**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

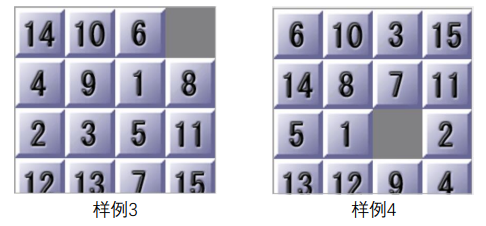
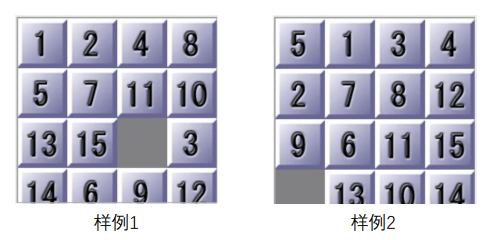
|  |  |  |  |
| --- | --- | --- | --- |
| 教学班级 | **系统结构班** | 专业（方向） | **计算机科学与技术** |
| 学号 | **21307358** | 姓名 | **曾慧蕾** |

# 实验题目

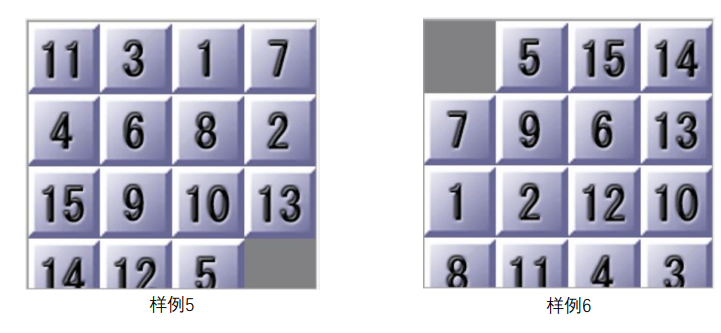
使用A\*与IDA\*算法解决15-puzzle问题，启发式函数可以自己选取，至少尝试两种不同的启发式函数。

需要完成的基本样例与挑战样例如下所示：

基本样例：



挑战样例：



# 实验内容

1. 算法原理

* **A\*算法：**

A\*算法可以看作是BFS算法的升级版，在原有的BFS算法的基础上加入了启发式信息。

其估价函数是f（x）=g（x）+h（x）

**算法描述：**

从起始节点开始，不断查询周围可到达节点的状态并计算它们的f（x），h（x）与g（x）的值，选取估价函数f（x）最小的节点进行下一步扩展，并同时更新已经被访问过的节点的g（x），直到找到目标节点。

**算法步骤：**

**开始**  
1.从起点开始，把其当成待处理点存入一个“开启列表”。  
2搜寻起点周围可能通过的节点，也把它们加入开启列表，为这些节点计算f(),g(x),h(x),并且将节点A存为“父节点’  
3.从开启列表中删除节点A,将其加入“关闭列表”(列表中保存所有不需要 再次检查的节点)

**循环直到找到目标节点或者“开启列表”为空(无路径)**4.从“开启列表”中找到估价函数值最低的节点C，并将它从“开启列表”中删除，添加到"关闭列表”中。  
5.检查C所有相邻节点，将其加入“开启列表”，将C作为它们的父节点。  
6.如果新的相邻节点已经在“"开启列表”,则更新它们的g(x)值

* **IDA\*算法**

IDA\*是迭代加深深度优先搜索算法（IDS）的扩展。因为它不需要去维护表，因此，它的空间复杂度远远小于A\*。在搜索图为稀疏有向图的时候，它的性能会比A\*更好。

其估价函数也是f（x）=g（x）+h（x）

**算法描述：**

在算法迭代的每一步，IDA\*都进行深度优先搜索，在某一步所有可访问结点对应的最小可估价函数值大于某个给定的阈值时，将会剪枝。

**算法步骤：**

对于给定的阈值bound，定义递归过程

1. 从开始节点C，计算所有邻居节点的估价函数，选取估价函数最小的节点作为下一个访问节点。
2. 对于某个节点，如果估价函数大于阈值，则返回当前节点的估值函数值。
3. 对于某个节点，如果是目标节点，则返回状态“到达“。
4. 伪代码

