中山大学数据科学与计算机学院

计算机科学与技术专业-人工智能

本科生实验报告

（2018-2019 学年秋季学期）

课程名称：**Artificial Intelligence**

教学班级	计科 2 班	专业（方向）	计算机科学与技术
学号	16337341	姓名	朱志儒

实验题目

启发式搜索

实验内容

· 算法原理

启发式搜索

启发式搜索又叫有信息的搜索，利用问题所拥有的启发信息来引导搜索，达到减少搜索范围，降低问题复杂度的目的。无信息搜索对所有的可能路径节点一视同仁，启发式搜索可指导搜索向最有希望的方向前进。

我们使用估价函数$f(x) = g(x) + h(x)$来估计节点的重要性，其中$g(x)$是从初始节点到节点 x 付出的实际代价，$h(x)$是从节点 x 到目标节点的最优路径的估计代价。$h(x)$代表启发式

搜索问题中的启发信息，是算法的关键，合理的定义 h(x)可让整个搜索算法找到问题的一个最优解。

A* 搜索算法

A* 搜索算法在 BFS 算法的基础上加入了启发式信息。算法步骤如下：

1. 从起始节点开始，将其作为待处理的节点加入“开启列表”；
2. 从“开启列表”中选取估价函数值最小的节点 C，将其加入“关闭列表”并从“开启列表”中删除；
3. 检查节点 C 的所有相邻节点，如果节点在“关闭列表”中，则检查另一相邻节点；如果节点不在“开启列表”中，则计算其 g(x)和 h(x)，设置其父节点为 C，将其加入“开启列表”；如果节点在“开启列表”中，则更新其 g(x)；
4. 检查“开启列表”是否为空，若为空，则返回 NOT_FOUND；若不为空，则继续下一步；
5. 判断“开启列表”中是否存在目标节点，若存在，则返回 FOUND；若不存在，则重复步骤 2。

IDA* 搜索算法

IDA* 搜索算法是迭代加深搜索算法（IDS）的一个扩展，由于不用维护列表，所有其空间复杂度远小于 A* 搜索算法，算法步骤如下：

1. 将阈值设为初始节点的 h(x)值；
2. 从起始节点开始进行深度受限搜索
3. 如果没有找到目标节点，则将阈值设为步骤 2 中返回的值，再重复步骤 2；如果找到目标节点，则结束搜索。

深度受限搜索算法步骤：

1. 判断节点的 f(x)值是否大于阈值，若大于，则返回该 f(x)值，若不大于，则进入下一步；

2. 判断节点是否为目标节点，若是，则返回 FOUND，若不是，则进入下一步；

3. 检查节点的所有相邻节点，计算它们的 g(x)和 h(x)，对于每个相邻节点重复步骤 1。

· 伪代码

A* 搜索算法

```
    function A_Star(start, end)
        closeset := {}
        openset := {start}
        father_node := 空的 map
        g_value := 默认值为无穷的 map
        g_value[start] := 0

        f_value := 默认值为无穷的 map

        f_value[start] := heuristic_cost_estimate(start, end)
        while openset is not empty

            current_node := openset 中 f_value 值最小的节点

            if current_node = end
                return reconstruct_path(father_node, current_node)
            openset.remove(current_node)
            closeset.add(current_node)
            for each neighbor_node of current_node
                if neighbor_node in closeset
                    continue
                tentative_g_value := g_value[current_node] +
dist_between(current_node, neighbor_node)
                if neighbor_node not in openset
                    openset.add(neighbor_node)
                else if tentative_g_value >= g_value[neighbor_node]
                    continue
                father_node[neighbor_node] := current_node
                g_value[neighbor_node] := tentative_g_value
                f_value[neighbor_node] := g_value[neighbor_node] +
heuristic_cost_estimate(neighbor_node, end)
```

IDA* 搜索算法

```
procedure ida_star(root)
    bound := h_value[root]
    path := [root]
    loop
        t := search(path, 0, bound)
        if t = FOUND
            return (path, bound)
        if t = infinity
            return NOT_FOUND
        bound := t
    end loop
end procedure

function search(path, g_value, bound)
    node := path.last
    f_value := g_value + h_value(node)
    if f_value > bound
        return f_value
    if is_end(node)
        return FOUND
    min := infinity
    for succ in successors(node)
        if succ not in path
            path.push(succ)
            t := search(path, g_value + cost(node, succ), bound)
            if t = FOUND
                return FOUND
            if t < min
                min := t
            path.pop()
    return min
```

