中山大学计算机学院

人工智能本科生实验报告

课程名称：Artificial Intelligence

教学班级	网安软工合班	专业（方向）	网络空间安全
学号	20337251	姓名	伍建霖

一、实验题目

盲目搜索和启发式搜索

二、实验内容

1.算法原理

BFS

1、创建一个空队列queue（用来存放节点）和一个空列表visit（用来存放已访问的节点）

2、依次将起始点及邻接点加入queue和visit中

3、poo出队列中最先进入的节点,从图中获取该节点的邻接点

4、如果邻接点不在visit中，则将该邻接点加入queue和visit中

5、输出pop出的节点

6、重复3、4、5，直至队列为空

DFS

1、创建一个空栈stack（用来存放节点）和一个空列表visit（用来存放已访问的节点）

2、依次将起始点及邻接点加入stack和visit中

3、poo出栈中最后进入的节点,从图中获取该节点的邻接点

4、如果邻接点不在visit中，则将该邻接点加入stack和visit中

5、输出pop出的节点

6、重复3、4、5，直至栈为空

(不过我没有用stack，而用了递归的操作，也类似于stack)

UCS

扩展路径消耗最小的节点N

(有点类似于djikstra算法)

(N点的路径消耗等于前一节点N-1的路径消耗加上N-1到N节点的路径消耗)

IDDFS

IDDFS是深度优先搜索的“升级版”：每次深搜都会有搜索的最大深度限制，如果没有找到解，那么就增大深度，再进行深搜，如此循环直到找到解为止，这样可以找到最浅层的解。

双向搜索

双向搜索主要有两种：双向BFS和双向IDDFS

双向BFS类似于BFS，但它维护两个而不是一个队列，然后轮流拓展两个队列。同时用数组或哈希表记录当前的搜索情况，给从两个方向拓展的节点以不同的标记。当某点被两种标记同时标记时，搜索结束。双向IDDFS同理，从两个方向迭代加深深度搜索

IDA*

迭代加深搜索算法，在搜索过程中采用估值函数，以减少不必要的搜索。设置每次可达的最大深度depth，若没有到达目标状态则加深最大深度。采用估值函数，剪掉f(n)大于depth的路径。

A*

采用广度优先搜索策略，在搜索过程中使用启发函数

A*算法通过下面这个函数来计算每个节点的优先级。

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

其中：

- f(n)是节点n的综合优先级。当我们选择下一个要遍历的节点时，我们总会选取综合优先级最高（值最小）的节点。
- g(n) 是节点n距离起点的代价。
- h(n)是节点n距离终点的预计代价，这也就是A*算法的启发函数。关于启发函数我们在下面详细讲解。

A*算法在运算过程中，每次从优先队列中选取f(n)值最小（优先级最高）的节点作为下一个待遍历的节点。

其实A*和 IDA*就类似于BFS和DFS加上启发式函数，给他们指定一下方向

2. 伪代码

盲目搜索

```
def BFS(start):
    将start加进队列中
    while (not ret):
        point = 队列第一个，pop
        将point标记为已访问
        row = point纵坐标
        col = point横坐标
    if point == end:
        ret = 1
    若point四周的点可行 and 未访问:
        将对应点加进队列

def DFS(point):
    将point标记为已访问
    row = point纵坐标
```

```
    col = point横坐标
    可行方向fd = [1, 1, 1, 1]

    if point == end:
        ret = 1
    else:
        若point四周的点可行 and 未访问过:
            fd对应位置改为对应点的坐标

        若fd没变:
            返回上一级DFS
        否则:
            遍历fd中可行的点:
                DFS(对应点)
                若ret为1:
                    可在这里输出路径
                    fd = [1, 1, 1, 1]

def BBFS():
    h = {}
    reach = false
    que = [[], [], []]
    que[1],que[2]分别加进start和end

    for d in range(rowlen*collen):
        dir为1时正向，2时反向
        遍历que[dir]:
            point = 队列第一个，pop
            将point标记为已访问
            row = point纵坐标
            col = point横坐标

            若point四周的点可行 and 未访问:
                将对应点加进队列

            若h中对应位置的值加dir == 3:
                表示存在节点被两边都搜过了
            h[对应位置] = dir

def uniformCostSearch(problem):
    # 初始化相关参数
    result = []
    explored = set()
    frontier = util.PriorityQueue()
    # 定义起始状态，其中包括开始的位置，对应的行动方案和行动代价
    start = (problem.getStartState(), [], 0)
    # 把起始状态放进frontier队列中，update方法会自动对其中的状态按照其行动代价进行排序
    frontier.update(start,0)
    # 构造循环，循环读取frontier中的状态，进行判定
    while not frontier.isEmpty():
        # 获取当前节点的各项信息
        (node, path, cost) = frontier.pop()
        # 如果弹出的节点状态满足目标要求，停止循环
        if problem.isGoalState(node):
            result = path
            break
        # 如果该节点该节点不满足目标要求，判定其是否访问过
        if node not in explored:
```

```
            explored.add(node)
            # 遍历这个节点的子节点，更新frontier队列
            for child,direction,step in problem.getSuccessors(node):
                newPath = path + [direction]
                newCost = cost + step
                newNode = (child, newPath, newCost)
                frontier.update(newNode, newCost)
    # 返回计算结果，即一个行动方案
    return result

def IDDFS(u, d):
    # 类似于DFS
    if d > limit:
        return
    else:
        for each edge (u, v):
            IDDFS(v, d + 1)
    return
```

