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	计科 2 班	专业（方向）	计算机科学与技术
学号	21311274	姓名	林宇浩

一、 实验题目

使用 A*搜索算法和 IDA*搜索算法解决十五数码问题（15-puzzle）

二、 实验内容

1. 算法原理

A*搜索算法是一种启发式搜索算法，它使用了估价函数来评估每个搜索状态的价值，并在搜索过程中优先考虑具有更高价值的状态。A*算法用于求解最短路径问题，其通过评估每个候选节点到目标节点的距离，选择最优路径，以达到最短路径的目标。A*算法综合了广度优先搜索和贪心算法的优点，同时也避免了它们的缺点。

IDA*是 A*算法的变种，其通过迭代加深搜索的方式来寻找最优解，其主要思想是在深度优先搜索的基础上使用启发式函数限制搜索的深度。IDA 算法每次对深度进行迭代，即从深度 1 开始搜索，直到找到解或者达到最大搜索深度为止。IDA*算法不需要存储搜索树的所有节点，这样可以节省空间并提高搜索效率。IDA 算法的时间复杂度与 A 算法相同，都是指数级的，但是 IDA 算法的空间复杂度较低，但是由于 IDA 算法使用深度优先搜索，可能会出现搜索路径相交的情况，导致算法搜索效率下降，可以视为是一种时间换空间的策略。