· 关键代码

1) 节点的定义

```
class Node:
    def __init__(self, matrix, size):
        self.matrix = matrix
        self.size = size
        self.father = None
```

```
        self.g = 0
        self.h = 0
        self.change_number = None

    def set_new_h(self, final_matrix):
        # 矩阵中每个点距目标位置的曼哈顿距离的和作为 H 值
        for x in range(self.size):
            for y in range(self.size):
                for i in range(self.size):
                    for j in range(self.size):
                        if self.matrix[x][y] == final_matrix[i][j]:
                            self.h += abs(x - i) + abs(y - j)
```

2) A* 搜索算法

找出“开放列表”中 F 值最小的节点:

```
    def find_min_f_node(self):
        # 找出“开放列表”中 F 值最小的节点
        min_node = self.openlist[0]
        for item in self.openlist:
            if item.g + item.h < min_node.g + min_node.h:
                min_node = item
        return min_node
```

判断节点是否在 list 中:

```
    def node_in_list(self, node, list):
        # 判断节点是否在 list 中
        for item in list:
            if item.matrix == node.matrix:
                return True
        return False
```

获得节点 node 在“开放列表”中的下标

```
    def get_index_in_openlist(self, node):
        # 获得节点 node 在“开放列表”中的下标
        for i in range(len(self.openlist)):
            if node.matrix == self.openlist[i].matrix:
                return i
```

```
        return None
```

检测一个相邻节点：

```
    def search_node(self, node):
        # 检测相邻节点 node
        if self.node_in_list(node, self.closelist):
            # 节点 node 在“关闭列表”中，直接返回
            return
        if not self.node_in_list(node, self.openlist):
            # 节点 node 不在“开启列表”中，计算其 G 和 H，设置其父节点为当前节点
            node.g = self.step
            node.set_new_h(self.final_matrix)
            node.father = self.current_node
            self.openlist.append(node)
        else:
            # 节点 node 在“开启列表”中，更新其 g(x)
            index = self.get_index_in_openlist(node)
            if self.current_node.g + self.step < self.openlist[index].g:
                self.openlist[index].g = self.current_node.g + self.step
                self.openlist[index].father = self.current_node
```

搜索路径：

```
    def search(self):
        self.start_node.set_new_h(self.final_matrix)
        self.start_node.g = self.step
        self.openlist.append(self.start_node)
        while True:
    # 从“开启列表”中选取估价函数值最小的节点，将其加入“关闭列表”并从“开启列表”中删除
            self.current_node = self.find_min_f_node()
            self.closelist.append(self.current_node)
            self.openlist.remove(self.current_node)
            self.step = self.current_node.g
            current_matrix = self.current_node.matrix
            x, y = find_zero(current_matrix, self.size)
            self.step += 1
            # 检查节点 C 的所有相邻节点
            if x + 1 < self.size:
```

```
                next_node = Node(move_zero(deepcopy(current_matrix), x,
y, x + 1, y), self.size)
                # 检测相邻节点
                self.search_node(next_node)
                next_node.change_number = current_matrix[x + 1][y]
            if x - 1 >= 0:
                next_node = Node(move_zero(deepcopy(current_matrix), x,
y, x - 1, y), self.size)
                # 检测相邻节点
                self.search_node(next_node)
                next_node.change_number = current_matrix[x - 1][y]
            if y + 1 < self.size:
                next_node = Node(move_zero(deepcopy(current_matrix), x,
y, x, y + 1), self.size)
                # 检测相邻节点
                self.search_node(next_node)
                next_node.change_number = current_matrix[x][y + 1]
            if y - 1 >= 0:
                next_node = Node(move_zero(deepcopy(current_matrix), x,
y, x, y - 1), self.size)
                # 检测相邻节点
                self.search_node(next_node)
                next_node.change_number = current_matrix[x][y - 1]
            # 判断“开启列表”中是否存在目标节点
            if self.node_in_list(Node(self.final_matrix, self.size),
self.openlist):
                # 若存在，结束搜索，返回路径
                index =
self.get_index_in_openlist(Node(self.final_matrix, self.size))
                tmp = self.openlist[index]
                path = [tmp]
                while tmp.father != None:
                    tmp = tmp.father
                    path.append(tmp)
                path.reverse()
                del path[0]
                return path
            elif len(self.openlist) == 0:
```

```
            # 若“开启列表”为空，结束搜索
            return
```

3) IDA* 搜索算法

更新新节点的 G 和 H 值:

```
def search_node(self, matrix, depth):
    # 更新新节点的 G 和 H 值
    node = Node(matrix, self.size)
    node.g = depth
    node.set_new_h(self.finalnode.matrix)
    return node
```

