A* 与IDA* 解决15-Puzzle问题

- 尝试使用python实现A* 和IDA* ， A* 算法没有成功跑出样例1和3，所以最后使用自己设计的样例进行测试，IDA* 算法的python版本能够测出样例2和样例4，时间在10分钟左右，需要很长时间跑出样例1和3，为了能够在短时间跑出全部样例，最后使用c++语言重新实现IDA* ，并作适当的改进，最终快速能够跑出四个样例，样例2和4只需要3s左右，样例1和3需要200s左右，经过大O优化，样例2和4时间需要1.5s左右，样例1和3时间需要130s左右。

（1）算法原理

1. A* （python）

   A* 算法基于BFS算法，利用类Node存储一个结点的状态、估价函数、移动路径，一开始初始化开启列表、关闭列表，其中开启列表使用优先队列实现，将根结点放到队列中去，每次从优先队列中选出估价函数最小的结点，判断是否达到目标状态，达到就返回，否则进行扩展，除了上次移动方向，其他三个方向进行扩展，判断是否越界和已经访问过，如果没有放入开启列表，进行下一步循环，直到找到最优解。

2. IDA*（python）

   每次进行DFS策略搜索，设置一个阈值进行限制，第一次迭代以初始状态到目标状态的估价函数为初始阈值，进入DFS函数，对当前结点进行判断是否达到目标状态，如果达到则直接返回；对当前三个扩展方向（避免走回头路）判断是否越界或已经访问，计算三个方向的估计代价，选择最低的估价函数值的方向进行扩展，如果此时的估价函数超过阈值，则结束搜索，否则更新关闭列表，递归调用下一层的DFS函数，如果找不到解则进行回溯，回溯时需要将关闭列表恢复，一直按这个策略找到最优解。

3. IDA* （c++）

   C++版本的IDA*基本思想与python版本差不多，在具体实现作了几个变动，比如减少传参开销，不再使用类存储结点，所有递归公用一个数组，只是不同调用次序对应的数组状态不一致，大大节省空间，同时基于检测开销与重复访问开销的比较，去除重复访问的检测，跟python版本的相比，将程序的效率提高了300倍以上。

（2） 伪代码

1. A*

```
Algorithm A*
    #遍历队列中所有结点，找到最优解
    while open_queue is not empty do:
        #取出估价最小的结点
        get node whose fn is lowest from open_queue
        #判断是否到达目标状态
        if node get to the destination do:
```

```
            over
        #遍历各个方向
        for moevment in move do:
            update x,y #更新空格坐标
            #检测越界
            if x,y crossing the border do:
                continue
            #检测是否走回头路
            if node go back last road do:
                continue
            create newNode #创建新结点
            #如果有访问过，下一个方向探索
            if newNode has been visited do:
                continue
            else:
                update close_queue #更新访问集
                push newNode to open_queue #入队
```

2. IDA*

```
Algorithm IDA*

function IDA
    #初始迭代以初始点到终点的估计代价为阈值
    bound <- h(x) from start to end
    while True do:
        init open_queue,close_queue #初始化
        result <- A_DFS(open_queue,close_queue) #调用深度优先的A*
        #找到终点退出
        if result == reach do:
            return success
        #大于1000层就停止搜索
        if result >= 1000:
            return fail
        bound <- result #将阈值更新为大于上次阈值的最小估价函数

function A_DFS
    #取出最后进队的结点
    get node from back of open_queue
    #判断是否到达终点
    if node reach the destinaiton do:
        return reach

    min_limit <- 10000 #找出最小的估价函数
    #选出最小fn的方向扩展
    for succ in choose_successor do:
        #检测是否已经访问
        if succ in close_queue do:
            continue
        #入队
        push succ to close_queue
```

```
        push succ to open_queue
        result = A_DFS() #递归调用下层的A_DFS
        #找到终点，返回成功
        if result == reach do:
            return reach
        #更新最小估价函数
        if result < min_limit do:
            min_limit <- result
        #回溯恢复
        close_queue.pop(-1)
        open_queue.pop(-1)
    return min_limit #返回最小估价函数值给上一级

function choose_successor
    success_list <- []
    #扩展各个方向
    for movement in move:
        update x,y #更新空格坐标
        create newNode #创建新结点
        #检查是否越界
        if newNode crossing the border do:
            continue
        #检查是否走回头路
        if newNode go back last road do:
            continue
        push newNode to success_list #入队
     sort success_list by fn #将结点按fn从小到大排列，保证出队结点是最小fn
     return success_list #返回列表
```