A\*算法：

输入：start，goal

输出：每次移动的方格

frontier=PriorityQueue() #open表

frontier.put(start,0)    #初始化

came\_from=dict()

cost\_so\_far=dict()

came\_from[start]=None    #记录父母

cost\_so\_far[start]=0     #记录f

while not frontier.empty():

    current=frontier.get()

    if current==goal:

        break

    for next in graph.neighbors(current): #访问当前节点的邻居节点

        new\_cost=cost\_so\_far[current]+graph.cost(current,next)

        if next not in cost\_so\_far or new\_cost<cost\_so\_far[next]: #判断是否要加入open表中

            cost\_so\_far[next]=new\_cost

            priority=new\_cost+heuristic(goal,next)

            frontier.put(next,priority)

            came\_from[next]=current

IDA\*算法：

#输入：start，goal

#输出：每次移动的方格

def search(way,g,bound):

    f=g+h(node)

    if(f>bound): return f

    if(node==goal): return-1

    min=6666

    for i in children(node):

        if i in close:

            continue

        way.append(i)

        t=dfs(way,g+1,bound)

        if t==-1: return -1

        if t<min: min=t

        way.pop() #找不到路 删除该待访问节点

        close.remove()

def IDA(start):

    bound=h(start)

    way=[start]  #更新的路径合集

    close.add(tuple(start))

    while(1):

        t=dfs(way,0,bound)

        if t==-1:

            return way

        if t>60:

            return None

        bound=t #更新bound

1. 关键代码展示（带注释）

**A\*算法：**

由于A\*算法开放列表中需要记录的成分较多，所有这里定义了一个node类：

class node():

    def \_\_init\_\_(self,f,arr,path,g):

        self.f=f

        self.arr=arr

        self.path=path

        self.g=g

    def \_\_lt\_\_(self, node): # heapq的比较函数 后续选最小节点时需要

        if self.f == node.f:

            return self.g > node.g

        return self.f < node.f

启发式函数：这里用了曼哈顿距离、曼哈顿距离+线性冲突两种启发式函数，代码如下：

def man\_dist(arr):  #计算曼哈顿距离

    ans=0

    for i in range(0,16):

        if arr[i]!=0 and arr[i]!=goal[i]:

            x=(arr[i]-1)//4

            y=arr[i]-4\*x-1

            ans+=abs(x-(i//4))+abs(y-(i%4))

    return ans

def linear\_conflict(arr):

    board=[arr[0:4],arr[4:8],arr[8:12],arr[12:16]]

    n = len(board)

    linear\_conflicts = 0

    # 计算每一行的线性冲突

    for i in range(n):

        for j in range(n):

            tile1 = board[i][j]

            if tile1 == 0:

                continue

            goal\_row1, goal\_col1 = (tile1 - 1) // n, (tile1 - 1) % n

            if goal\_row1 == i:  # 需要在同一行内进一步检查

                for k in range(j + 1, n):

                    tile2 = board[i][k]

                    if tile2 == 0:

                        continue

                    goal\_row2, goal\_col2 = (tile2 - 1) // n, (tile2 - 1) % n

                    if goal\_row2 == i and goal\_col2 < goal\_col1:

                        linear\_conflicts += 1

    # 计算每一列的线性冲突

    for j in range(n):

        for i in range(n):

            tile1 = board[i][j]

            if tile1 == 0:

                continue

            goal\_row1, goal\_col1 = (tile1 - 1) // n, (tile1 - 1) % n

            if goal\_col1 == j:  # 需要在同一列内进一步检查

                for k in range(i + 1, n):

                    tile2 = board[k][j]

                    if tile2 == 0:

                        continue

                    goal\_row2, goal\_col2 = (tile2 - 1) // n, (tile2 - 1) % n

                    if goal\_col2 == j and goal\_row2 < goal\_row1:

                        linear\_conflicts += 1

    return man\_dist(arr) + linear\_conflicts

def h(arr):  #h(n) 应用不同的启发函数时就改变这里

    # return man\_dist(arr)

    return linear\_conflict(arr)

A\*算法本身的主要函数就比较简单，分别是遍历函数、访问邻接结点的函数。如下所示：

def A\_star(start): #遍历用

    n1=node(h(start),start,"",0)

    open=[n1]

    heapq.heapify(open)  #用堆来记录，省时

    history=0

    while len(open)!=0:

        top=heapq.heappop(open)

        history+=1

        if flag!=-1:

            break

        for i in move(top.arr,top.path,history):

            if tuple(i[0]) in close:continue

            close.add(tuple(i[0]))

            child=node(top.g+h(i[0]),i[0],str(top.path+i[1]+" "),top.g+1)

            heapq.heappush(open,child)

    if len(open)==0: print("no solution")

def move(arr\_,path,history): #生成子节点

    for i in range(0,16): #找到0的坐标

        if arr\_[i]==0:

            index=i

            break

    ans=[]

    x=index//4

    y=index%4

    for i in range(0,4):

