中山大学计算机学院

人工智能

本科生实验报告

（2021 学年春季学期）

课程名称：Artificial Intelligence

教学班级		专业（方向）	
学号		姓名	

一、 实验题目

盲目搜索与启发式搜索练习
盲目搜索：编写程序，实现深度优先搜索、广度优先搜索和自选的某一盲目搜索算法（本实验报告选取的为迭代加深搜索），来解决给定的迷宫问题。
启发式搜索：实现 A*算法和迭代 A*算法，来解决给定的 15-Puzzle 问题题目。

二、 实验内容

1. 算法原理

深度优先搜索（DFS）通过栈实现，每次从栈顶取出状态，并出栈。每个状态拓展出的状态放入栈顶，直到找到目标状态。在 DFS 过程中，若新拓展的状态存在于当前路径中，则不需要拓展，称为路径检测。消耗的空间正比于路径长度，比 BFS 更优。

宽度优先搜索（BFS）通过队列实现。每次从队头取出状态，并出队。每个状态拓展出的状态放入队尾，直到找到目标状态，利用了队列先进先出的特性。状态转移代价都相同时能保证求出最优解。消耗的空间正比于拓展出的状态数。

迭代加深搜索的前提是题目要有解。它首先深度优先搜索 k 层，若没有找到可行解，再深度优先搜索 k+1 层，直到找到可行解为止。比较像 DFS 和 BFS 二者的融合。

一致代价搜索指在 BFS 基础上，将队列改为优先队列，每次从队列中选出目前代价最小的节点拓展，在转移代价非负时能保证求出最短路径。一致代价搜索时，若拓展到的节点已经被拓展（即新子节点的最优代价更小）则该次拓展不会贡献答案，可以不执行，称为环检测。

设初始状态为 S，目标状态为 T。定义状态 x 的估计函数 $f(x) = h(x) + g(x)$。启发式函数 $h(x)$为由 x 到 T 最短路径长度的估计值，$g(x)$为由 S 到 x 目前路径的长度。

A*算法在一致代价搜索基础上，每次选取估价函数最小的状态拓展，直到找到最优解。

IDA*算法需要可接受的启发式函数，在设置上界，每次从 S 状态开始 DFS，DFS 时需要按照子状态的估价函数从小到大搜索，直到找到目标状态。若失败，则增加上界，重新开始 DFS。在估价函数离散时，能保证找到最优解。该问题启

发式函数可使用曼哈顿距离和，纵横逆序对和，或者两者取最大。但曼哈顿距离通过预处理和位运算大幅增加效率，故不同启发式函数计算复杂度有很大差别，不能单纯比较不同启发式函数对性能的影响。

下面的程序，我们选用的是曼哈顿距离和。

2. 伪代码

深度优先搜索，广度优先搜索的流程图请见压缩包中的【流程图】文件夹。

迭代加深搜索是基于深度优先搜索改进的，我们有伪代码：

```
1. function Depth-Limited-Search(problem, limit)
2.  if problem.Goal-
   Test(node.State) then return Solution(node)
3.  if limit = 0 then return cutoff    //no solution
4.  cutoff_occurred? <- false
5.     for each action in problem.Action(node.State) do
6.      child <- Child-Node(problem, node, action)
7.         result <= Recusive-DLS(child, problem, limit - 1)
8.         if result = cutoff then cutoff_occurred ? <- true
9.         else if result ≠ failure then return result
10.    if cutoff_occurred ? then return cutoff    //no solutio
   n
11. else return failure
12.
13.fuction Iterative-Deepening-Search(problem, limit)
14. for depth = 0 to ∞ do
15.     result <- Depth-Limited-Search(problem, limit)
16.        if result ≠ cutoff then return result
```