(3) 关键代码

1. 启发式函数

- 曼哈顿距离

  计算矩阵每个元素的横坐标、纵坐标与目标状态的横坐标、纵坐标的差的绝对值并累加，作为启发式函数，由于该函数接近实际花费的代价，因此效果比较理想。

```
#计算曼哈顿距离
def heuristic_manhadun(state,target):
    current_array = state  #矩阵状态
    length_x = len(current_array)
    length_y = len(current_array[0]) #计算矩阵的size
    hn = 0
    for i in range(length_x):
        for j in range(length_y):
            if (current_array[i][j] != 0): #排除0的影响，保证可采纳性
                end_x,end_y = target[current_array[i][j]]
                x_dis = abs(end_x - i)
                y_dis = abs(end_y - j)

                hn = hn + x_dis + y_dis  #加上横坐标和纵坐标差的绝对值
```

```
                # print(x_dis+y_dis,current_array[i][j],target[current_array[i][j]],i,j)
    return hn
```

- 切比雪夫距离

  比较矩阵每个元素的横坐标、纵坐标与目标元素状态的横坐标、纵坐标的差的绝对值，选取绝对值比较大的作为启发式函数的一部分，衡量该元素到达目标状态的代价，累加到启发式函数中，符合可采纳性、一致性的条件。

```
#计算切比雪夫距离
def heuristic_chebyshev_distance(state,target):
    current_array = state
    length_x = len(current_array) #计算矩阵大小的size
    length_y = len(current_array[0])
    hn = 0
    for i in range(length_x):  #遍历计算每个元素
        for j in range(length_y):
            end_x, end_y = target[current_array[i][j]]  #寻找元素目标位置
            x_dis = abs(end_x - i) #横坐标差绝对值
            y_dis = abs(end_y - j) #纵坐标差绝对值
            hn = hn + max(x_dis, y_dis) #取横、纵坐标差绝对值的最大值相加

    return hn  #评估值
```

- 正确位置的元素个数

  不再从计算横纵坐标下手，而是直接统计当前矩阵中有多少个元素已经到达目标状态，进而作为启发式函数评估当前状态到目标状态的代价。

```
#计算元素正确位置个数
def heuristic_element_num(state,target):
    current_array = state  #当前十五数码状态
    length_x = len(current_array) #矩阵的size
    length_y = len(current_array[0])
    hn = 0
    for i in range(length_x): #遍历每个元素，统计
        for j in range(length_y):
            end_x, end_y = target[current_array[i][j]] #得到元素的目标坐标
            if end_x ! = i or end_y ! = j: #判断元素是否到达目标位置，如果不是，加1
                hn += 1

    return hn #返回元素正确到达位置个数
```

- 当前元素与目标相同位置元素的值的差的绝对值。

  累加每个元素与当前位置的目标元素的差的绝对值作为启发式函数。

由于该方法没有满足可采纳性和一致性，虽然运行速度很快，但是没有办法找到最优解，因此该启发式函数不可用。

```
int right_value(int array1[ARRAY_SIZE][ARRAY_SIZE]) {
    int hn = 0,x,y;
    int value;
    for (int i=0;i < ARRAY_SIZE;i++) {
        for (int j=0;j < ARRAY_SIZE;j++) {
            if (array1[i][j] != 0) {
                value = i*4 + j + 1; //计算目标位置的元素值
                hn += abs(array1[i][j] - value); //求两个元素的差，累加其绝对值
            }
        }
    }
    return hn;
}
```

2. A*

- 存储结构：使用类结构存储一个结点，为了方便显示，统一将结点的各个信息存储进来，记录上次移动的元素，方便输出路径，记录父结点，方便找到上一个结点，fn、gn、hn都是移动的代价，state是当前状态，是一个矩阵。

```
class Node():
    def __init__(self,state,parent,gn,hn,element = 0):
        self.state = state  #矩阵状态
        self.parent = parent #父结点
        self.gn = gn  #目前花费代价
        self.hn = hn #预估代价
        self.fn = gn + hn #估价值
        self.element = element #上次移动的元素
```