2. A*搜索算法代码展示

```
import copy
import time
import heapq
from sortedcontainers import SortedList


class Node():  # 结点类
    def __init__(self, square, g_x, parent):
        self.square = square  # 十六宫图
        self.g_x = g_x  # g(x)
        self.h_x = 0  # h(x)
        self.f_x = 0  # f(x)
        self.parent = parent  # 父结点指针

    def __lt__(self, node):  # 重载<运算符
        if self.f_x == node.f_x:
            return self.h_x < node.h_x  # f(x)一样时优先选择启发式函数小的
        else:
```

```
            return self.f_x < node.f_x

    def caculate_h_and_f(self):  # 计算 h(x)和 f(x)
        self.h_x = Heuristic_Function(self.square)
        self.f_x = self.g_x + self.h_x


def Get_Input_Square():  # 从文件中获取初始十六宫图
    Square = []
    file = open('./data4.txt', mode='r')  # 读取数据文件
    for i in range(4):
        data = file.readline().split()
        Square.append(data)
    for i in range(4):
        for j in range(4):
            Square[i][j] = int(Square[i][j])
    return Square


h_list = [[[0, 0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5], [0, 1, 0, 1, 2,
2, 1, 2, 3, 3, 2, 3, 4, 4, 3, 4], [0, 2, 1, 0, 1, 3, 2, 1, 2, 4, 3, 2, 3, 5,
4, 3], [0, 3, 2, 1, 0, 4, 3, 2, 1, 5, 4, 3, 2, 6, 5, 4]],
          [[0, 1, 2, 3, 4, 0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4], [0, 2, 1, 2, 3,
1, 0, 1, 2, 2, 1, 2, 3, 3, 2, 3], [0, 3, 2, 1, 2, 2, 1, 0, 1, 3, 2, 1, 2, 4,
3, 2], [0, 4, 3, 2, 1, 3, 2, 1, 0, 4, 3, 2, 1, 5, 4, 3]],
          [[0, 2, 3, 4, 5, 1, 2, 3, 4, 0, 1, 2, 3, 1, 2, 3], [0, 3, 2, 3, 4,
2, 1, 2, 3, 1, 0, 1, 2, 2, 1, 2], [0, 4, 3, 2, 3, 3, 2, 1, 2, 2, 1, 0, 1, 3,
2, 1], [0, 5, 4, 3, 2, 4, 3, 2, 1, 3, 2, 1, 0, 4, 3, 2]],
          [[0, 3, 4, 5, 6, 2, 3, 4, 5, 1, 2, 3, 4, 0, 1, 2], [0, 4, 3, 4, 5,
3, 2, 3, 4, 2, 1, 2, 3, 1, 0, 1], [0, 5, 4, 3, 4, 4, 3, 2, 3, 3, 2, 1, 2, 2,
1, 0], [0, 6, 5, 4, 3, 5, 4, 3, 2, 4, 3, 2, 1, 3, 2, 1]]]


def Heuristic_Function(Square):  # 启发式函数——曼哈顿距离
    h = 0
    for i in range(4):
        for j in range(4):
            h += h_list[i][j][Square[i][j]]
    return h


def Heuristic_Function_2(Square):  # 启发式函数——切比雪夫距离
    h = 0
    for i in range(4):
        for j in range(4):
            if Square[i][j] == 0:
                continue
            x = int((Square[i][j] - 1) / 4)
            y = Square[i][j] - x * 4 - 1
            h += max(abs(x - i), abs(y - j))
    return h


def move_up_0(square, i, j):  # 将空格向上移动
    new = copy.deepcopy(square)
    new[i][j] = new[i - 1][j]
    new[i - 1][j] = 0
    return new


def move_down_0(square, i, j):  # 将空格向下移动
    new = copy.deepcopy(square)
    new[i][j] = new[i + 1][j]
    new[i + 1][j] = 0
    return new
```

```
def move_left_0(square, i, j):  # 将空格向左移动
    new = copy.deepcopy(square)
    new[i][j] = new[i][j - 1]
    new[i][j - 1] = 0
    return new


def move_right_0(square, i, j):  # 将空格向右移动
    new = copy.deepcopy(square)
    new[i][j] = new[i][j + 1]
    new[i][j + 1] = 0
    return new


def expand(min_node):  # 扩展结点
    expanded_nodes = []
    for i in range(4):
        if 0 in min_node.square[i]:
            j = min_node.square[i].index(0)

            if i != 3:  # 如果空不在最上方就可以向下移动，下面类似
                expanded_nodes.append(Node(move_down_0(min_node.square, i, j),
min_node.g_x + 1, min_node))
            if i != 0:
                expanded_nodes.append(Node(move_up_0(min_node.square, i, j),
min_node.g_x + 1, min_node))
            if j != 3:
                expanded_nodes.append(Node(move_right_0(min_node.square, i, j),
min_node.g_x + 1, min_node))
            if j != 0:
                expanded_nodes.append(Node(move_left_0(min_node.square, i, j),
min_node.g_x + 1, min_node))
            return expanded_nodes


def print_squares(squares):  # 打印十六宫图
    for x in range(len(squares)):
        for i in range(4):
            for j in range(4):
                print(squares[x][i][j], end=' ')
            print('')
        print('')


def find_different(square1, square2):  # 查找两个十六宫图之间移动了哪步
    for i in range(4):
        for j in range(4):
            if square1[i][j] != square2[i][j]:
                if square1[i][j] != 0:
                    return square1[i][j]
                else:
                    return square2[i][j]


def print_moved_num(squares):  # 打印与空格交换位置的数字
    for i in range(len(squares) - 1):
        print(find_different(squares[i], squares[i + 1]), end=' ')
        if (i + 1) % 10 == 0:
            print()
    print()


def print_path(min_node):  # 打印路径
    squares_for_output = [min_node.square]
    parent = min_node.parent
    while parent != None:
        squares_for_output.insert(0, parent.square)
```

```
        parent = parent.parent
    print_squares(squares_for_output)
    print("——以上路径的图形化显示——\n")
    print("下面是路径中每次移动的数字：")
    print_moved_num(squares_for_output)


def A_star_Search(init_square):  # A*搜索
    goal_square = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15,
0]]
    open = []  # open 采用堆实现
    close = SortedList()  # close 采用有序表实现

    init_node = Node(init_square, 0, None)
    init_node.caculate_h_and_f()
    open.append(init_node)  # 把起始结点放入 open 列表

    while open != []:
        min_node = heapq.heappop(open)  # 堆顶结点即 f(x)值最小的结点

        if min_node.square == goal_square:  # 如果是目标结点则输出路径然后结束
            print_path(min_node)
            print("结点数:", len(close) + len(open))
            return

        close.add(min_node.square)  # 移动到 close 中
        expanded_nodes = expand(min_node)  # 对 f(x)值最小的结点进行扩展
        for node in expanded_nodes:
            if node.square in close:  # 如果子结点已经在 close 表中，即已经被扩展过了，
则剪枝
                continue
            else:  # 否则将子结点移动到 open 中
                node.caculate_h_and_f()
                heapq.heappush(open, node)

    print("目标状态无法到达")
    return None


init_square = Get_Input_Square()
T1 = time.time()
A_star_Search(init_square)
T2 = time.time()
print("用时:", T2 - T1)
```