A*算法的流程图请见压缩包中的【流程图】文件夹。

迭代 A*算法是迭代加深的 A*算法，我们有伪代码：

```
1. function h():  //启发式函数
2.
3. function IDA_search(node, g, bound):  //node 为当前节点，g 为
   当前节点的路径消耗，bound 为阈值
4.     f = g + h(node)
5.     if f > bound:
6.         return f
7.     if node == E:  //E 为终点
8.         return FOUND
9.     min = r
10.    for succ in successors(node) do:
11.        t = search(succ, g + cost(node, succ), bound)
12.    if t == FOUND:
13.        return FOUND
14.    if t < min:
15.        min = t
```

```
    return min

bound = h(A)
while(True):
    t = IDA_search(root, 0, bound)
    if t == FOUND:
        return bound
    if t = ∞:
        return NOT_FOUND
    bound := t
```

3. 关键代码展示（带注释）

任务一（盲目搜索）：

首先是三种盲目搜索算法：深度优先搜索、广度优先搜索、迭代加深搜索。深度优先搜索的代码如下：

```
# 深度优先搜索
class Stack:  # 栈
    element = []

    def __init__(self):
        self.element = []

    def pop(self):
        self.element = self.element[:-1]

    def push(self, data):
        self.element.append(data)

    def empty(self):
        return len(self.element) == 0

    def size(self):
        return len(self.element)


d_xy = [[0, 0, 1, -1], [-1, 1, 0, 0]]  # 表示不同的方向


def dfs(start_x, start_y, end_x, end_y, stack_x, stack_y, data, visited, father):  # 深度优先
    for i in range(4):  # 针对上下左右四个不同方向
        x = start_x + d_xy[0][i]
```

```
27.         y = start_y + d_xy[1][i]
28.         if data[x][y] != '1' and visited[x][y] == 1:  # 可被访问
29.             visited[x][y] = 0
30.             stack_x.push(x)  # 入栈
31.             stack_y.push(y)
32.             father[x][y] = [start_x, start_y]
33.             if visited[end_x][end_y] == 0:  # 如果到达了终点
34.                 return
35.             dfs(x, y, end_x, end_y, stack_x, stack_y, data, visited, father)  # 向更深处搜索
36.             if visited[end_x][end_y] == 0:  # 到达了终点
37.                 return
38.             stack_x.pop()  # 出栈
39.             stack_y.pop()
40.     return
41.
42.
43. def readfile(filename, list_out):  # 从文件读入
44.     with open(filename, 'r') as f:
45.         for line in f.readlines():
46.             line = list(line.replace("\n", ""))
47.             list_out.append(line)
48.
49.
50. def main():
51.     list_of_data = []
52.     readfile("MazeData.txt", list_of_data)
53.     start = []
54.     end = []
55.     for i in range(len(list_of_data)):
56.         for j in range(len(list_of_data[i])):
57.             if list_of_data[i][j] == "S":  # 找起点
58.                 start.append(i)
59.                 start.append(j)
60.             elif list_of_data[i][j] == "E":  # 找终点
61.                 end.append(i)
62.                 end.append(j)
63.                 break
64.     print("起点[", start[0], ",", start[1], "]")
65.     print("终点[", end[0], ",", end[1], "]")
66.
```

```
    visited = []  # 记录所有从起点出发可访问到的点，0 代表可访问到（已被访问），1 代表不可访问到（或者未被访问），参考边界用 1 表示
    way = []  # 记录从起点出发到终点的路径上的点
    father = []  # 用于记录某点的父亲节点是谁，如 father[x][y] = [a][b] ,表示是从(a,b)出发访问到(x,y)
    for i in range(len(list_of_data)):
        temp = [1] * len(list_of_data[0])
        visited.append(temp[:])
        way.append(temp[:])
        temp = [[]] * len(list_of_data[0])
        father.append(temp[:])
    visited[start[0]][start[1]] = 0  # 表示从起点出发，起点是可达的

    stack_x, stack_y = Stack(), Stack()  # 用于记录未访问的可达点的坐标
    dfs(start[0], start[1], end[0], end[1], stack_x, stack_y, list_of_data, visited, father)

    x, y = end[0], end[1]
    while len(father[end[0]][end[1]]) != 0:  # 从终点出发回溯路径上的点,生成关键路径
        if x == start[0] and y == start[1]:
            break
        way[x][y] = 0
        x, y = father[x][y][0], father[x][y][1]

    temp = way[:]  # 关键路径上的点
    for i in range(len(temp)):  # 输出路径上的所有点
        tmp = ""
        for j in range(len(temp[i])):
            if i == 0 or j == 0 or i == len(temp) - 1 or j == len(temp[i]) - 1:  # 输出边界
                tmp += "1"
            elif i == start[0] and j == start[1]:  # 输出起点
                tmp += "S"
            elif i == end[0] and j == end[1]:  # 输出终点
                tmp += "E"
            elif temp[i][j] != 1:  # 输出路径
                tmp += str(temp[i][j])
            else:
                tmp += " "
        print(tmp)
```

```
103.
104.
105.  if __name__ == '__main__':
106.      main()
```