- 优先队列：按照A* 算法，每次出队都选出估价函数fn最小的结点进行扩展，因此为了提高排序、出队效率，使用堆heap实现一个优先队列类，队列存储的是一个Node类结点，按照估价函数fn进行排序，使得每次出队pop时都是选择fn最小的结点。

```
class PriorityQueue:
    #初始化
    def __init__(self):
        self.heap = []
        self.count = 0
    #入队，将结点插入队列，同时输入fn作为排序依据
    def push(self, item, priority):
        entry = (priority, self.count, item)

        heapq.heappush(self.heap, entry)
```

```
        self.count += 1
    #取出fn最小的结点
    def pop(self):
        (_, _, item) = heapq.heappop(self.heap)
        return item
    #判断是否为空
    def isEmpty(self):
        return len(self.heap) == 0
    #更新，将结点按fn进行堆排序
    def update(self, item, priority):

        for index, (p, c, i) in enumerate(self.heap):
            if i == item:
                if p <= priority:
                    break
                del self.heap[index]
                self.heap.append((priority, c, item))
                heapq.heapify(self.heap)
                break
        else:
            self.push(item, priority)
```

- 首先进行初始化，使用优先队列open_queue存储探索的结点，列表close_queue存储以扩展的结点。

```
    close_queue = [] #已访问的结点集合
    close_queue.append(start_node.state)

    open_queue = PriorityQueue() #优先队列初始化
    open_queue.push(start_node,0)  #插入根结点
```

- 每次选最优结点，判断是否达到目标状态。

```
        node = open_queue.pop() #出队，选fn最小的结点
        #如果是目标状态，返回当前结点
        if node.state == dst_node.state:
            return node
```

- 寻找空格的二维下标

```
        for i in range(len(node.state)):
            for j in range(len(node.state[0])):
                if node.state[i][j] == 0: #找到空格记录位置
                    move_x = i
                    move_y = j

                    break
```

- 对每个方向进行扩展，判断是否越界，以及走回头路（上次扩展来的方向）。

```
            x = move_x + movement[0] #更新x
            y = move_y + movement[1] #更新y
            #越界探测
            if x >= 4 or x < 0 or y >= 4 or y < 0:
                continue
            #保证不走回头路
            if node.state[x][y] == node.element:
                continue
```

- 创建新的结点，更新结点各个属性。

```
            new_state = copy.deepcopy(node.state) #深复制，避免影响上一个结点
            new_state[move_x][move_y] = new_state[x][y] #移动位置
            new_state[x][y] = 0 #更换0的位置
            element = new_state[move_x][move_y] #存储移动的元素数值
            gn = node.gn+1 #成本加1
            hn = heuristic_manhadun(new_state, target) #求估计代价
            new_node = Node(new_state,node,gn,hn,element) #创建新结点
```

- 判断新状态是否已经访问，如果没有访问，则入队，并更新访问集。

```
           #如果新结点状态已经探测，则放弃该方向扩展
            if new_node.state in close_queue:
                continue
            #否则进行扩展，置为已经访问
            else:
                close_queue.append(node.state)
                open_queue.push(new_node,new_node.fn)
```

3. IDA* （python）

- 调用IDA函数，计算初始结点到终点的估计代价，作为第一次迭代的阈值，每次初始化开启列表、关闭列表，如果在当前深度找不到解，将返回的最小估价函数作为下一次迭代的阈值，当超过一定深度后，不再搜索，直接返回。

```
def IDA(start_node,dst_node,target,move):
    bound = heuristic_manhadun(start_node.state,target) #以初始点到终点的估价函数值为阈值
    open_queue = [] #开启列表

    open_queue.append(start_node) #初始点入队
```

```
    while True:
        print(bound)
        close_queue = [] #关闭列表
        close_queue.append(start_node.state) #初始点置为已经访问
        #将阈值作为探索的深度，调用以DFS为基础的A*算法函数
        result = A_DFS(open_queue,0,bound,close_queue,target,move,dst_node)
        #结果等于-2，则找到正确结果
        if result == -2:
            return open_queue
        #超过一定深度，不再探索
        if result > 100:
            return None
        #将返回的估计函数最小值作为下一次迭代的阈值
        bound = result
    return None
```

- 进入A_DFS函数，按照深度优先规则，取出结点，判断是否到达目标。

```
    node = open_queue[-1] #取最后入队的一个结点进行扩展
    if node.state == dst_node.state: #如果到达目标状态，返回
        return -2
```

