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
计科三班	计算机科学与技术	21307185	张礼贤

一、实验题目

使用A_Star算法和IDA_Star算法解决十五数码问题

二、实验内容

1、算法原理

本实验主要针对两种启发式搜索A_Star和IDA_Star算法的实现。本实验使用了启发信息来引导搜索结果，可以降低搜索的盲目性和搜索的状态空间，可以达到优化搜索的目的。算法的设计分为两个大板块，一个是启发式函数的设计，另一个是具体算法的实现。

· 启发式函数的设计

启发式函数的设计需要满足两个特征：

1、**可采纳的：** 即h(n) <= h*(n)，其中h(n)表示到终点代价的预估值，为启发函数给出；h*(n)表示到终点的实际代价。

2、**单调的：** 即h(n) <= h(n') + cost(n',n)

在本实验中，由于数码可以向上下左右四个方向进行移动，因此选择曼哈顿距离会比较方便计算，并且更加接近真实状态.

易错点：在计算曼哈顿距离的时候，不能把0也算进去。如果计算，根据最后一步，h(n) = 2 , h'(n) = 1不满足可采纳性，因此不能将0纳入计算范围

·A_Star算法实现原理

A*算法相当于改进后的bfs算法，在此基础上加入了启发式函数进行信息引导。但是open表使用优先队列进行存储，每次从优先队列中弹出 `f(x) = g(x) + h(x)` 值最小的节点并进行扩展，检测扩展子节点的边界状态是否越界(即上下左右移动是否超出范围)。并对子节点进行环检测，即close表中是否有子节点，再将子节点生成，父节点记录为扩展而来的节点，并加入OPEN表中，重复以上操作，直到找到相应的路径返回。最后通过回溯将路径打印。

·IDA_Star算法实现原理

IDA_Star算法是对迭代加深算法的改进，通过不断对bound(即边界)进行迭代加深，但是这里的深度不同于迭代加深的树的高度，而是**f(x)的值**。之后反复进行dfs算法的复用，直至找到一条最优路径 **(因为环检测保最优性)**

2、伪代码及流程图：

1、A_Star算法：

·开始

1、将初始节点加入open表中

·循环

1、将open表中f(x)值最小的节点弹出

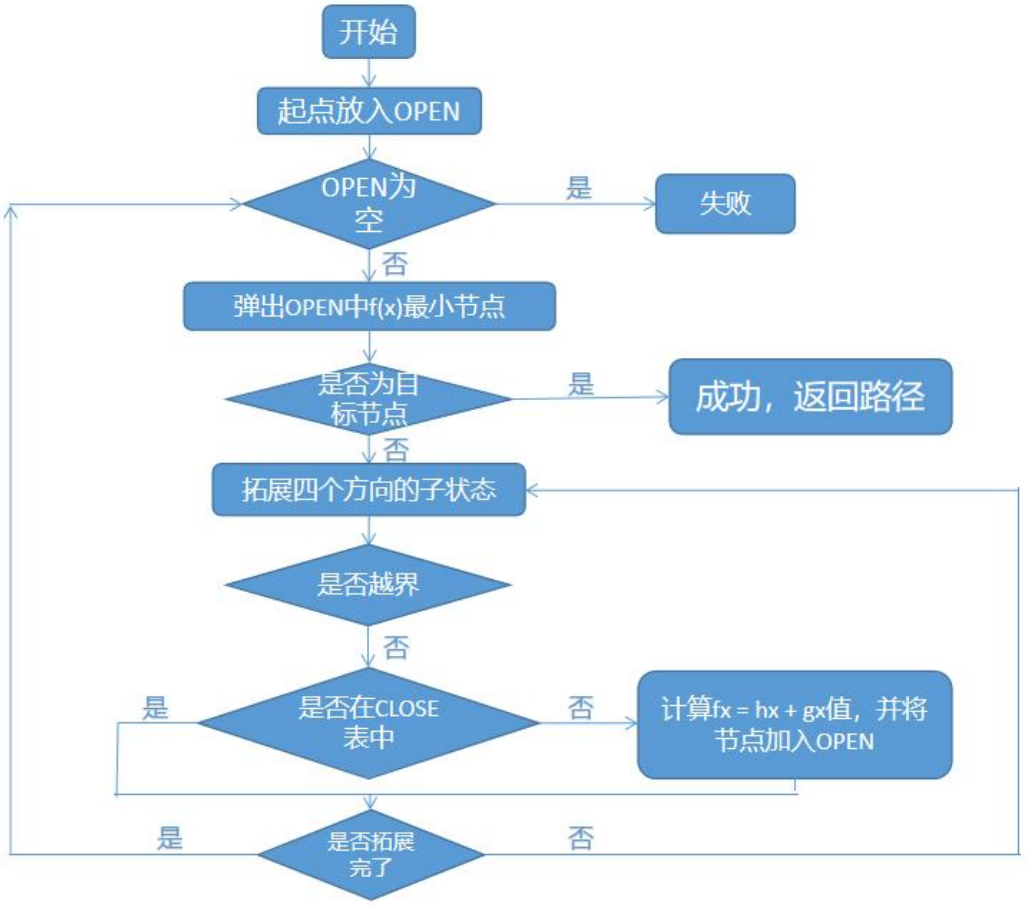
2、对弹出节点进行环检测，如果在close表中，则跳过，如果不在则加入close中