深度受限搜索:

```
def subsearch(self, node, pre, depth, limit):
    if node.g + node.h > limit:
        # 节点的 F 值大于阈值，结束搜索，将该 F 值加入 cutoff 列表
        self.cutoff.append(node.g + node.h)
        return
    if node.matrix == self.finalnode.matrix:
        # 节点是目标节点，结束搜索，保存父节点
        node.father = pre
        self.flag = True
        self.endnode = node
        return
    # 记录父节点
    node.father = pre
    # 将节点加入已访问的列表
    self.visited.append(node.matrix)
    x, y = find_zero(node.matrix, self.size)
    # 递归搜索所有相邻节点
    if x + 1 < self.size and not self.flag:
        next_matrix = move_zero(deepcopy(node.matrix),x, y, x + 1, y)
        if next_matrix not in self.visited:
            # 节点不在已访问的列表中，更新新节点的 G 和 H 值
            next_node = self.search_node(next_matrix, depth)
            next_node.change_number = next_matrix[x][y]
```

```
                self.subsearch(next_node, node, depth + 1, limit)
        if x - 1 >= 0 and not self.flag:
            next_matrix = move_zero(deepcopy(node.matrix),x, y, x - 1, y)
            if next_matrix not in self.visited:
                # 节点不在已访问的列表中，更新新节点的 G 和 H 值
                next_node = self.search_node(next_matrix, depth)
                next_node.change_number = next_matrix[x][y]
                self.subsearch(next_node, node, depth + 1, limit)
        if y + 1 < self.size and not self.flag:
            next_matrix = move_zero(deepcopy(node.matrix),x, y, x, y + 1)
            if next_matrix not in self.visited:
                # 节点不在已访问的列表中，更新新节点的 G 和 H 值
                next_node = self.search_node(next_matrix, depth)
                next_node.change_number = next_matrix[x][y]
                self.subsearch(next_node, node, depth + 1, limit)
        if y - 1 >= 0 and not self.flag:
            next_matrix = move_zero(deepcopy(node.matrix),x, y, x, y - 1)
            if next_matrix not in self.visited:
                # 节点不在已访问的列表中，更新新节点的 G 和 H 值
                next_node = self.search_node(next_matrix, depth)
                next_node.change_number = next_matrix[x][y]
                self.subsearch(next_node, node, depth + 1, limit)
        # 回溯时将节点从已访问列表中删除
        self.visited.remove(node.matrix)
```

搜索路径：

```
    def search(self):
        # 阈值设为初始节点的 H 值
        limit = self.startnode.h
        while not self.flag:
            # 没有找到目标节点，将阈值设为 cutoff 中最小值
            self.subsearch(self.startnode, None, 1, limit)
            limit = min(self.cutoff)
            self.cutoff = []
        tmp = self.endnode
        # 找到目标节点，返回路径
        path = [tmp]
        while tmp.father != None:
```

```
            tmp = tmp.father
            path.append(tmp)
        path.reverse()
        del path[0]
        return path
```

实验结果及分析

· 实验结果展示

曼哈顿距离：

IDA* 算法

```
  0  5  1  7
  2 11  4  3
  9 13  6 15
 10 14 12  8
时间：  7.4798805713653564
移动步数：  32
路径：
  5  1  4  6 15  8 12 14 13 11
  2  9 10 13 14 15 11 10  9  5
  1  2  6  4  7  3  4  7  3  4
  8 12
```

```
  5  1  2  4
  9  6  3  8
 13 15 10 11
 14  0  7 12
时间：  0.016954660415649414
移动步数：  14
路径：
 15 10  7 15 14 13  9  5  1  2
  3  7 11 12
```

```
  5  7 11  0
  9  1 15  3
 13  2 14  4
 10  6 12  8
时间：  0.06283283233642578
移动步数：  27
路径：
  3  4  8 12 14 15 11  7  1  2
  6 10 13  9  5  1  2  6 10 14
 15 11  7  3  4  8 12
```

A* 算法

```
  0  5  1  7
  2 11  4  3
  9 13  6 15
 10 14 12  8
时间：  73.74794316291809
移动步数：  32
路径：
  5  1  4  3  7  4  3 11  2  9
 13  6 12 14 10 13  9  5  1  2
  6 10 14 12 15  8 12 15 11  7
  8 12
```

```
  5  1  2  4
  9  6  3  8
 13 15 10 11
 14  0  7 12
时间：  0.03091716766357422
移动步数：  14
路径：
 15 10  7 15 14 13  9  5  1  2
  3  7 11 12
```

```
  5  7 11  0
  9  1 15  3
 13  2 14  4
 10  6 12  8
时间：  0.05580639839172363
移动步数：  27
路径：
  3  4  8 12 14 15 11  7  1  2
  6 10 13  9  5  1  2  6 10 14
 15 11  7  3  4  8 12
```