- 对于当前的结点，对空3个运动方向进行扩展，更新坐标，判断是否越界，避免走回头路，创建新的结点，将新的结点加入自己创建的队列中，然后按照fn从小到大排序，方便后面当前结点对三个方向选择最小fn的方向优先进行扩展，符合DFS的原理。

```
    success_list = []
    for movement in move:
        x = move_x + movement[0] #更新x
        y = move_y + movement[1] #更新y
        #判断是否越界
        if x >= 4 or x < 0 or y >= 4 or y < 0:
            continue
        #判断是否走回头路
        if node.state[x][y] == node.element:
            continue
        new_state = copy.deepcopy(node.state) #深复制
        new_state[move_x][move_y] = new_state[x][y] #移动元素
        new_state[x][y] = 0 #移动空格
        element = new_state[move_x][move_y] #记录移动的元素
        gn = node.gn+1 #改变实际代价
        hn = heuristic_manhadun(new_state, target) #求估计代价
        # hn = heuristic_chebyshev_distance(new_state,target)
        new_node = Node(new_state,node,gn,hn,element) #创建新结点
        success_list.append(new_node) #加入队列
    #对扩展的3个方向进行选择，按照fn从小到大
    return sorted(success_list,key = lambda x : x.fn)
```

- 对于当前的结点，设置min_limit，这是为了找出大于阈值的最小估价函数值，为了下一次迭代备用，针对上面的列表success_list，依次从中选出最小的结点进行扩展，首先判断是否在开启列表或者关闭列表，如果不在进行下一步，如果扩展的新结点的fn大于阈值，直接返回fn，递归调用A_DFS，根据返回值比较，如果找到就返回-2，否则跟min_limit进行比较，更新最小的值为min_limit，最后更新开启列表、关闭列表，返回min_limit给上一级。

```
    min_limit = 10000 #找出最小的估价函数值
    #按照fn从小到达选择要扩展的方向
    for succ in choose_successor(node,target,move):
        new_node = succ
        isvisit = 0
        #判断结点是否在开启队列中
        for node1 in open_queue:
            if new_node.state == node1.state:
                isvisit = 1
                break
        if isvisit == 1:
            continue
        else:
            #判断结点是否在关闭队列中
            if new_node.state in close_queue:
                continue
            #如果新结点的fn大于阈值，停止扩展，返回
            if new_node.fn > bound:
                return new_node.fn
            #添加结点到关闭队列
            close_queue.append(new_node.state)
            #添加结点到开启队列
            open_queue.append(new_node)
            #递归调用A_DFS函数
            result = A_DFS(open_queue,g+1,bound,close_queue,target,move,dst_node)
            #如果找到结果，直接返回-2
            if result == -2:
                return result
            #否则比较当前的估价函数值，得出最小的更新到min_limit
            elif result < min_limit:
                min_limit = result
            open_queue.pop(-1) #出队，恢复
            close_queue.pop(-1) #出队，恢复

    return min_limit #返回最小的估价函数值
```

4. IDA* （c++）

   由于IDA*基本思想上面已描述，不再重复，c++版本在几个方面做了进一步的优化，下面有说，因此此处只粘贴部分代码。

- IDA

```
void IDA(int x_pos,int y_pos) {
    level = manhadun(array1);
    while (level <= 100 && is_target == 0) {
        road.clear();
        DFS(0,-5,x_pos,y_pos);
        if (is_target) {
            break;
        }
        level += 2;
    }
}
```

- DFS

```
void DFS( int gn,int back,int x_pos,int y_pos) {
    int hn = manhadun(array1);
    if (is_target) {
        return ;
    }
    if (hn == 0) {
        is_target = 1;
        min_step = gn;
        return ;
    }
    for (int i=0;i < 4;++i) {
        int x = x_pos + movement[i][0];
        int y = y_pos + movement[i][1];
        if (x < 0 || x >= 4 || y < 0 || y >= 4) {
            continue;
        }
        if (back == -5 || i != (back+2) % 4) {
            road.push_back(array1[x][y]);
            my_swap(&array1[x][y],&array1[x_pos][y_pos]);
            hn = manhadun(array1);
            if (gn+hn <= level) {
                DFS(gn+1,i,x,y);
                if (is_target == 1) {
                    return;
                }
            }
            my_swap(&array1[x][y],&array1[x_pos][y_pos]);
            road.pop_back();
        }
    }
}
```

(4) 优化

1. IDA* c++版本是基于python版本上修改，并且在算法上进行了优化，使得运行时间有了300倍的提高。

- 减少函数参数传递，为了能够提高递归调用效率，将重复使用的变量数组统一声明为全局变量。
- 不再使用类结构存储结点，如果每次都复制一次矩阵再存入类中，则随着深度增加，存储空间占用越多，随时可能内存不足，因此，由此始终都是使用一个数组，每次更新只是将数组的对应元素调转，回溯的时候再恢复，这样可以大大节省空间。
- 省略检测是否已经访问过当前状态的环节，保留检测上次探测方向，保证不走回头路，经过比对，只要保证空格移动方向不是上次探索来时的方向，即不走回头路，那么在更深的层探索到已经探测过的状态的几率比较小，其带来的开销要比检测的开销要小，因为层数越深，探索的结点非常多，即使使用哈希查找，每次探索都要进行检测，其带来的时间开销也不少，因此省略检测在解决数码问题中对效率提升有帮助。