3、扩展弹出节点的子状态，将子状态加入open表中

·结束

1、搜索到相匹配的节点的时候，即返回节点（实现的时候是利用父节点记录并进行回溯找到路径）

流程图：



2、IDA_Star算法：

·开始

1、将初始bound边界值设置为初始起点的f(x)值，从根节点进行搜索

·循环

1、进入dfs函数

2、弹出**path** 列表的最后一个元素，将它作为扩展结点

3、对于节点f(x)超过bound边界的情况，即返回当前的f(x)值，如果找到了相匹配的节点状态，则**返回0**进行标识

4、如果没有超过边界则进行环检测，并将节点加入**path** 继续进行递归搜索

5、设置一个res值记录在边界上的所有节点的最小f(x)值(这里是通过递归返回的值确定的)

6、撤销之前的操作，恢复原来进行递归的状态

7、返回res(这里是从dfs函数返回出来)

8、如果找到路径了，即break

8、更新bound值

3、关键代码及注释:

A_Star算法(优化前)

```
import numpy as np
from numpy import loadtxt
import copy
from queue import PriorityQueue
from timeit import default_timer as timer
dis = [[0,-1],[0,1],[-1,0],[1,0]]

class Node:
    """定义搜索的节点"""
    def __init__(self,s,parent = 0,cost = 0,level = 0):
        self.store = s           #存储二维数组表示状态
        self.parent = parent     #存储节点的父亲节点，方便回溯
        self.cost = cost         #存储h(x)的计算结果
        self.level = level       #存储g(x)的计算结果

    def get_str(self):
        return str(self.store)  #获取二位数组的字符串形式

    def __lt__(self,other):
        """定义内置比较函数,优先队列可以根据该函数进行优先级的判
定"""
        if(self.cost + self.level == other.cost +
other.level):return self.cost + self.level > other.cost +
other.level
        return self.cost + self.level < other.cost +
other.level

def calculate_h(store):
    """计算启发式函数:曼哈顿距离"""
    sum = 0
    for i in range(len(store)):
        for j in range(len(store[i])):
            s = store[i][j]
            if(s == 0):continue
            index1 = (s-1) //4
            index2 = (s-1) % 4
            sum += abs(i - index1) + abs(j - index2)
    return sum
```

```
def valid(x,y,m,n):
    """判断对二维数组的访问是否越界"""
    return x<m and x>=0 and y<n and y>=0

def swap(s,new_x,new_y,x,y):
    """自定义交换函数"""
    temp = s[new_x][new_y]
    s[new_x][new_y] = s[x][y]
    s[x][y] = temp

def A_Star(root,initial,close):
    """A*搜索算法"""
    store = PriorityQueue() #创建优先队列
    store.put(root)      #将根节点push进去
    close.add(hash(root.get_str()))    #将根节点的字符串表达式加入close集合中，进行环检测
    while(not store.empty()):
        node = store.get()  #获取f(x) = h(x) + g(x)最小的节点并弹出
        if(str(node.store) == str(initial)):return node #如果找到了目标节点的状态即返回节点
        index = np.where(node.store == 0)    #定位0在二维数组中的位置
        x = index[0][0]
        y = index[1][0]
        for i in range(4):
            #生成子节点，进行广度优先搜索（不全是，因为有优先队列的参与，因此可能会回到上一层进行搜索）
            new_x = x + dis[i][0]
            new_y = y + dis[i][1]    #获取新的即将访问的节点的行和列的坐标
            if(not valid(new_x,new_y,4,4)):continue      #如果越界则跳过
            temp = copy.deepcopy(node.store)     #进行深拷贝，防止篡改原先node中的状态
            swap(temp,new_x,new_y,x,y)       #进行交换，相当于移动滑块的操作
```

```
            hx = calculate_h(temp)
            s = hash(str(temp))
            if (s in close):continue     #进行环检测，如果节点在
close中，则跳过
            close.add(s)                 #如果没有则加入

            cur = Node(temp,node,hx,node.level+1)     #创建新的
树节点
            store.put(cur)           #将该新生成的子节点加入优先
队列中，进行下一轮的循环

initial = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],
[13,14,15,0]])
start = loadtxt('test4.txt',dtype = int,delimiter = ',')     #
以文件的形式读入初始数组
close = set()
s = Node(start,0,calculate_h(start),0)   #创建初始根节点
tic = timer()   #调用测试时间函数进行搜索时间的测量
root = A_Star(s,initial,close)
toc = timer()
res = []
while(root):
    res.append(str(root.store))
    root = root.parent  #从叶子节点往根节点回溯
res.reverse()       #反转答案
s1 = "We tried "+str(len(res)-1)+" times\n"
line = "---------------\n"
s2 = "The search time is "+str(toc-tic)+" seconds!\n"
with open('A_result4.txt','w') as f:
    for i in range(len(res)):
        f.writelines(str(res[i]))
        if(i>=1):
            f.writelines("   step "+str(i))
        f.writelines('\n'+line)
    f.writelines(s1)
    f.writelines(s2)
```