广度优先搜索与深度优先搜索的代码框架的不同主要在于实现的是栈的类而不是队列的类，我们只把它最关键的两部分放在下面（首先是队列的实现）：

```
1.  class Queue:  # 队列
2.      element = []
3.
4.      def __init__(self):
5.          self.element = []
6.
7.      def dequeue(self):  # 出队
8.          self.element = self.element[1:]
9.
10.     def push(self, data):
11.         self.element.append(data)
12.
13.     def empty(self):
14.         return len(self.element) == 0
15.
16.     def size(self):
17.         return len(self.element)
18.
19.     def front(self):  # 队首
20.         return self.element[0]
```

然后是广度优先搜索的实现：

```
1.  def bfs(start_x, start_y, end_x, end_y, queue_x, queue_y, d
    ata, visited, father):   # 广度优先
2.      visited[start_x][start_y] = 0
3.      for i in range(4):  # 针对上下左右四个不同方向
4.          x = start_x + d_xy[0][i]
5.          y = start_y + d_xy[1][i]
6.          if data[x][y] != '1' and visited[x][y] == 1:  # 可被
    访问
7.              visited[x][y] = 0
8.              father[x][y] = [start_x, start_y]  # 记录当前点
    的来源
9.              if visited[end_x][end_y] == 0:  # 如果到达了终点
10.                 return
11.             queue_x.push(x)  # 入队列
12.             queue_y.push(y)
13.     if queue_x.empty():  # 如果队列为空
14.         return
```

```
    x = queue_x.front()  # 获取队列中第一个点，并做出队列操作
    y = queue_y.front()
    queue_x.dequeue()
    queue_y.dequeue()
    bfs(x, y, end_x, end_y, queue_x, queue_y, data, visited, father)
    return
```

最后我们列出迭代加深算法的关键部分，它同时用到了队列和栈：

首先是在 main 函数，我们比前两个算法多出的部分：

```
    stack_x, stack_y = Stack(), Stack()  # 用于记录未访问的可达点的坐标
    high_x, high_y = Queue(), Queue()  # 用于存储 ids 中处于最大深度的可达点的坐标
    ids(start[0], start[1], end[0], end[1], stack_x, stack_y, list_of_data, visited, 0, 10, high_x, high_y, father)
```

相比深度优先搜索，它多了一个用于记录最大深度的可达点的队列。下面是迭代加深搜索算法的实现部分：

```
def ids(start_x, start_y, end_x, end_y, stack_x, stack_y, data, visited, now_depth, max_depth, high_x, high_y,
        father):  # 迭代加深
    if now_depth == max_depth:  # 使用队列记录当前最大深度的节点
        high_x.push(start_x)
        high_y.push(start_y)
        return
    for i in range(4):  # 针对上下左右四个不同方向
        x = start_x + d_xy[0][i]
        y = start_y + d_xy[1][i]
        if data[x][y] != '1' and visited[x][y] == 1:  # 可被访问
            visited[x][y] = 0
            stack_x.push(x)  # 入栈
            stack_y.push(y)
            father[x][y] = [start_x, start_y]
            if visited[end_x][end_y] == 0:  # 如果到达了终点
                return
            ids(x, y, end_x, end_y, stack_x, stack_y, data, visited, now_depth + 1, max_depth, high_x, high_y, father)
            stack_x.pop()  # 出栈
            stack_y.pop()
    if stack_x.empty():  # 如果在当前深度没找到目标位置，逐个再对最大深度节点进行 ids
        while not (high_x.empty() or high_y.empty()):
```

```
            x = high_x.front()  # 获取队列中第一个点，并做出队列操作
            y = high_y.front()
            high_x.dequeue()
            high_y.dequeue()
            ids(x, y, end_x, end_y, stack_x, stack_y, data, visited, 0, max_depth, high_x, high_y, father)
    return
```