从图中可以看出 IDA* 算法和 A* 算法均可以找到最优路径，当移动步数较小时，两种

算法运行时间差距不大，但当移动步数较大时，IDA* 算法比 A* 算法用时明显较少。

下图是 PPT 中样例 IDA* 算法运行的结果，而 A* 算法并不能在相同的时间内找到路径。

样例 4

```
 6 10  3 15
14  8  7 11
 5  1  0  2
13 12  9  4
时间:  24358.499668359756
移动步数:  48
路径:
 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
```

样例 2

```
14 10  6  0
 4  9  1  8
 2  3  5 11
12 13  7 15
时间:  5070.928587913513
移动步数:  49
路径:
 6 10  9  4 14  9  4  1 10  4
 1  3  2 14  9  1  3  2 13 12
14 13  5 11  8  6  4  3  2  5
12  7 11 12  7 14 13  9  5 10
 6  8 12  7 10  6  7 11 15
```

欧氏距离（IDA* 算法）：

欧式距离

```
 2  3  4  8
 1  6  7  0
 5 10 11 12
 9 13 14 15
时间:  0.00797891616821289
移动步数:  10
路径:
 8  4  3  2  1  5  9 13 14 15
```

曼哈顿距离

```
 2  3  4  8
 1  6  7  0
 5 10 11 12
 9 13 14 15
时间:  0.010976791381835938
移动步数:  10
路径:
 8  4  3  2  1  5  9 13 14 15
```

```
 5  1  2  4
 9  6  3  8
13 15 10 11
14  0  7 12
时间:  0.08377528190612793
移动步数:  14
路径:
15 10  7 15 14 13  9  5  1  2
 3  7 11 12
```

```
 5  1  2  4
 9  6  3  8
13 15 10 11
14  0  7 12
时间:  0.016954660415649414
移动步数:  14
路径:
15 10  7 15 14 13  9  5  1  2
 3  7 11 12
```

```
 5  7 11  0
 9  1 15  3
13  2 14  4
10  6 12  8
时间： 6.909576177597046
移动步数： 27
路径：
 3  4  8 12 14 15 11  7  1  2
 6 10 13  9  5  1  2  6 10 14
15 11  7  3  4  8 12
```

```
 5  7 11  0
 9  1 15  3
13  2 14  4
10  6 12  8
时间： 0.06283283233642578
移动步数： 27
路径：
 3  4  8 12 14 15 11  7  1  2
 6 10 13  9  5  1  2  6 10 14
15 11  7  3  4  8 12
```

从上图中可以看出，使用欧氏距离作为启发式函数能找到最优路径，运行的时间多于使用曼哈顿距离所运行的时间。当总的移动步数大于 30 时，使用欧式距离的时间远大于使用曼哈顿距离使用的时间。

1.5 倍曼哈顿距离（IDA* 算法）：

1.5 倍曼哈顿距离

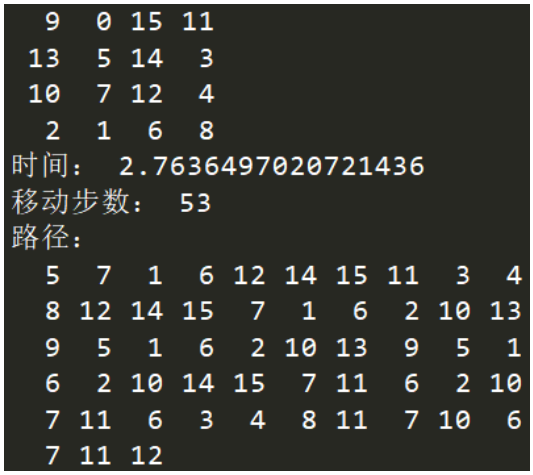
```
 0  5  1  7
 2 11  4  3
 9 13  6 15
10 14 12  8
时间： 12.042864084243774
移动步数： 36
路径：
 5  1  4  6 12  8 15 12  8 14
13 11  6  3  7  4  3  6  2  9
10 13 14 15 12  8 11 10  9  5
 1  2  6  7  8 12
```