启发式搜索

```
def astar(原图):
    用c表示当前g值
    将原图加进open中
    while 1:
        graph1 = 优先级最高的，从open中pop # f = g + h
        将graph1加入close
        遍历graph可能变成的情况: # 用g表示
            如果它在close中:
                pass
            否则:
                如果它不在open中:
                    将g加入open
                    把当前方格设置为它的父亲
                    记录该方格的f,g,h值
                否则:
                    当前c小于其g值:
                        把它的父亲设置为当前方格
                        重新计算它的g, f
    若目标graph已在open中:
        break
    否则:
        若open为空:
            print("----fail----")
            return
ptpath()

def dfs(当前图，深度，h):
    若深度+h大于最大深度:
        return false
    若当前图 == 目标图:
        return true
    下一图
    遍历下一图的可能性:
        重新计算h值
        记录路径
        若dfs(下一图，深度+1，新h值)
```

```
            return true
    return false


def idastar():
    计算h值
    最大深度 = h
    while(not dfs(argv...))
        最大深度 = 最大深度 + 1
    return 最大深度
```

3.关键代码展示（带注释）

BFS

```
def BFS(startpoint):
    global que
    global ret
    global state
    que.append(startpoint)
    while not ret:
        cpoint = que.pop(0)  # current point
        state.append(cpoint)
        row = cpoint[0]
        col = cpoint[1]
        if row == end[0] and col == end[1]:
            print(cpoint)
            ret = 1

        if (graph[row][col-1] != "1") and (not ([row, col-1] in state)):  # west
            que.append([row, col-1])
        if (graph[row+1][col] != "1") and (not ([row+1, col] in state)):  # south
            que.append([row+1, col])
        if (graph[row][col+1] != "1") and (not ([row, col+1] in state)):  # east
            que.append([row, col+1])
        if (graph[row-1][col] != "1") and (not ([row-1, col] in state)):  # north
            que.append([row-1, col])
```

DFS

```python
def DFS(point):  # point = [row, col]
    global state
    global ret
    state.append(point)
    row = point[0]
    col = point[1]
    fedirc = [1, 1, 1, 1]

    if row == end[0] and col == end[1]:
        print(point)
        ret = 1
    else:
        if (graph[row][col-1] != "1") and (not ([row, col-1] in state)):  # west
            fedirc[0] = [row, col-1]
        if (graph[row+1][col] != "1") and (not ([row+1, col] in state)):  # south
            fedirc[1] = [row+1, col]
        if (graph[row][col+1] != "1") and (not ([row, col+1] in state)):  # east
            fedirc[2] = [row, col+1]
        if (graph[row-1][col] != "1") and (not ([row-1, col] in state)):  # north
            fedirc[3] = [row-1, col]

        if fedirc.count(1) == 4:
            pass
        else:
            for x in range(4):
                if fedirc[x] != 1:
                    DFS(fedirc[x])
                    if ret:
                        print(point)
                        fedirc = [1, 1, 1, 1]
```