3. IDA*搜索算法代码展示

```
import copy
import time
import heapq
from sortedcontainers import SortedList


class Node():  # 结点类
    def __init__(self, square, g_x, parent):
        self.square = square
        self.g_x = g_x
        self.h_x = Heuristic_Function(square)
        self.f_x = self.g_x + self.h_x
        self.parent = parent

    def __lt__(self, node):  # 重载<运算符
        if self.f_x == node.f_x:
            return self.h_x < node.h_x  # 优先选择启发式函数小的
        else:
            return self.f_x < node.f_x
```

```
def Get_Input_Square():  # 从文件中获取起始状态十六宫图
    Square = []
    file = open('./data1.txt', mode='r')  # 读取数据文件
    for i in range(4):
        data = file.readline().split()
        Square.append(data)
    for i in range(4):
        for j in range(4):
            Square[i][j] = int(Square[i][j])
    return Square


h_list = [[[0, 0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5], [0, 1, 0, 1, 2,
2, 1, 2, 3, 3, 2, 3, 4, 4, 3, 4], [0, 2, 1, 0, 1, 3, 2, 1, 2, 4, 3, 2, 3, 5,
4, 3], [0, 3, 2, 1, 0, 4, 3, 2, 1, 5, 4, 3, 2, 6, 5, 4]],
          [[0, 1, 2, 3, 4, 0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4], [0, 2, 1, 2, 3,
1, 0, 1, 2, 2, 1, 2, 3, 3, 2, 3], [0, 3, 2, 1, 2, 2, 1, 0, 1, 3, 2, 1, 2, 4,
3, 2], [0, 4, 3, 2, 1, 3, 2, 1, 0, 4, 3, 2, 1, 5, 4, 3]],
          [[0, 2, 3, 4, 5, 1, 2, 3, 4, 0, 1, 2, 3, 1, 2, 3], [0, 3, 2, 3, 4,
2, 1, 2, 3, 1, 0, 1, 2, 2, 1, 2], [0, 4, 3, 2, 3, 3, 2, 1, 2, 2, 1, 0, 1, 3,
2, 1], [0, 5, 4, 3, 2, 4, 3, 2, 1, 3, 2, 1, 0, 4, 3, 2]],
          [[0, 3, 4, 5, 6, 2, 3, 4, 5, 1, 2, 3, 4, 0, 1, 2], [0, 4, 3, 4, 5,
3, 2, 3, 4, 2, 1, 2, 3, 1, 0, 1], [0, 5, 4, 3, 4, 4, 3, 2, 3, 3, 2, 1, 2, 2,
1, 0], [0, 6, 5, 4, 3, 5, 4, 3, 2, 4, 3, 2, 1, 3, 2, 1]]]


def Heuristic_Function(Square):  # 启发式函数—曼哈顿距离
    h = 0
    for i in range(4):
        for j in range(4):
            h += h_list[i][j][Square[i][j]]
    return h


def Heuristic_Function_2(Square):  # 启发式函数—切比雪夫距离
    h = 0
    for i in range(4):
        for j in range(4):
            if Square[i][j] == 0:
                continue
            x = int((Square[i][j] - 1) / 4)
            y = Square[i][j] - x * 4 - 1
            h += max(abs(x - i), abs(y - j))
    return h


def move_up_0(square, i, j):  # 将空格向上移动
    new = copy.deepcopy(square)
    new[i][j] = new[i - 1][j]
    new[i - 1][j] = 0
    return new


def move_down_0(square, i, j):  # 将空格向下移动
    new = copy.deepcopy(square)
    new[i][j] = new[i + 1][j]
    new[i + 1][j] = 0
    return new


def move_left_0(square, i, j):  # 将空格向左移动
    new = copy.deepcopy(square)
    new[i][j] = new[i][j - 1]
    new[i][j - 1] = 0
    return new
```

```
def move_right_0(square, i, j):  # 将空格向右移动
    new = copy.deepcopy(square)
    new[i][j] = new[i][j + 1]
    new[i][j + 1] = 0
    return new


def expand(min_node):  # 扩展结点
    expanded_nodes = []
    for i in range(4):
        if 0 in min_node.square[i]:
            j = min_node.square[i].index(0)

            if i != 3:
                expanded_nodes.append(Node(move_down_0(min_node.square, i, j),
min_node.g_x + 1, min_node))
            if i != 0:
                expanded_nodes.append(Node(move_up_0(min_node.square, i, j),
min_node.g_x + 1, min_node))
            if j != 3:
                expanded_nodes.append(Node(move_right_0(min_node.square, i, j),
min_node.g_x + 1, min_node))
            if j != 0:
                expanded_nodes.append(Node(move_left_0(min_node.square, i, j),
min_node.g_x + 1, min_node))
            return expanded_nodes


def print_squares(squares):  # 打印十六宫图
    for x in range(len(squares)):
        for i in range(4):
            for j in range(4):
                print(squares[x][i][j], end=' ')
            print('')
        print('')


def find_different(square1, square2):   # 查找两个十六宫图之间移动了哪步
    for i in range(4):
        for j in range(4):
            if square1[i][j] != square2[i][j]:
                if square1[i][j] != 0:
                    return square1[i][j]
                else:
                    return square2[i][j]


def print_moved_num(squares):  # 打印与空格交换位置的数字
    for i in range(len(squares) - 1):
        print(find_different(squares[i], squares[i + 1]), end=' ')
        if (i + 1) % 10 == 0:
            print()
    print()


def print_path(min_node):  # 打印路径
    squares_for_output = [min_node.square]
    parent = min_node.parent
    while parent != None:
        squares_for_output.insert(0, parent.square)
        parent = parent.parent
    print_squares(squares_for_output)
    print("——以上路径的图形化显示——\n")
    print("下面是路径中每次移动的数字：")
    print_moved_num(squares_for_output)
```

```
goal_square = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 0]]


def IDA_star_Search(init_square, deep=1):  # IDA*搜索
    new_deep = float('inf')  # 用于更新深度限制，初始时为无穷

    init_node = Node(init_square, 0, None)
    open = [init_node]  # 把起始结点放入 open 列表
    close = SortedList()

    while open != []:
        node = heapq.heappop(open)  # 弹出 open 中第一个结点
        close.add(node.square)  # 移动到 close 中

        if node.square == goal_square:  # 如果是目标结点则输出路径然后结束
            print_path(node)
            print("结点数:", len(close) + len(open))
            return

        expanded_nodes = expand(node)  # 对这个结点进行扩展
        for child_node in expanded_nodes:
            if child_node.f_x <= deep:  # 如果在深度限制内则可以加入
                if child_node.square in close:  # 如果已经被扩展过则剪枝
                    continue
                heapq.heappush(open, child_node)
            else:  # 如果不在深度限制内，则用于更新下一轮的深度限制
                new_deep = min(child_node.f_x, new_deep)

    if new_deep == float('inf'):  # 如果下一轮的深度限制没有被更新，说明目标状态无法到
达
        print("目标状态无法到达")
        return
    else:  # 进入下一轮
        IDA_star_Search(init_square, copy.deepcopy(new_deep))


T1 = time.time()
init_square = Get_Input_Square()
IDA_star_Search(init_square)
T2 = time.time()
print(T2 - T1)
```