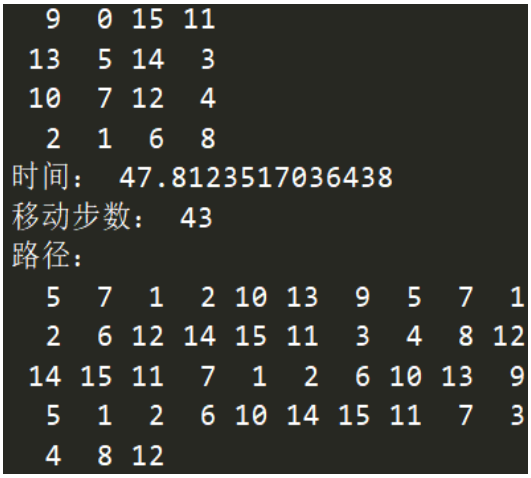
```
 5  1  2  4
 9  6  3  8
13 15 10 11
14  0  7 12
时间： 0.00993037223815918
移动步数： 14
路径：
15 10  7 15 14 13  9  5  1  2
 3  7 11 12
```

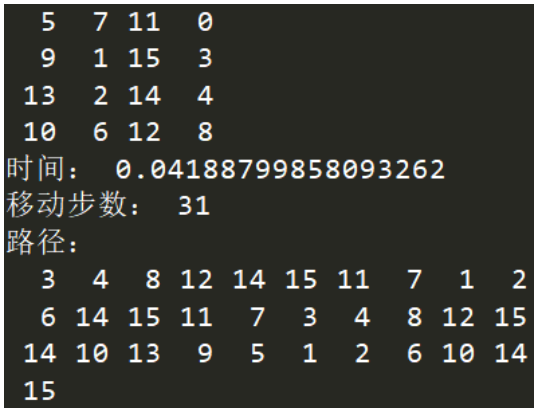
曼哈顿距离

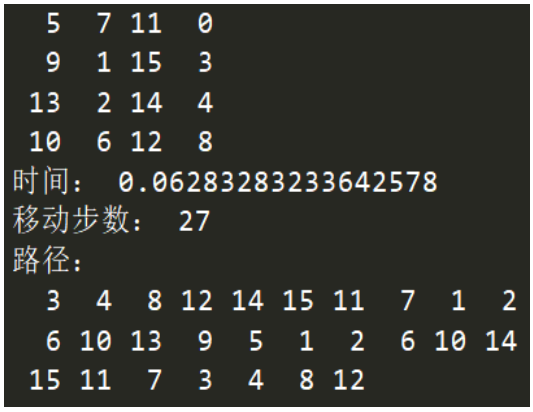
```
 0  5  1  7
 2 11  4  3
 9 13  6 15
10 14 12  8
时间： 7.4798805713653564
移动步数： 32
路径：
 5  1  4  6 15  8 12 14 13 11
 2  9 10 13 14 15 11 10  9  5
 1  2  6  4  7  3  4  7  3  4
 8 12
```

```
 5  1  2  4
 9  6  3  8
13 15 10 11
14  0  7 12
时间： 0.016954660415649414
移动步数： 14
路径：
15 10  7 15 14 13  9  5  1  2
 3  7 11 12
```









从上图可以看出 1.5 倍曼哈顿距离可以明显加快 IDA* 算法运行的速度，但它并不能确保可以找到最优路径。

· 评测指标展示

使用曼哈顿距离作为启发式函数时，h(x)满足$h(x) \le h^*(x)$，预估值小于真实值，即可采纳性，h(x)也满足$h(x) - h(x') \le \cos(x, x')$，即单调性，所以使用该启发式函数时可以找到一条从起始节点到目标节点的最短的路径。

使用欧式距离作为启发式函数时，h(x)满足 $h(x) \le h^*(x)$，预估值小于真实值，即可采纳性，h(x)也满足$h(x) - h(x') \le \cos(x, x')$，即单调性，所以使用该启发式函数时可以找到一条从起始节点到目标节点的最短的路径。

使用 1.5 倍曼哈顿距离作为启发式函数时，h(x)并不满足$h(x) \le h^*(x)$，即不满足采纳性，所以使用该启发式函数时不一定能找到一条从起始节点到目标节点的最短的路径。

设$h_1(x)$表示使用曼哈顿距离时的启发式函数，$h_2(x)$表示使用欧氏距离时的启发式函数，对于所有的节点满足$h_2(x) \le h_1(x) \le h^*(x)$，所以$h_1(x)$更具有优势，探索的节点比 $h_2(x)$更少，故使用曼哈顿距离所运行的时间小于使用欧氏距离所运行的时间。