A_Star算法(优化后)：

```
import numpy as np
from numpy import loadtxt
import copy
from queue import PriorityQueue
from timeit import default_timer as timer

node_num = 0

class Node:
    """定义搜索的节点"""
    def __init__(self,s,parent = 0,cost = 0,level = 0):
        self.store = s           #存储二维数组表示状态
        self.parent = parent     #存储节点的父亲节点，方便回溯
        self.cost = cost         #存储h(x)的计算结果
        self.level = level       #存储g(x)的计算结果
        self.fn = self.level + self.cost

    def __lt__(self,other):
        """定义内置比较函数,优先队列可以根据该函数进行优先级的判
定"""
        if(self.fn == other.fn):
            return self.cost < other.cost
        return self.fn < other.fn

def calculate_h(store):
    """计算启发式函数:曼哈顿距离"""
    sum = 0
    for i in range(len(store)):
        if(store[i] == 0):continue
        index1 = (store[i]-1) //4
        index2 = (store[i]-1) % 4
        j = i//4
        k = i%4
        sum += abs(j - index1) + abs(k - index2)
    return sum

def A_Star(root,initial,close):
    """A*搜索算法"""
```

```
    store = PriorityQueue() #创建优先队列
    store.put(root)      #将根节点push进去
    while(not store.empty()):
        global node_num
        node_num += 1
        node = store.get()  #获取f(x) = h(x) + g(x)最小的节点并
弹出
        if(node.store == initial):return node #如果找到了目标节
点的状态即返回节点
        if(node.store in close):continue     #进行环检测
        else :close.add(node.store)
        index = node.store.index(0)    #定位0在二维数组中的位置
        y = index // 4
        x = index % 4
        moves = []
        if x > 0 : moves.append(-1)     # 左移
        if x < 3 : moves.append(+1)     # 右移
        if y > 0 : moves.append(-4)     # 上移
        if y < 3 : moves.append(+4)     # 下移

        for m in moves:
            #生成子节点，进行广度优先搜索（不全是，因为有优先队列的
参与，因此可能会回到上一层进行搜索）
            s = list(node.store)
            s[index] = s[index+m]
            s[index+m] = 0
            temp = tuple(s)
            if(temp in close):continue
            hx = calculate_h(temp)
            cur = Node(temp,node,hx,node.level+1)     #创建新的
树节点
            if(temp == initial):return cur
            store.put(cur)           #将该新生成的子节点加入优先
队列中，进行下一轮的循环

def print_answer(root):
    res = []
    while(root):
```

```
            res.append(root.store)
            root = root.parent  #从叶子节点往根节点回溯
        res.reverse()       #反转答案
        s1 = "We tried "+str(len(res)-1)+" times\n"
        line = "---------------\n"
        s2 = "The search time is "+str(toc-tic)+" seconds!\n"
        with open('A_result.txt','w') as f:
            for i in range(len(res)):
                r = np.array(list(res[i])).reshape(4,4)
                f.writelines(str(r))
                if(i>=1):
                    f.writelines("   step "+str(i))
                f.writelines('\n'+line)
            f.writelines(s1)
            f.writelines(s2)
        print(node_num)

if __name__ == '__main__':
    end_state = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 0)
    start = loadtxt('test5.txt',dtype = int,delimiter = ',')
#以文件的形式读入初始数组
    close = set()
    start_state = tuple(start.flatten())
    s = Node(start_state,0,calculate_h(start_state),0)   #创建
初始根节点
    tic = timer()   #调用测试时间函数进行搜索时间的测量
    root = A_Star(s,end_state,close)
    toc = timer()
    print_answer(root)
```