4. 创新点&优化

1. 当两个节点估值函数值相等时，优先选择启发式函数值更小的结点。这相当于选择了更靠近目标结点的结点，以试图更快找到目标结点。

2. 优化启发式函数的计算。在尝试自己设计启发式函数的时候，发现启发式函数的计算会占用很大一部分时间。在测试了多个启发式函数后，发现曼哈顿距离的性能最好，于是对曼哈顿距离的计算进行了优化。优化后的启发式函数添加了一个 h_list 用于直接存储每个数字在不同位置的曼哈顿距离，之后每个数字只用从 h_list 中取出距离即可，不用再计算。

3、open 使用最小堆实现。由于每次要从 open 中找到 f(x)值最小的结点，根据数据结构课的经验，这里选择了用最小堆实现 open，寻找估计函数值最小的结点的时间复杂度降至 O(logN)。

4、close 使用有序表实现，这里使用了 sortedcontainers 库中的 SortedList。由于新扩展出的结点需要判断其是否已经在 close 中存在，所以需要对 close 频繁进行查找操作。在尝试了列表查找、字典查找、集合查找、有序表查找等查找方式后，发现有序表性能最好，所以使用了有序表查找。

5、启发式函数在剪枝结束后才计算。前面说到启发函数的计算是会占很大一部分时间的，同时剪枝也是极其频繁的操作。如果一个结点在 close 表中重复了需要被剪枝，则为其