任务二（启发式搜索）：

首先我们给出的是 A*算法的实现：

```
import time

expanded = {}
# 预处理出 2^64-1 挖掉第 i 个与第 j 个位置的状态，方便运算
pre1 = [[0 for i in range(80)] for j in range(80)]
for i in range(80):
    for j in range(80):
        pre1[i][j] = (1 << 64) - 1 - (15 << i) - (15 << j)

# 把 64 位截成 4 段，每段 16 位，预处理出每个 16 位在不同段时的贡献，实际计算时就可以只需要 4 的复杂度
pre2 = [[0 for i in range(1 << 16)] for j in range(4)]
for k in range(4):
    for i in range(1 << 16):
        for j in range(4):
            y = ((i >> (j << 2)) & 15) - 1
            if y == -1:
                continue
            pre2[k][i] += abs((y >> 2) - k) + abs((y & 3) - j)


def work(x):  # 启发式函数：曼哈顿距离和
    tmp = 0
    for i in range(4):
        tmp += pre2[i][(x >> (i << 4)) & ((1 << 16) - 1)]
    return tmp


def translate(A):  # 将列表转换为一个二进制 64 位数
    tmp = 0
    for i in range(16):
        tmp += (A[i] << (i << 2))
```

```
    return tmp


# 通过位运算算出新状态
def exchange(x, y, z):  # 交换y和z位置的数，通常y会是我们找到的0的位置
    y <<= 2
    z <<= 2
    return (x & pre1[y][z]) | (((x >> z) & 15) << y)


# 找到0的位置
def pos_0(x):
    for i in range(16):  # 对64位，每4位对比一次
        if ((x >> (i << 2)) & 15) == 0:
            return i


# 用递归方式输出方案
def outputresult(x, step):
    if expanded[x] != -1:  # 先不断回溯，从前面的步骤到后面的步骤输出
        outputresult(exchange(x, pos_0(x), expanded[x]), step - 1)
    outputfile.write("第" + str(step) + "步：\n")
    for i in range(16):
        outputfile.write(str((x >> (i << 2)) & 15) + " ")
        if i == 3 or i == 7 or i == 11 or i == 15:
            outputfile.write("\n")
    outputfile.write("\n")


def astar(S, E):  # 初始状态与最终状态
    state = [[] for i in range(128)]  # 进行状态记录

    exist = 0
    state[work(S)].append([S, pos_0(S), -1, 0])  # 分别对应的是64位表示的状态，当前0的位置，上一级状态0的位置，走到当前状态路径的长度
    while True:
        while not state[exist]:
            exist += 1
```

```
69.         [x, y, last_p_0, steps] = state[exist][-1]  # 将
    state 中的值转储
70.         state[exist].pop()  # 去除现有的"当前状态路径长度"，等
    待更新
71.         if x in expanded:
72.             continue
73.         expanded[x] = last_p_0  # 一致代价搜索的环检测
74.
75.         if x == E:  # 到达了最终状态就输出并结束
76.             print("共", steps, "步")
77.             outputfile.write("共" + str(steps) + "步,耗费
    " + str(time.perf_counter() - timer) + "秒\n")
78.             outputresult(x, steps)  # 输出到文件中
79.             return
80.
81.         # 探索当前点的4 个方向，同时防止返回上一级的状态
82.         if y - 4 >= 0 and y - 4 != last_p_0:  # 上
83.             state[steps + 1 + work(exchange(x, y, y - 4))].
    append([exchange(x, y, y - 4), y - 4, y, steps + 1])
84.
85.         if y & 3 and y - 1 != last_p_0:  # 左
86.             state[steps + 1 + work(exchange(x, y, y - 1))].
    append([exchange(x, y, y - 1), y - 1, y, steps + 1])
87.
88.         if y & 3 != 3 and y + 1 != last_p_0:  # 右
89.             state[steps + 1 + work(exchange(x, y, y + 1))].
    append([exchange(x, y, y + 1), y + 1, y, steps + 1])
90.
91.         if y + 4 < 16 and y + 4 != last_p_0:  # 下
92.             state[steps + 1 + work(exchange(x, y, y + 4))].
    append([exchange(x, y, y + 4), y + 4, y, steps + 1])
93.
94.
95.f = open("input.txt")
96.outputfile = open("output_a_star.txt", "w")
97.
98.data = []
99.for i in range(4):
100.     data += list(map(int, f.readline().split()))
101.
102.  timer = time.perf_counter()  # 用于计时
103.  end_result = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
    14, 15, 0]  # 目标结果
```

```
104.  astar(translate(data), translate(end_result))  # 使用A*算法处理
105.  print("用时：", time.perf_counter() - timer, "秒")
106.  print("步骤已输出到指定文件中")
```