IDA_Star算法(未优化)

```
import numpy as np
from numpy import loadtxt
import random
import copy
from queue import PriorityQueue
from timeit import default_timer as timer
dis = [[-1,0],[0,-1],[0,1],[1,0]]   #方向数组
choice = [1,2,3,0]
def calculate_h(store):
    """计算启发式函数:曼哈顿距离"""
    sum = 0
    for i in range(len(store)):
        for j in range(len(store[i])):
            s = store[i][j]
            if(s == 0):continue
            index1 = (s-1)//4
            index2 = (s-1) % 4
            sum += abs(i - index1) + abs(j - index2)
    return sum

def valid(x,y,m,n):
    """判断是否访问越界"""
    return x<m and x>=0 and y<n and y>=0

def swap(s,new_x,new_y,x,y):
    """自定义交换函数"""
    temp = s[new_x][new_y]
    s[new_x][new_y] = s[x][y]
    s[x][y] = temp

def str_hash(s):
    return hash(str(s))

def dfs(close,path,bound,initial,g):
    """迭代加深启发式搜索"""
    node = path[-1]     #获取路径上的最后一个节点
    expense = g + calculate_h(node)       #计算代价即h(x)+g(x)
    if(expense > bound):return expense  #如果代价大于界限，则返
```

```
回该代价
    if(str_hash(node) == str_hash(initial)):  #如果找到了相匹配的目标状态，则返回0，表示成功找到
        return 0
    index = np.where(node == 0) #定位0在二维数组中的位置
    x = index[0][0]
    y = index[1][0]
    res = 100           #定义res，用它来找到子节点中的最小f(x)
    for i in range(4):
        #生成子节点
        new_x = x + dis[i][0]
        new_y = y + dis[i][1]
        if(not valid(new_x, new_y, 4, 4)):continue
        temp = copy.deepcopy(node)
        swap(temp,new_x,new_y,x,y)

        if(str_hash(temp) in close):continue #进行环检测
        else :close.add(str_hash(temp))
        path.append(temp)   #将生成的子节点加入path路径中
        ans = dfs(close,path,bound,initial,g+1)      #递归搜索
        if(ans == 0):return 0   #如果返回值为零则表示已经找到了答案，这时候便可以返回0
        if(ans < res): res = ans      #如果没有找到答案，则更新res值，取较小的一个
        path.pop()          #删去最后一个节点，进行回溯
        close.remove(str_hash(temp))     #同理删去close表中的新生成的节点，以便回溯
    return res

def IDA_Star(initial,close,start):
    '''IDA_Star算法核心循环体'''
    path = []
    bound = calculate_h(start)
    path.append(start)
    close.add(str_hash(start))
    while(True):
        #假定存在答案，进行循环搜索
        ans = dfs(close, path, bound, initial, 0)
```

```
        if(ans == 0):return [bound,path]    #返回边界和路径
        bound = ans     #根据搜索到的子节点的f(x)的最小值更新
bound值，进行下一轮的搜索

initial = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],
[13,14,15,0]])
start = loadtxt('test1.txt',dtype = int, delimiter = ',')   #
读入文件中的测试样例，转换为np数组
close = set()   #创建close表，使用set数据结构，方便进行环检测
tic = timer()
root = IDA_Star(initial, close, start)
toc = timer()
#时间函数的测量，计算运行时间
res = root[1]
s1 = "We tried "+str(len(res)-1)+" times\n"
line = "---------------\n"
s2 = "The search time is "+str(toc-tic)+" seconds!\n"
with open('IDA_result4.txt','w') as f:
    for i in range(len(res)):
        f.writelines(str(res[i]))
        if(i>=1):
            f.writelines("   step "+str(i))
        f.writelines('\n'+line)
    f.writelines(s1)
    f.writelines(s2)
```

