, 3, 2, 14, 9, 1, 3, 2, 5, 11, 8, 6, 4, 3, 2, 5, 13, 12, 14, 13, 12, 7, 11, 12, 7, 14, 13, 9, 5, 10, 6, 8, 12, 7, 10, 6, 7, 11, 15]
49
IDA*: time cost 245.15968203544617 s
```

8. PPT样例三

0 5 15 14  
7 9 6 13  
1 2 12 10  
8 11 4 3



TextOut of Result

0 5 15 14  
7 9 6 13  
1 2 12 10  
8 11 4 3  
LowerBound 44 moves  
A optimal solution 62 moves  
Used time 4 sec  
7 9 2 1 9 2 5 7 2 5  
1 11 8 9 5 1 6 12 10 3  
4 8 11 10 12 13 3 4 8 12  
13 15 14 3 4 8 12 13 15 14  
7 2 1 5 10 11 13 15 14 7  
3 4 8 12 15 14 11 10 9 13  
14 15


A\*算法：使用了f=g+1.5h的启发式函数

```
PS C:\Users\刘俊杰\Desktop\code\ai> C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\python.exe "c:\Users\刘俊杰\Desktop\code\ai\E4\e4_tuple.py"
Finding 4118629 nodes
[7, 9, 2, 1, 9, 2, 5, 7, 2, 5, 1, 11, 8, 9, 5, 1, 6, 12, 10, 3, 4, 8, 11, 10, 12, 13, 3, 4, 8, 12, 13, 15, 14, 3, 4, 8, 12, 13, 15, 14, 7, 2, 1, 5, 10, 11, 13, 15, 14, 7, 3, 4, 8, 12, 15, 14, 11, 10, 9, 13, 14, 15]
62
A*: time cost 8537.527533054352 s
PS C:\Users\刘俊杰\Desktop\code\ai>
```

IDA\*算法：

```
7 9 2 1 9 2 5 7 2 5 1 11 8 9 5 1 6 12 10 3 4 8
62
IDA*: time cost 42842.5228061676s
```

9. PPT样例四



TextOut of Result

```
6 10 3 15
14 8 7 11
5 1 0 2
13 12 9 4
LowerBound 32 moves
A optimal solution 48 moves
Used time 0 sec
9 12 13 5 1 9 7 11 2 4
12 13 9 7 11 2 15 3 2 15
4 11 15 8 14 1 5 9 13 15
7 14 10 6 1 5 9 13 14 10
6 2 3 4 8 7 11 12
```

```
6 10 3 15
14 8 7 11
5 1 0 2
13 12 9 4
```

```
PS C:\Users\刘俊杰\Desktop\code\ai> C:\Users\刘俊杰\AppData\Local\Programs\Python\Python39\python.exe "c:\Users\刘俊杰\Desktop\code\ai\E4\e4_tuple.py"
Finding 2494136 nodes
[9, 12, 13, 5, 1, 9, 7, 11, 2, 4, 12, 13, 9, 7, 11, 2, 15, 3, 2, 15, 4, 11, 15, 8, 14, 1, 5, 9, 13, 15, 7, 14, 10, 6, 1, 5, 9, 13, 14, 10, 6, 2, 3, 4, 8, 7, 11, 12]
48
A*: time cost 255.44039845466614 s
[9, 12, 13, 5, 1, 9, 7, 11, 2, 4, 12, 13, 9, 7, 11, 2, 15, 3, 2, 15, 4, 11, 15, 8, 14, 1, 5, 9, 13, 15, 7, 14, 10, 6, 1, 5, 9, 13, 14, 10, 6, 2, 3, 4, 8, 7, 11, 12]
48
IDA*: time cost 465.87745571136475 s
```

2. 评测指标展示及分析

实例	答案步数	未优化 A* 算法运行时间	优化后 A* 算法运行时间	A* 搜索节点个数	优化前 IDA* 算法运行时间	优化后 IDA* 算法运行时间
压缩包1	40	1.4720518589019775s	0.45366954803466797s	3480	3.350346088409424s	0.7846040725708008s
压缩包2	40	0.6634604930877686s	0.16642498970031738s	1390	0.9972481727600098s	0.6845748424530029s
压缩包3	40	0.24857878684997559	0.06250381469726562s	557	0.3535940647125244s	0.10638952255249023s
压缩包4	40	5.214388847351074s	1.389331340789795s	11829	8.199525356292725s	8.177341938018799s
PPT(开头样例)	50	257.0468544960022s	78.19264936447144s	799732	597.9593591690063s	243.28965044021606s



实例	答案步数	未优化 A* 算法运行时间	优化后 A* 算法运行时间	A* 搜索节点个数	优化前 IDA* 算法运行时间	优化后 IDA* 算法运行时间
PPT1	56	内存超出限制	5007.7800552845s	28838241	运行时间过长，未运行出结果	20110.355590343475s
PPT2	49	132.3359088897705s	43.0419979095459s	435953	507.0727288722992s	245.15968203544617s
PPT3	62	内存超出限制	8537.527533054352s	4118629	运行时间过长，未运行出结果	42842.5228061676s
PPT4	48	249.5972819328308s	255.44039845466614s	2494136	2672.5074956417084s	465.87745571136475s

分析

1. PPT样例3(62步的样例)时使用改善后的启发式函数，用原来的启发式函数内存超出限制，使用改善后的启发式函数会优先搜索更靠近目标的状态，所以搜索保存的状态只有4118629个，远远小于原本可能需要的几千万个状态。

2. 通过实验结果分析可以得出

A\* 算法：

优点：

与广度优先搜索策略和深度优先搜索策略相比，A\*算法不是盲目搜索，而是有提示的搜索。

缺点：

该算法一般要使用大量的空间用于存储已搜索过的中间状态，防止重复搜索。

IDA\*算法：

优点：

使用回溯方法，不用保存中间状态，大大节省了空间，在一定的样例中IDA\*的运行速度可能比A\*快。

缺点：

重复搜索：回溯过程中每次depth变大都要再次从头搜索。

四、思考题

1. 压缩包和PPT的样例全都运行成功且得出正确结果，见上文"三、实验结果及分析"。
2. 算法实现优化分析（使用数据结构、利用性质的剪枝等）见上文"4.创新点&优化"。
- 3.未提及的启发式函数实现、对比和分析
- (1)使用 **错位排列**的启发式函数，运行简单的样例都需要几十分钟的时间，稍微长一些的就运行时间太长、运行不出结果。(2)使用 **曼哈顿**的启发式函数，运行时间远远快于**错位排列**，对除了62步的A外都能运行中都能取得正确的结果。(3)使用同学分享的  **$f=g+1.15h$**  的启发式函数，速度会进一步加快运行速度，对62步的A搜索的状态更少，使得要占用的空间更少，且速度也更快。(4)总结来说，更好的启发式函数，能使得我们选择搜索更好的选择，这样使得运行的时间和需要保存的空间都有更大的优化，但是对更好的启发式函数的可采纳性和单调性的证明会更加复杂。

五、参考资料

- 理论课和实验课课件
- 实验课课上同学分享
- [https://blog.csdn.net/m0\\_52387305/article/details/123531945?spm=1001.2014.3001.5506](https://blog.csdn.net/m0_52387305/article/details/123531945?spm=1001.2014.3001.5506)