由于迭代A*的实现是基于A*算法的实现的工作，所以很多函数是被重用的，我们省去这重复的部分，给出迭代A*算法的实现的关键代码：

```
1. f = open("input.txt")
2. outputfile = open("output_ida_star.txt", "w")
3.
4. data = []
5. for i in range(4):
6.     data += list(map(int, f.readline().split()))
7.
8. timer = time.perf_counter()  # 用于计时
9. end_result = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]  # 目标结果
10.S = translate(data)  # 将列表转换成64位表示状态
11.E = translate(end_result)
12.
13.lim = -2  # 防止越界的限制
14.state = []
15.finished = 0  # 标识是否到达最终状态，控制while循环
16.while not finished:
17.    lim += 2
18.    while state:
19.        state.pop()
20.    state.append([work(S), S, pos_0(S), 0])  # 分别为启发式函数，64位状态，0的位置，g(x)（或者说步数）
21.    later_state = []
22.    expanded = {}
23.
24.    while state:
25.        [xx, x, y, steps] = state[-1]  # 取栈顶，分别为启发式函数，64位状态，0的位置，g(x)(或者说步数)
26.
27.        if x in expanded:  # 对于dfs中的回溯过程，不执行拓展
28.            state.pop()
29.            del expanded[x]
30.            continue
31.        expanded[x] = 1
32.
33.        if x == E:  # 到达了最终状态就输出并结束，并利用栈内信息输出路径
```

```
34.             print("共", steps, "步")
35.             outputfile.write("共" + str(steps) + "步,耗费
    " + str(time.perf_counter() - timer) + "秒\n")
36.
37.             now_output = 0  # 从第0步开始输出
38.             last = state[0]
39.             for i in state:
40.                 steps = i[3]  # 读取该state中的g(x)
41.                 if steps > now_output:  # 筛选掉state中多出
    的部分
42.                     x = last[1]
43.                     outputfile.write("第
    " + str(now_output) + "步: \n")
44.                     now_output += 1
45.                     for j in range(16):  # 从该步的64位状态
    中解析并写入文件
46.                         outputfile.write(str((x >> (j << 2)
    ) & 15) + " ")
47.                         if j == 3 or j == 7 or j == 11 or j
     == 15:
48.                             outputfile.write("\n")
49.                     outputfile.write("\n")
50.                 last = i
51.             finished = 1  # 表示ida_star过程结束
52.             break  # 从while循环跳出
53.
54.         # 探索当前点的4个方向,进行路径检测（判断估计函数是否越
    界）
55.         if y - 4 >= 0:  # 上
56.             if exchange(x, y, y - 4) not in expanded and wo
    rk(exchange(x, y, y - 4)) + steps + 1 <= lim:
57.                 later_state.append([work(exchange(x, y, y -
     4)), exchange(x, y, y - 4), y - 4, steps + 1])
58.
59.         if y & 3:  # 左
60.             if exchange(x, y, y - 1) not in expanded and wo
    rk(exchange(x, y, y - 1)) + steps + 1 <= lim:
61.                 later_state.append([work(exchange(x, y, y -
     1)), exchange(x, y, y - 1), y - 1, steps + 1])
62.
63.         if y & 3 != 3:  # 右
64.             if exchange(x, y, y + 1) not in expanded and wo
    rk(exchange(x, y, y + 1)) + steps + 1 <= lim:
```

```
                later_state.append([work(exchange(x, y, y + 1)), exchange(x, y, y + 1), y + 1, steps + 1])

        if y + 4 < 16:  # 下
            if exchange(x, y, y + 4) not in expanded and work(exchange(x, y, y + 4)) + steps + 1 <= lim:
                later_state.append([work(exchange(x, y, y + 4)), exchange(x, y, y + 4), y + 4, steps + 1])

        later_state.sort()  # 为后继状态排序，从大到小加入栈中
        while later_state:
            state.append(later_state[-1])
            later_state.pop()

print("用时：", time.perf_counter() - timer, "秒")
print("步骤已输出到指定文件中")
```