IDA_Star算法(优化后)

```
import numpy as np
from numpy import loadtxt
import random
import copy
from queue import PriorityQueue
from timeit import default_timer as timer

path =[]
close = set()
end_state = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 0)
#上面的作为全局变量

def calculate_h(store):
    """计算启发式函数:曼哈顿距离"""
    sum = 0
    for i in range(len(store)):
        if(store[i] == 0):continue
        index1 = (store[i]-1) //4
        index2 = (store[i]-1) % 4
        j = i//4
        k = i%4
        sum += abs(j - index1) + abs(k - index2)
    return sum

def swap(s,x,y):
    """自定义交换函数"""
    temp = s[x]
    s[x] = s[y]
    s[y] = temp

def dfs(bound,g):
    """迭代加深启发式搜索"""
    node = path[-1]      #获取路径上的最后一个节点
    expense = g + calculate_h(node)       #计算代价即h(x)+g(x)
    if(expense > bound):return expense  #如果代价大于界限，则返
回该代价
    if(node == end_state):  #如果找到了相匹配的目标状态，则返回
```

```
0，表示成功找到
        return 0
    index = node.index(0)    #获取0在数组中的位置
    x = index % 4
    y = index // 4
    moves = []  #创建moves数组，存储偏移量
    if x > 0 and index - 1 >= 0 : moves.append(-1)     # 左移
    if x < 3 and index + 1 < 16 : moves.append(+1)     # 右移
    if y > 0 and index - 4 >= 0 : moves.append(-4)     # 上移
    if y < 3 and index + 4 < 16 : moves.append(+4)     # 下移
    res = 1000           #定义res，用它来找到子节点中的最小f(x)
    for m in moves:
        #生成子节点
        temp = list(node)    #将tuple转换为list，方便修改生成子状
态
        swap(temp,index,index+m)     #进行交换
        s = tuple(temp)      #将list转换为tuple，实现格式对齐
        if(s in close):continue #进行环检测
        else :close.add(s)
        path.append(s)    #将生成的子节点加入path路径中
        ans = dfs(bound,g+1)      #递归搜索
        if(ans == 0):return 0    #如果返回值为零则表示已经找到了答
案，这时候便可以返回0
        if(ans < res): res = ans      #如果没有找到答案，则更新
res值，取较小的一个
        path.pop()          #删去最后一个节点，进行回溯
        close.remove(s)     #同理删去close表中的新生成的节点，以便
回溯
    return res

def IDA_Star(start):
    '''IDA_Star算法核心运行循环体'''
    bound = calculate_h(start)  #以初始节点的f(x)值作为初始bound
    path.append(start)
    close.add(start)
    while(True):
        #假定存在答案，进行循环搜索
        ans = dfs(bound, 0)
```

```
        if(ans == 0):return [bound,path]     #返回边界和路径
        bound = ans      #根据搜索到的子节点的f(x)的最小值更新
bound值，进行下一轮的搜索

if __name__ == '__main__':
    start = loadtxt('test1.txt',dtype = int, delimiter = ',')
    start_state = tuple(start.flatten())
    close = set()
    tic = timer()
    combine = IDA_Star(start_state)
    toc = timer()
    res = combine[1]
    s1 = "We tried "+str(len(res)-1)+" times\n"
    line = "---------------\n"
    s2 = "The search time is "+str(toc-tic)+" seconds!\n"

    with open('IDA_result1.txt','w') as f:
        '''将输入打印到结果文件中'''
        for i in range(len(res)):
            r = np.array(list(res[i])).reshape(4,4)
            f.writelines(str(r))
            if(i>=1):
                f.writelines("   step "+str(i))
            f.writelines('\n'+line)
        f.writelines(s1)
        f.writelines(s2)
```