        fx=next[i][0]

        fy=next[i][1]

        new\_index=index+fx\*4+fy

        if x+fx>-1 and x+fx<4 and y+fy>-1 and y+fy<4:

            arr=copy.deepcopy(arr\_)

            arr[index]=arr[new\_index]

            arr[new\_index]=0

            ans.append([arr,str(arr[index])])

            if arr==goal:

                global flag

                flag=1

                print(path+str(arr[index])+" ")

                print(history)

    return ans

IDA\*算法：

IDA\*算法的启发式函数部分与A\*算法的启发式函数部分完全相同，所以这里不再赘述。

IDA\*的主体函数部分也可概括为三部分：遍历部分（每次超过阈值返回后，更新阈值并重新递归的函数）、递归部分（在没超过阈值前一直递归的函数）以及生成子节点的部分。三个部分具体代码如下：

def IDA(start): #遍历

    bound=h(start)

    way=[start]  #更新的路径合集

    close.add(tuple(start))

    while(1):

        t=dfs(way,0,bound)

        if t==-1:

            return way

        if t>60:

            return None

        bound=t #更新bound

def dfs(way,g,bound): #递归

    global history

    node=way[-1]

    f=g+h(node)

    history+=1

    if f>bound:

        return f

    if node==goal:

        return -1 #找到目标

    min=6666

    for i in children(node):

        k=tuple(i)

        if k in close: continue

        way.append(i)

        close.add(k) #路经检测

        t=dfs(way,g+1,bound)

        if t==-1:

            return -1

        if t<min:

            min=t

        way.pop()

        close.remove(k)

return min

def children(arr\_):

    index=0

    for i in range(0,16):

        if arr\_[i]==0:

            index=i  #找到0的坐标

    ans=[]

    x=index//4

    y=index%4

    for i in range(0,4):

        fx=next[i][0]

        fy=next[i][1]

        new\_index=index+fx\*4+fy

        if x+fx>-1 and x+fx<4 and y+fy>-1 and y+fy<4:

            arr=copy.deepcopy(arr\_)

            arr[index]=arr[new\_index]

            arr[new\_index]=0

            ans.append(arr)

    return sorted(ans,key=lambda x:h(x))

1. 创新点&优化（如果有）

优化比较大的三个部分是：1、定义了node类，2、用堆来维护待放问结点，3、用集合来维护已访问结点，4、从二维列表改成一维的

* 由于open表中需要存放的每个节点状态非常多，容易搞混，且后续的堆排序中也需要node类中的排序算法，所以在此定义了node类
* 最开始的时候是直接用列表来记录待访问节点，但是这样一来每次扩展新的节点并进行排序时会非常耗时。特别是搜索后期open表中几万的数据，还要一个个排序的话对时间消耗非常大，也是这个原因所以在优化前案例三和案例四要跑若干小时。
* 用集合的好处一是不用挨个比对，节省时间；二是不会记录重复的结点，可以节省用于查重的时间。
* 二维数组在搜索的深度加大时会有非常巨大的开销，导致算法速度非常慢。在群里看到了其他同学的建议后我改用了一维。

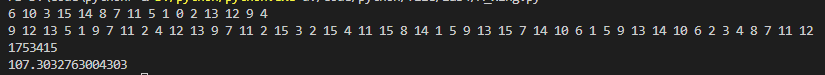
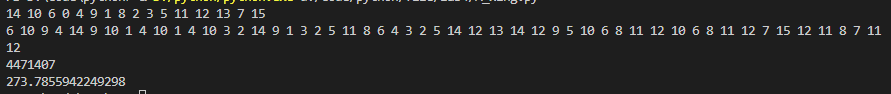
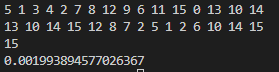
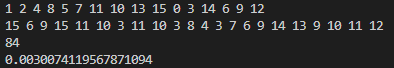
# 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

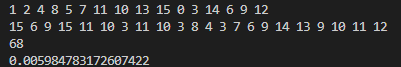
从上到下：第一行为输入的测试用例，第二行为每次要移动的数字块，第三行为曾访问过的节点数，第四行为搜索用时