（5）实验结果及分析

实验结果展示与评测指标分析

- IDA*　语言c++，不同启发式函数的运行时间比较

PPT样例	样例1	样例2	样例3	样例4
A 曼哈顿	238.253 s	2.662 s	184.553 s	2.239 s
B 切比雪夫	大于1 h	214.282 s	大于1 h	193.744 s
C 正确元素个数	大于1 h	大于1 h	大于1 h	大于1 h
D 元素正确位置值的差绝对值	78 s	0.99 s	101 s	0.98 s

由上表比对可知，D类型的启发式函数虽然运行时间短，但是由于其不符合可采纳性，估计代价高于实际代价，因此最后得出的结果没有不是最优路径，因此不符合我们的要求；A、B、C三种方式均符合启发式函数可采纳性、一致性的要求，因此最终能够找到最优解。通过比较，曼哈顿方式的运行时间是最优，样例2和4能够在3

s内找到最优解，样例1和样例3也是能够在200s左右找到最优解，经过优化的c++版本比之前用python的方式做效率提高300倍以上，三者对比，效率大小为A > B > C，这是因为h(n)与h *(n)差距大小比较的原因，A曼哈顿的h(n)是与实际移动的步数距离h *(n)最接近的，B切比雪夫只计算横纵坐标差绝对值的最大值，C只计算正确元素个数，因此C方式h(n)比B和A方式远小于h *(n),B方式远小于A方式，而h(n)越接近h *(n)，能够确保找到最优解的同时运行速度更快，因此A的效率比B高，B的效率比C高。

- IDA*　c++与python的运行时间比较：启发式函数选择曼哈顿距离。

PPT样例	样例1	样例2	样例3	样例4
c++	238.253 s	2.662 s	184.553 s	2.239 s
python	大于1 h	663.768 s	大于1 h	479.756 s

由于C++与python的差异，使得两者在运算上能力有所差异，而在c++中，作了进一步的优化，减少参数传递、不存储数组状态，公用一个数组、不进行访问检测等，从几个方面大胆作了改变，最后在保证取得最优解的前提下提高了效率，如样例2比较，c++版本效率提高了330倍以上。

- IDA* 与A*算法运行时间比较

PPT样例	样例1	样例2	样例3	样例4
A*	大于1 h	11 min	大于1 h	9 min
IDA*	238.253 s	2.662 s	184.553 s	2.239 s

IDA* 与A* 对比，IDA* 完胜A* ，这是因为A* 采用BFS方式，每次将扩展的结点放入队列后，下次扩展就从队列中选择估价函数值最小的结点进行扩展，因此相当于是一层层所有结点遍历下来，而这里的层是指实际代价gn，以样例2为例子，其最优解为49，那么实际上A* 需要将49层以上的所有结点都遍历一次，这需要很长的时间，而IDA* 算法则不一样，基于阈值进行迭代，采用DFS思想，每次迭代判断当前结点是否大于阈值，大于则结束探索，以大于阈值的最小估价函数值fn为下一次迭代的阈值，注意的是这里的迭代加深不是gn层数迭代加深，而是阈值fn，因此以样例2为例，刚开始每次迭代其实只要fn>阈值就结束，而实际此时只探索了十几层结点（gn+hn=fn，hn此时比较大），因此速度比较快，但是当阈值到49时进行本次迭代，上一次迭代其实也只有探索二十几层的结点（gn=20+），但是本次迭代中，基于深度优先的方式，IDA* 会优先找到一条最短路径直接到达49层找到最优解直接结束返回，而不需要遍历gn=20+到40+层所有结点，而A* 则需要遍历这个过程，而这个过程又是决定效率的最关键一步，因此IDA* 的效率要比A* 高很多，总的来说，虽然IDA* 在未找到解的阈值范围时，也需要遍历之前的阈值范围，但是在本题，最后一个阈值范围要探索的结点数量比之前所有阈值范围要探索的结点数量要高出不止一个数量级，这就是性能的差异。