4、算法优化和创新点

算法优化：

1、环检测优化：

对于环检测，一开始使用的是list，由于list查找时间复杂度为O(n)，然而节点的扩展会非常多，因此会导致非常多的时间浪费在环检测上。

因此，我们可以考虑采用更好的数据结构，以实现快速查找和插入。于是我们可以使用set()集合，它在C++中是基于红黑树的底层实现，查找时间复杂度为O(log(n))；在python中一般基于哈希表的思想实现，查找速度接近于O(1),能够很

快地处理大规模数据。

2、路径检测优化：

同上环检测，环检测是相对于A_Star算法而言的，而路经检测是相对于IDA_Star算法而言的。

3、元组优化：

对于上面的环检测优化，还有一点需要注意的是，将一个唯一可标识地状态存入集合，必须要求该状态是要可哈希的。通过测试，发现**list是不可哈希的**，因此如果采用将其转换为str的形式进而存入集合中的方法会耗费大量的时间在转换list->str的过程。

因此，我们要考虑更快的方法，由于**元组**不可更改而且**可直接哈希**，因此可以考虑直接使用元组进行数据的存储。我这里采用的是一维元组，方便定位0的位置。

在生成子状态的时候，可以采用list过渡的形式，先将tuple转换为list，修改后转换为tuple，环检测的时候直接判断或者直接进行加入，可以优化10倍左右的效率

创新点：

1、 如上面所提到的，采用了set替换list，tuple替换list存储的方法，节省了大量的时间和空间。

2、 对于A_Star算法，在node节点中记录parent，方便找到节点后回溯；对于IDA_Star算法，利用path记录路径，找到目标节点后直接顺序打印即可。

3、 A_Star算法的双向搜索优化创新(不确保路径的最优性，但是时间非常快，10秒以内)：

对于双向搜索，相当于在原来的循环基础上在增添一反向搜索的代码块。而搜索策略采用局部搜索，即动态更新终点。

具体步骤如下：

1、将起点和终点分别加入OPEN1和OPEN2正反两个优先队列中，并分别加入CLOSE1和CLOSE2中

2、从OPEN2中弹出f(x)值最小的节点，设为target1，从OPEN1中弹出F(x)值最小的节点，设为node1。将target1作为新的终点

3、判断target1节点是否在CLOSE1中，如果在则直接返回正向和反向节点

4、扩展node1，并进行环检测

5、生成node1的子节点，将节点中的元组信息和整个节点建立键值对关系，方便return的时候能够找到返回的节点(因为判断是否应该返回是通过比较元组进行判断的)

6、从OPEN1中弹出f(x)值最小的节点，设为target2，从OPEN2中弹出f(x)值最小的节点node2，将target2作为新的终点.

7、重复类似3~5的操作进行反向搜索

三、实验结果及分析

实验结果展示示例

实验结果详情见文件

test1 ~ test8表示测试文件
A_result1 ~ A_result8表示A_Star算法运行结果
IDA_Star1 ~ IDA_Star8表示IDA_Star算法运行结果
A_better1 ~ A_better8表示双向优化搜索的运行结果

评测指标展示及分析

优化前：
在没有优化前，由于存储状态采用的是列表，在进行set集合环检测的时候，由于list不可哈希，因此要转换成字符串再加入。因此在扩展每个节点的时候都要花费一定的时间，导致性能不好

测试样例	步数(A*)	运行时间(A*)	步数(IDA*)	运行时间(IDA*)
1	40	1.8489897 seconds	40	11.3278796 seconds
2	40	1.0734810 seconds	40	3.18315870 seconds
3	40	1.3206081 seconds	40	1.24297460 seconds
4	40	8.9434440 seconds	40	23.9139577 seconds
5	X	内存超限	X	时间超限
6	X	内存超限	49	1203.3372927 seconds
7	X	内存超限	X	时间超限
8	48	1021.2520267 seconds	X	时间超限