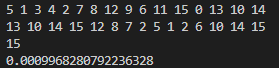
**A\*算法：**

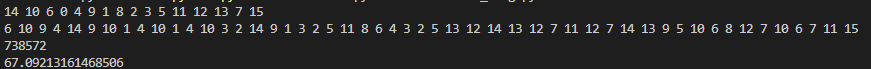
H（x）：曼哈顿函数

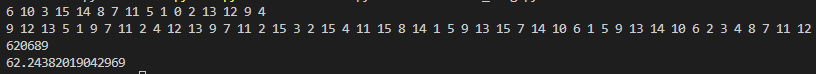


H（x）：曼哈顿函数+线性冲突

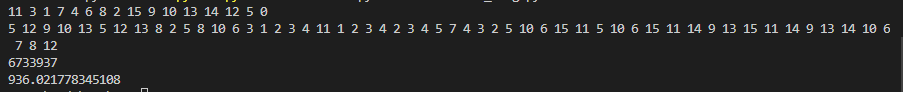


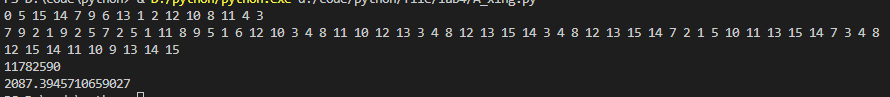






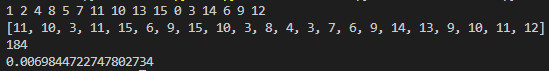
A\*算法 挑战案例 H（x）：曼哈顿距离+线性冲突：

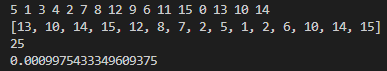


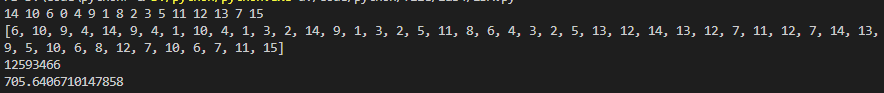


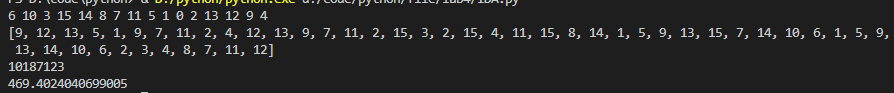
**IDA\*算法：**

H（x）：曼哈顿函数

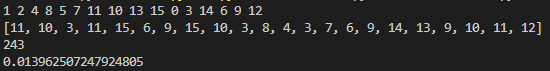


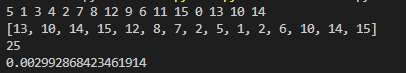


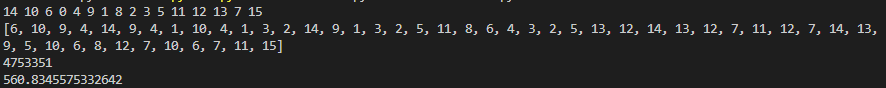


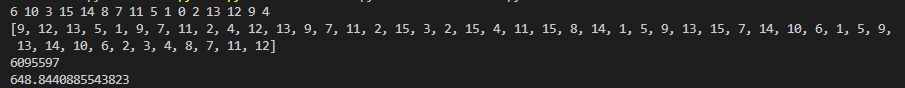


H（x）：曼哈顿函数+线性冲突

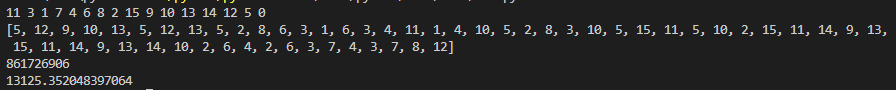


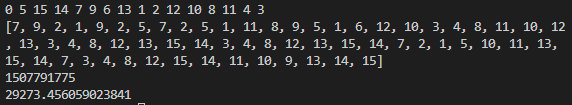






A\*算法 挑战案例 H（x）：曼哈顿距离+线性冲突

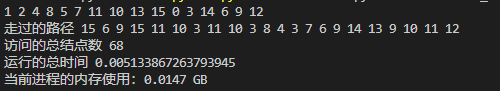




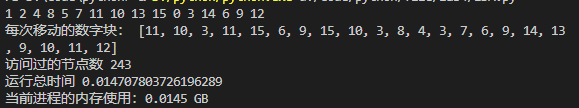
2. A\*算法与IDA\*算法的对比与分析

为了控制变量，两种算法采用相同的启发式函数（md+线性冲突），且尽量减少非主体函数以外的函数调用。对于同一个测试样例，两者运行结果如下所示：

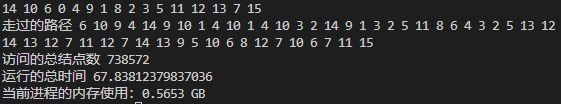
A\*算法对样例1



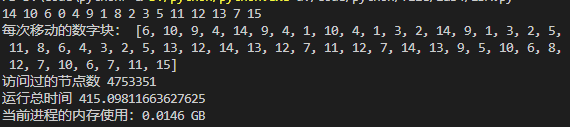
IDA\*算法对样例1：



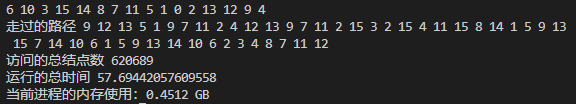
A\*算法对样例3：



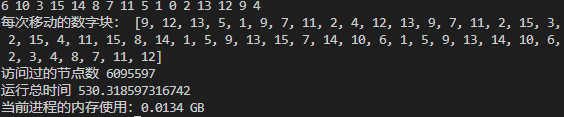
IDA\*算法对样例3：



A\*算法对样例4：



IDA\*算法对样例4：



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 测试用例 | 访问节点A\* | 访问节点IDA\* | 用时A\* | 用时IDA\* | 内存使用A\*/GB | 内存使用IDA\*/GB |