- 由于A* 算法没有跑出样例1和3，因此自主设计补充一些样例，证明算法的正确性。

```
样例1
7 5 4 12
1 6 13 8
11 3 0 15
10 9 2 14
move step:  38
[0, 13, 6, 5, 7, 1, 5, 3, 11, 10, 9, 2, 13, 11, 2, 13, 14, 15, 11, 6, 8, 12, 4, 7, 3, 2, 6, 11,
12, 8, 7, 3, 2, 6, 10, 9, 13, 14, 15]
run time :  216.6380707133256  s

样例2
2 3 4 0
1 5 6 7
10 11 12 8
9 13 14 15
move step:  15
[0, 4, 3, 2, 1, 5, 6, 7, 8, 12, 11, 10, 9, 13, 14, 15]
run time :  0.002077449366419477  s
```

由上表显示，经过验证，样例1和样例2的最优步数是38和15，并且显示的路径正确，样例1步数稍微复杂，因此需要216s，样例2比较简单，状态数小，因此时间为0.002s，由此证明算法正确。

- IDA* 与A* 空间复杂度对比
    1. A* 算法基于BFS算法进行扩展，需要一层层遍历所有结点，因此空间复杂度可达到指数级，比较大，层数较深时，容易出现内存不足情况。
    2. IDA* 算法基于DFS算法扩展，每次遍历时需要存储一条路径的空间，因此空间存储为bm，b为结点的子结点数，m为最长路径长度，然而改为c++版本后，不再存储结点状态，也即不存新的二维数组，所有结点共用一个二维数组，因此内存空间对于结点来说，只需要存储一个结点的状态，大大节省存储空间。

PPT样例	样例1	样例2	样例3	样例4
IDA*	17.4 MB	10.8 MB	11.0MB	9.5 MB
A*	None	530 MB	None	620 MB

由表可知，IDA* 采用深度优先，不需要存储所有结点，空间占比只有10-20MB，而A* 存储空间比较大，样例1和样例3需要8GB以上，存储空间比较大。

- IDA* 大O优化 曼哈顿距离

PPT样例	样例1	样例2	样例3	样例4
IDA*	135.536 s	1.51 s	95.308 s	0.98 s

经过进一步优化，运行各个样例时间进一步降低，样例4达到1.0s内，样例1和3效率有了明显提高，效率提高将近一倍。

- 最优性与完备性
    - IDA*
        1. 最优性：由于IDA*是以一层层迭代遍历的，如果上一层没有找到解，而在下一层找到解，那么通过不同的路径到达该层的任何一个解都是相同的最优步数，都是最优路径，因此此时的深度优先具有最优性。
        2. 完备性：由于问题是有解的，而每次扩展的状态数有限，最多为3个，因此一层层往下扩展最终肯定能找到解。
    - A*
        1. 最优性：A*基于gn和hn每次从队列中选出最优结点进行扩展，是升级版的BFS，因此一层层的搜索下来，最先找到的解自然就是路径最短的，因此具有最优性。
        2. 完备性：问题提示有解，不会搜索不到，同时每次扩展状态数有限，最多只有3个子结点，因此根据BFS的原理，肯定能找到解，具有完备性。

总结

- h(n) 与h* (n)
  - 当h(n)为0时，此时退化为dijkstra算法，相当于无信息搜索，在实际代价为正的前提下，dijkstra算法能够找到最优解。
  - 当h(n) < h* (n)时，此时满足启发式函数可采纳性和一致性，因此可以保证找到最优解，并且当h(n)越接近h* (n)时，找到解的速度更快，比如上面实验提到的A方式曼哈顿，比B和C方式运行效率更高，这是因为h(n)越小，扩展的结点数就会更多，从而导致算法变慢。
  - 当h(n) = h* (n)时，此时保证最优解的同时，可以以最快的速度找到最优解，这是因为扩展的每一个结点都在最优路径上，不会扩展其他不在最优路径的结点，尽管比较难找到合适的启发式函数达到这种情况，但这是提高效率的最好方法。
  - 当h(n) > h* (n)时，如上面实验列举的D方式，由于不满足可采纳性和一致性，因此不能找到最优解，但是运行速度更快。
  - 如果h(n)与g(n)比较接近，此时h(n)起决定作用，实际上算法等价于宽度优先搜索。
  - 综上，在寻找解时，需要在算法效率和最优解上作分析，h(n)较小，一定可以找到最优解，但是运行速度慢，h(n)较大，运行速度快，但是不一定找到最优解，因此启发式函数向实际代价h* (n)接近但不超过h* (n)为最理想的设计。