BBFS

```
def BBFS():
    h = {}
    reach = false
    que = [[], [], []]
    que[1].append(start)
    que[2].append(end)
    for d in range(rowlen*collen):
        dir = (d & 1) + 1
        sz = len(que[dir])
        for i in range(sz):
            x = que[dir].pop(0)
            row = x[0]
            col = x[1]
            state.append(x)
            if (graph[row][col-1] != "1") and (not ([row, col-1] in state)):  # west
                que[dir].append([row, col-1])
            if (graph[row+1][col] != "1") and (not ([row+1, col] in state)):  # south
                que[dir].append([row+1, col])
            if (graph[row][col+1] != "1") and (not ([row, col+1] in state)):  # east
                que[dir].append([row, col+1])
            if (graph[row-1][col] != "1") and (not ([row-1, col] in state)):  # north
                que[dir].append([row-1, col])
            hash = x[0]*1000 + x[1]
            if h.get(hash, 0)+dir == 3:
                reach = true
            h[hash] = dir
    if reach:
        print("found")
    else:
        print("haven't found")
```

UCS

```
def ucs(maze, begin, end):
    dist = [[MAX for col in range(m)]for row in range(n)]
    prev = [[MAX for col in range(m)]for row in range(n)]
    dirx = [0, 1, 0, -1]
    diry = [1, 0, -1, 0]  # 四个方向各一个单位
    explore = []
    dist[begin.x][begin.y] = 0
    inqueue(begin)
    while True:
        if len(frontier) == 0:
            return
        now = frontier.pop(0)
        if now.x == end.x and now.y == end.y:
            output = now
            s = []
            s.append(now)
            print('沿途路径：')
            while True:
                pre = prev[output.x][output.y]
                s.append(pre)
                if pre.x == begin.x and pre.y == begin.y:
                    s = s[::-1]
                    for data in s:
                        print('(%d,%d)' % (data.x, data.y))
                    break
                output = pre
            return
        explore.append(Po(now.x, now.y))
        for i in range(4):
            newx, newy = now.x+dirx[i], now.y+diry[i]
            if (n > newx >= 0) and (m > newy >= 0) and (maze[newx][newy] != '1'):
                point = Po(newx, newy)
                if point not in explore and not in_frontier(point):
                    dist[newx][newy] = dist[now.x][now.y] + 1
                    prev[newx][newy] = Po(now.x, now.y)
                    inqueue(Point(newx, newy, dist[newx][newy]))
```

IDDFS

```
def ids():
    for i in range(len(maze)):
        for j in range(len(maze[i])):
            if maze[i][j] == 'S':
                sx, sy = i, j
            if maze[i][j] == 'E':
                ex, ey = i, j
    for i in range(len(maze)):
        maze[i] = list(maze[i])
    maze[sx][sy] = '0'
    maze[ex][ey] = '0'
    depth = 1  # 搜索限制深度
    while True:
        deep = 0  # 当前搜索深度
        stack = []
        stack.append((sx, sy))
        for i in range(len(maze)):
            for j in range(len(maze[i])):
                if maze[i][j] == '2':
                    maze[i][j] = '0'
        node = []
        while (len(stack)) > 0:
            curr = stack[-1]
            if curr[0] == ex and curr[1] == ey:
                print('沿途路径: ')
                for output in stack:
                    print(output)
                return True
            if deep == depth:
                node.append((curr[0], curr[1]))
                stack.pop()
                deep -= 1
                curr = stack[-1]
            for dir in func:
                next = dir(curr[0], curr[1])
                if maze[next[0]][next[1]] == '0':
                    stack.append(next)
                    maze[next[0]][next[1]] = '2'
                    deep += 1
                    break
            else:
                maze[curr[0]][curr[1]] = '2'
                for i in range(len(node)):
```

```
                    de = node[i]
                    maze[de[0]][de[1]] = '0'
                node = []
                stack.pop()
                deep -= 1
        depth += 1
```