| 1 | 68 | 243 | 0.0051338 | 0.0147078 | 0.0147 | 0.0145 |
| 3 | 738572 | 4753351 | 67.838123 | 415.09811 | 0.5653 | 0.0146 |
| 4 | 620689 | 6095597 | 57.694420 | 530.31859 | 0.4512 | 0.0134 |

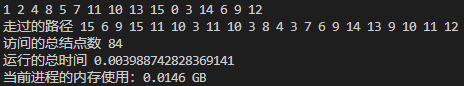
对比可知，在搜索树结构较为简单的时候，A\*算法和IDA\*算法的性能类似。但当搜索树结构复杂起来时，二者的优劣就会明显表现出来。在搜索过程中A\*算法需要占用大量内存去保存中间状态，与之相应的就是A\*算法的速度要更快；IDA\*算法不需要去储存中间状态，节省了内存开销，但是也需要花费大量的时间去重复搜索已经搜索过的结点。在挑战案例的搜索中这一点就更明显：A\*算法能很快搜索出结果，而IDA\*算法却需要以小时为单位的时间去搜索。

3. A\*算法下的启发式函数对比与分析

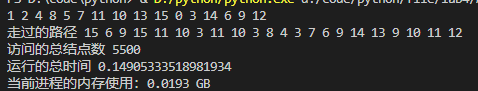
为了控制变量，我在这里仅用A\*算法来说明。使用到的启发式函数有：曼哈顿距离、错牌数、曼哈顿距离+线性冲突

对于样例一，三者性能如下所示：

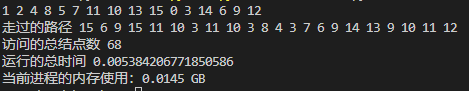
曼哈顿距离：



错牌数：



曼哈顿距离+线性冲突：



对比可知，用曼哈顿距离+线性冲突时算法效率最高，用错牌数时算法效率最低。这是因为前者能够很好的模拟一个数字块移动到另一个数字块大概需要几步，估计值和真实值的差距较小，因此能够很好的模拟真实情况，后者则不然。

4. 分析估价值与真实值的差距会造成算法性能差距的原因



拿A\*算法举例：在 A\* 算法中，启发式函数用于估算从当前节点到目标节点的距离。如果启发式函数能够准确地估算出这个距离，那么 A\* 算法就能够快速地找到最短路径。但是，如果启发式函数的估计值与真实值存在较大的差距（如错牌数），那么算法的性能就会受到影响，可能会导致算法失效或找到次优解。

例如，在使用欧几里得距离作为启发式函数的 A\* 算法中，如果节点之间存在障碍物（如被其他的数字块或边框堵住能扩展的位置），那么实际到达目标节点的距离可能会比启发式函数估计的距离更长。这种情况下，如果启发式函数的估计值与真实值存在较大的差距，那么 A\* 算法可能会忽略一些更优的路径，导致找到的解不是最短路径。同样的，若是使用错牌数，由于一个牌到其目标位置不一定是可以一步完成的，而在这个启发式函数中统一算为1步，这就可能导致真实代价远远大于错牌数，从而降低算法效率

因此，在设计A\*算法时，启发式函数的好坏对A\*算法的性能影响会非常大。在设计启发式函数时，需要确保估计值与真实值的差距尽可能小，以提高算法的性能和效率。同时，根据实际情况选择合适的启发式函数也非常重要，不同的问题可能需要不同的启发式函数来保证算法的正确性和性能。

# 参考资料

超算习堂-人工智能-课时十一第五讲

超算习堂-人工智能-课时十三第七讲-实验课