4. 创新点&优化（如果有）

任务一（盲目搜索）：

无。主要是根据基本的算法原理来实现。

任务二（启发式搜索）：

A*算法在运行过程中会占用大量的空间。对于状态的表示，我们使用 64 位的整数来表示状态，并且进行一些预处理、使用位运算来提升运行效率，节约内存。

我们首先将 4 乘 4 的方格中的 16 个元素，按从左往右、从上到下的顺序放入列表，然后再从列表中将 16 个数字用位运算转换为用一个 64 位的整数表示的状态。

为此我们实现了几种需要用到的操作：

首先是 0 和相邻位置交换并更新状态，我们先初始化了一些值（扣去某两个位置的值）：

```
# 预处理出 2^64-1 挖掉第 i 个与第 j 个位置的状态，方便运算
pre1 = [[0 for i in range(80)] for j in range(80)]
for i in range(80):
    for j in range(80):
        pre1[i][j] = (1 << 64) - 1 - (15 << i) - (15 << j)
```

这些初始化的值，在我们将 0 与相邻位置交换时算出新状态会用到：

```
# 通过位运算算出新状态
def exchange(x, y, z):  # 交换 y 和 z 位置的数，通常 y 会是我们找到的 0 的位置
    y <<= 2
    z <<= 2
    return (x & pre2[y][z]) | (((x >> z) & 15) << y)
```

然后是计算曼哈顿距离的和。我们上面提到，我们的启发式函数用到的是曼哈顿距离，那么同样可以用预处理的方式来提升速率：

```
# 把64位截成4段，每段16位，预处理出每个16位在不同段时的贡献，实际计算时就可以只需要4的复杂度
pre2 = [[0 for i in range(1 << 16)] for j in range(4)]
for k in range(4):
    for i in range(1 << 16):
        for j in range(4):
            y = ((i >> (j << 2)) & 15) - 1
            if y == -1:
                continue
            pre2[k][i] += abs((y >> 2) - k) + abs((y & 3) - j)
```

然后我们在计算时就可以通过位运算来实现：

```
def work(x):  # 启发式函数：曼哈顿距离和
    tmp = 0
    for i in range(4):
        tmp += pre2[i][(x >> (i << 4)) & ((1 << 16) - 1)]
    return tmp
```

除此之外，由一个 64 位整数表示的状态中，找到 0 的位置（也就是空位置）也是很重要的。不过由于要每 4 位比较一次，进行一次查找比较耗费时间，会增加时间复杂度：

```
# 找到0的位置
def pos_0(x):
    for i in range(16):  # 对64位，每4位对比一次
        if ((x >> (i << 2)) & 15) == 0:
            return i
```