A*

```
def a_star(start_board, heuristic):
    """
    :param start_board: Start board holds the current board
    :param heuristic: Flag to tell if manhattan or displaced heuristic
    :return: Returns true or false (if a solution was found within 15 seconds
    """
    pq = []
    visited = {}
    global HEURISTIC

    HEURISTIC = heuristic
    b = Board(start_board)

    if HEURISTIC == 'M':
        h = b.manhattan_heuristic()
    else:
        h = b.displaced_tiles_heuristic()

    cb_as_list = b.board_as_string_list
    visited[' '.join(str(e) for e in cb_as_list)] = [h, h, 'NULL']

    heappush(pq, (h, cb_as_list))
    curr_max = 0

    curr_time = time.time()

    while len(pq) != 0:

        if (time.time() - curr_time) * 1000 > 30000:
            print('Sorry... We did not find a solution within 15 seconds... Terminating the program.')
            break
        node = heappop(pq)

        if getsizeof(pq) > curr_max:
            curr_max = getsizeof(pq)

        # node[0] == 0 or (use for next project, put that or cond in if cond
        if node[1] == GOAL_STATE_15_AS_ILIST or node[1] == GOAL_STATE_15_AS_SLIST:
            get_path(GOAL_STATE_15, visited)

            if HEURISTIC == 'M':
                heur = 'Manhattan Heuristic'
            else:
                heur = 'Displaced Tiles Heuristic'

            print('*** Solution Found Using ', heur, ' A*!     ***')
            print('*** The Solution Path has been printed out for you. ***')

            process = psutil.Process(os.getpid())
            memory = process.memory_info().rss
            memory_bfs = memory / 1000000
            globals.memory_bfs = memory_bfs

            print('*** Heuristic Values are printed for each state.    ***'
```

```
while len(pq) != 0:

    if (time.time() - curr_time) * 1000 > 30000:
        print('Sorry... We did not find a solution within 15 seconds... Terminating the program.')
        break
    node = heappop(pq)

    if getsizeof(pq) > curr_max:
        curr_max = getsizeof(pq)

    # node[0] == 0 or (use for next project, put that or cond in if cond
    if node[1] == GOAL_STATE_15_AS_ILIST or node[1] == GOAL_STATE_15_AS_SLIST:
        get_path(GOAL_STATE_15, visited)

        if HEURISTIC == 'M':
            heur = 'Manhattan Heuristic'
        else:
            heur = 'Displaced Tiles Heuristic'

        print('*** Solution Found Using ', heur, ' A*!     ***')
        print('*** The Solution Path has been printed out for you. ***')

        process = psutil.Process(os.getpid())
        memory = process.memory_info().rss
        memory_bfs = memory / 1000000
        globals.memory_bfs = memory_bfs

        print('*** Heuristic Values are printed for each state.    ***'
              '\n*** Heuristic used: ', heur, '              ***'
              '\n*** Search Algorithm Used: Breadth First Search     ***')
        print('BFS Size Tree Reached: ', curr_max / 1048576)
        print('Using getsizeof on hashmap, the size is: ', getsizeof(visited), ' bytes.')
        visited.clear()
        return True

    # GET BOARD AS MATRIX
    curr_board = transform_to_matrix(node[1])

    # Find num zeros coordinates
    z_coord = find_zero(curr_board)

    # node[0] holds the parent's heuristic,
    # curr_board holds the parent board [in a matrix]
    # deepcopy(curr_board) is used for the child board
    # z_coord is the position of the 0
    move_up(node[0], curr_board, deepcopy(curr_board), z_coord, pq, visited)
    move_down(node[0], curr_board, deepcopy(curr_board), z_coord, pq, visited)
    move_left(node[0], curr_board, deepcopy(curr_board), z_coord, pq, visited)
    move_right(node[0], curr_board, deepcopy(curr_board), z_coord, pq, visited)

print('There was no solution.')

return False
```

IDA*

```
def search(path, g, bound):
    node = path[-1]
    f = g + h(node)
    if(f > bound):
        return f
    if(is_goal(node)):
        return -1

    Min = 9999
    for succ in successors(node):
        if succ not in path:
            path.append(succ)
            t = search(path, g+1, bound)
            if(t == -1):
                return -1
            if(t < Min):
                Min = t
            path.pop()

    return Min


def ida_star(root):
    bound = h(root)
    path = [root]

    while(True):
        t = search(path, 0, bound)
        if(t == -1):
            return (path, bound)
        if(t > 70):
```

```
                return ([], bound)

            bound = t
```

```
def successors(node):
    x, y = 0, 0
    for i in range(4):
        for j in range(4):
            if(node[i][j] == 0):
                x, y = i, j
    success = []
    moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    for i, j in moves:
        a, b = x+i, y+j
        if(a < 4 and a > -1 and b < 4 and b > -1):
            temp = [[num for num in col] for col in node]
            temp[x][y] = temp[a][b]
            temp[a][b] = 0
            success.append(temp)

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


def is_goal(node):
    index = 1
    for row in node:
        for col in row:
            if(index != col):
                break
            index += 1
    return index == 16
```