优化后：

采用了元组优化，由于元组直接可哈希，可以直接加入set集合中，因此可以省去转换为字符串的时间，缩短运行时间

测试样例	步数(A*)	运行时间(A*)	步数(IDA*)	运行时间(IDA*)
1	40	0.1414206 seconds	40	0.2903845 seconds
2	40	0.0641422 seconds	40	0.1487615 seconds
3	40	0.0352199 seconds	40	0.0806829 seconds
4	40	0.4425620 seconds	40	2.3567988 seconds
5	56	1002.4435007 seconds	56	37852.2294486 seconds
6	49	95.5534819 seconds	49	53.873191 seconds
7	X	内存超限	62	28601.4116929 seconds
8	48	91.2407716 seconds	48	384.8310241 seconds

对于A* 算法，步数越多，花费的时间一般越大，这还与其复杂性有一定关联。而对于IDA* 算法，花费的时间于解在搜索树上的位置有关，更与扩展搜索方向的优先级有关，因此出现了花费步数多的test运行时间比花费步数少的test运行时间还要短的情况

IDA_Star和A_Star算法的比较分析：

A*算法：

优点： 采用广度优先搜索策略，但不盲目搜索，加入启发式函数引导搜索方向。由于每次的深度是一步一步加深扩展，因此找到的路径是最优路径。

缺点： 利用广度优先搜索的策略在空间上都会存在内存上的问题，当搜索层

数增大的时候，存储的节点进一步变多，纵使有环检测，但是基本上以指数级别递增，导致最后**内存超限**。

IDA*算法：

优点： 采用深度优先策略，每次根据超过边界节点f(x)值的最小值确定bound值，一步一步迭代加深。类似于迭代加深算法，只不过深度被替换成了 **f(x) = h(x) + g(x)** 。 由于深度不断迭代加深，所以搜索到的结果一定是最优解。且**空间复杂度相对较为良好**，为O(bm)(b为每个状态扩展的子节点的个数，m为树的深度)

缺点： 迭代加深bound虽然可以在解存在的情况下找到最优解，但是每次迭代都会**重复搜索**，走了很多无用的道路，导致有很多时间花在没用的搜索上。而且如果搜索的方向固定(即固定的上、下、左、右)也会导致解在搜索树的不同位置产生不同的搜索效率(解如果在搜索树的左边则快，如果在右边则慢)。这种差别体现在test5和test7上，test7总共要走62步，但是test5走了56步，但是test7运行比test5还要快，说明test7的解在搜索树上更接近于搜索的优先方向。

综合分析：

根据上面的分析，以及实验结果，我们可以大致得出实验结论：

即对于状态空间较小的问题求解，A* 算法能够在平均更短的时间内找到问题的解；

而对于状态空间大的问题求解，IDA*算法可以更好的应对，在更短的时间内找到问题的解。但是要取决于搜索方向的顺序和优先级，如果固定方向，而解的位置与方向相反，可能会花费更久的时间。(参考test5)

四、对运行时间进一步优化的思考

鉴于在测试test7的时候，A*算法内存超限，可以进一步考虑优化空间。那么空间的臃肿是由优先队列的指数级增长导致的，于是，可以考虑使用双向搜索，并且动态更新起点和终点(相当于局部搜索)进行时间和空间方面的优化。

但是由于双向搜索的策略并不保证最优性，所以搜索出来的结果只能具备可解性，即存在这个解。但由于双向搜索 + 局部搜索，所以时间会非常快，对于每个测试样例都不超过十秒，但是所花费的步骤相比最优解多了不少。具体步骤详见**创**

新点 3

运行时间与步数：

测试样例	运行时间	运行步数
test1	0.1931815 seconds	44
test2	0.1173185 seconds	56
test3	0.0078335 seconds	48
test4	0.2526883 seconds	56
test5	1.8949151 seconds	76
test6	1.4760733 seconds	63
test7	8.6370868 seconds	88
test8	1.6642869 seconds	62