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）

任务一（盲目搜索）：

深度优先搜索

```
起点[ 1 , 34 ]
终点[ 16 , 1 ]
1111111111111111111111111111111111111
10000000000000000000000000000000S1
10                                  1
10  000 000                         1
10  0 0 0 0                         1
10  0 0 0 0 000000000               1
10  0 0 0 0 0       0               1
10  0 0 0 000       0               1
10  0 0 0           00000000        1
10  0 000                   0       1
10000                       0       1
1                           0       1
1                           0       1
1                           0       1
1                           0       1
1         000000000000000000        1
1E000000000                         1
1111111111111111111111111111111111111
```

广度优先搜索

```
起点[ 1 , 34 ]
终点[ 16 , 1 ]
1111111111111111111111111111111111111
1                         000000000S1
1                         0         1
1                         000       1
1                           0       1
1                   00000  0        1
1                   0   0000        1
1                   0               1
1                   00000000        1
1                          0        1
1                          0        1
1                          0        1
1                          0        1
1                          0        1
1                          0        1
1         000000000000000000        1
1E000000000                         1
1111111111111111111111111111111111111
```

迭代加深搜索

```
起点[ 1 , 34 ]
终点[ 16 , 1 ]
1111111111111111111111111111111111111
1                       000000000S1
1                       0         1
1                       000       1
1                         0       1
1                  000000  0       1
1                  00   0000       1
1                   0              1
1                   00000000       1
1                          0       1
1                          0       1
1                          0       1
1                          0       1
1                          0       1
1                          0       1
1         0000000000000000000       1
1E000000000                        1
1111111111111111111111111111111111111
```

任务二（启发式搜索）：

由于输出的结果实在太长，我们直接把运行的结果贴在下面：

-	基本样例 1	基本样例 2	基本样例 3	基本样例 4	挑战性样例 2	挑战性样例 3
步数	22 步	15 步	49 步	48 步	40 步	40 步
A* 算法	0.000765 秒	0.000399 秒	6.365 秒	15.795 秒	0.019 秒	0.016 秒
IDA* 算法	0.002612 秒	0.000495 秒	53.130 秒	39.229 秒	0.061 秒	0.019 秒

由于 A*算法会占用大量的空间，导致占用内存超出限制，所以上面的结果是经过优化后的A*算法的结果。

同时经过排查发现，其实“内存超出限制”的主要原因是因为我使用了 **32 位版本的 python**，最大也就 2G 内存，导致对于步数比较多的样例，很容易报 **MemoryError** 的错误，跑不出结果，所以最大的优化其实是重新调整了 python 环境（

2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

任务一（盲目搜索）：

从运行时间上来看，深度优先搜索效果最优，其次是迭代加深搜索（实际上对于我们这个问题，迭代加深搜索比深度优先搜索的确更为负责），耗时最长的是广度优先搜索。

从运行效果上来看，广度优先搜索的效果最好，迭代加深搜索其次，效果最差的反而是用时最短的深度优先搜索。

但是从运行时间上的分析和运行效果上的分析仅仅限制于本题所给的测试

题目，由于起点和终点位置的差异，结果也会造成不同。具体来说，我们应该看终点与起点间的位置关系。比如表现比较适中的迭代加深搜索，其实也只是比较适用于终点位置深度比较浅的情况。

任务二（启发式搜索）：

我们把运行时间贴在了上面。对于给定的测试样例，我们从运行时间上看，A*算法的表现比迭代 A*算法要好。我觉得主要在于迭代 A*算法拓展的结点数量更多，而且它在做转移时要进行路径检测，判断估计函数是否越界，本身每一步就需要耗费更多资源。

但是 A*算法需要保存拓展的结点，长时间运行会占用更多的空间，对于步数更多的情况，它的空间占用会快速增加，这时 IDA*算法会相比更具优势。

四、 参考资料

博客园 – 浅谈迭代加深算法
https://www.cnblogs.com/fusiwei/p/12236592.html
知乎 - 路径规划之 A 算法*
https://zhuanlan.zhihu.com/p/54510444
CSDN - IDA 算法(Iterative deepening A*)*
https://blog.csdn.net/xiaohu50/article/details/73431955