4.创新点&优化（如果有）

尝试了多种评估函数

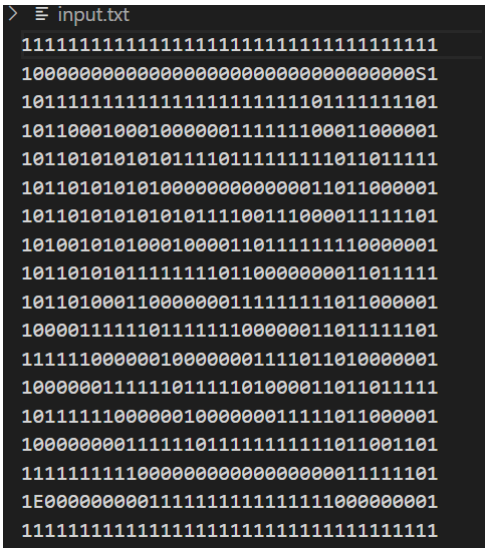
h1():当前图和目标图的不同之处的个数

h2():当前图任意节点到正确位置的步数

三、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）

盲目搜索



BFS	DFS	BBFS	UCS	IDDFS
3.998ms	35.98ms	3.990ms	500.09ms	135.99ms

启发式搜索

M表示曼哈顿函数，D表示评估位置不正确的启发式函数，单位：ms

算法	样例一	样例三	样例六	样例五
A*	M:287.9, D:2792	M:19143, D:2749	M:18381, D:6023	M:5.87s, D:4.998s
IDA*	0.008s	71.4s	大于20min	452.6s

2.评测指标展示及分析（其它可分析运行时间等）

盲目搜索

从实验结果来看，BFS = BBFS > DFS > IDDFS >UCS，但由于样例太少，不具有一般性

UCS

完备性：一致代价搜索对解路径的步数并不关心，只关心路径总代价。所以，如果存在零代价行动就可能陷入死循环。如果每一步的代价都大于等于某个小的正常数，那么一致代价搜索是完备的。从算法中也可以看到，一致搜索将地图中相关联的路径都入了队列，不存在遗漏的点线，所以如果目标结点在图中，最终总会找到一条路径到达目标结点。

最优性：一致代价搜索是最优的。一致代价搜索目标检测应用于结点被选择扩展时，而不是在结点生成的时候，因为第一个生成的目标结点可能在次优路径上。而且如果边缘中的结点有更好的路径到达该结点那么会引入一个测试。简单来说，优先队列保证了每一个出队扩展的结点都是最优的，如此得到了一个最优的路径。

时间和空间复杂度：一致代价搜索由路径代价而不是深度来导引。引入C表示最优解的代价，假设每个行动的代价至少为e，那么最坏情况下，算法的时间和空间复杂度为O（b（1＋[C / e]））。

IDDFS

好处：

1.时间复杂度只比BFS稍差一点（虽然搜索k+1层时会重复搜索k层，但是整体而言并不比广搜慢很多）。

2.空间复杂度与深搜相同，却比广搜小很多。

3.利于剪枝。

启发式搜索

在使用下面两个算法解决15puzzle问题之前，我尝试使用BFS和IDDFS来解决，但无一例外都耗时过长，无法跑出结果

A*

根据代码跑出来的结果来看，曼哈顿函数访问的节点少于misplaced的，即使用了更少的内存，而misplaced则花费了更少的时间。但从网上找的资料来看，曼哈顿函数应该是使用内存少同时花费更少的时间，与结果不符

IDA*

在我的认识中，IDA*是要优于A*的，但由于评估函数对两个算法的影响过大，导致IDA*测出来的时间多于A*

IDA*算法：

迭代加深搜索算法，在搜索过程中采用估值函数，以减少不必要的搜索。

设置每次可达的最大深度depth，若没有到达目标状态则加深最大深度。
采用估值函数，剪掉f(n)大于depth的路径。

优点：

使用回溯方法，不用保存中间状态，大大节省了空间。

缺点：

重复搜索：回溯过程中每次depth变大都要再次从头搜索。

用途：

和A*算法大致相同。

四、参考资料

A*算法视频讲解：
http://www.bilibili.com/video/av7830414/

A*算法可视化：
PathFinding.js

参考链接：

堪称最好的A*算法 - 我的程序世界 - CSDN博客
搜索--最佳优先搜索 - 沉淀，累积 - CSDN博客
八数码 poj1077 Eight（A、*IDA*)
八数码的八境界 - liugoodness - 博客园
菜鸟系列--八数码八境界 - MyWorld - CSDN博客
最短路径 A*算法 应用堆优化 - user-agent:this - CSDN博客
A*寻路极限优化 - Zzz的专栏 - CSDN博客