计算启发式函数的值是没有意义的。所以设计了结点初始化时不计算 h(x)，确定了不会被剪枝之后才计算 h(x)。

6、不对 open 表进行更新：（最后一段为具体说明，前三段是合理性铺垫）

这里首先先讨论一下启发式函数单调性的问题。我们知道如果一个启发式函数不满足单调性，则环检测会使得算法失去最优性，虽然可以通过将 close 表中应该重新扩展的结点更新后再放入 open 表中解决，但这会使算法的效率降低。所以启发式函数能够具有单调性是一个优势条件，这能够使我们算法设计更简洁，效率也更高。

然后我们讨论怎么判断一个启发式函数是否具有单调性。直接用数学证明是比较难的，特别是对我们自己设计的一些比较独特的启发式函数来说。在工程上，我们可以进行统计来大致判断其是否是单调的。如果一个启发式函数不是单调的，则新扩展出来的结点如果在 close 中重复了，新结点的 f(x)值可能会更小。如果一个启发式函数是单调的，则 close 中结点的 f(x)值都已经是最小的了，这是最优性所导致的。所以我们可以跑几十万个结点测试一下，看一下是否出现了新结点 f(x)值更小的情况，如果都没有出现的话，我们可以大致认定其应该是具有单调性的。如果想在工程上百分百保证没错，可以每次跑都进行检测，哪怕使用的启发式函数不是单调的，只要没有出现 f(x)值更小的情况，最优性都是可以保证的。

通过上面的办法，测试了一下使用的几个启发式函数，其都可以认定为是单调的。所以当新结点在 close 中重复时，我们直接剪枝即可。

但是我们仍然需要对 open 中重复的结点进行更新。我统计了一下，open 表中发生重复的概率并不是很低，大概有不到 10%左右的结点会在 open 表中重复。在 open 中进行查找会花费大量的时间，即使也为 open 创建一个有序表也很难让算法效率提升。所以我们选择了不对 open 表查重，直接将新结点放入 open 表。这样做的合理性是由 close 可以直接剪枝提供的，所以上面用了大量篇幅谈及 close 直接剪枝的合理性。如果 open 表中有重复的结点，那么必定 g(x)值更小的结点会先被弹出。这样的话，在重复结点中，f(x)最小的就会被最先放入 close 中，而其他重复结点就会被剪枝，效果和对 open 表进行更新是一样的。这个策略会导致 open 表中有更多的结点，不过由于 open 是用堆实现，不到 10%的增幅对 O(logN)的时间复杂度而言影响不大，然而却能剩下大量查找 open 所需要的时间，算法效率有很大的提升，这个策略相当于空间换时间思想。

优化效果估计：

	压缩包 1	压缩包 2	压缩包 3	压缩包 4
优化前 A*运行时间(s)	28.66661	16.09242	3.71964	738.65820
优化后 A*运行时间(s)	0.14501	0.05782	0.02488	0.45491
提升倍数	197.68	278.32	149.50	1623.75

可以看到，结点越多越复杂，优化越能发挥作用。在结点数最多的压缩包 4 案例中，性能提升了 1623 倍。在结点数最少的压缩包 3 案例中，性能也提升了 150 倍。

三、 实验结果及分析

1. 实验结果展示示例

（图形化显示内容比较多没有放进来，详见运行输出）

压缩包 1：

```
下面是路径中每次移动的数字：
6 11 4 14 5 8 9 5 8 3
2 6 14 8 15 7 10 4 8 15
3 2 6 14 15 10 7 3 2 6
14 15 10 7 3 2 6 10 11 12
```

压缩包 2：

```
下面是路径中每次移动的数字：
15 14 4 2 12 15 14 8 10 4
8 9 3 5 11 13 5 14 2 12
15 11 14 2 9 10 7 3 2 9
10 7 3 2 6 5 9 10 11 15
```

压缩包 3：

```
下面是路径中每次移动的数字：
11 14 9 1 12 4 1 8 2 13
15 12 8 2 3 11 14 9 6 1
2 3 11 7 13 11 7 14 10 13
14 10 9 5 1 2 3 7 11 15
```

压缩包 4：

```
下面是路径中每次移动的数字：
1 8 4 1 5 11 15 3 13 15
3 10 1 5 8 4 2 1 7 14
1 7 5 3 10 6 14 5 7 2
3 7 6 14 9 13 14 10 11 12
```

ppt2：

```
下面是路径中每次移动的数字：
6 10 9 4 14 9 4 1 10 4
1 3 2 14 9 1 3 2 5 11
8 6 4 3 2 5 13 12 14 13
12 7 11 12 7 14 13 9 5 10
6 8 12 7 10 6 7 11 15
```

ppt4:

```
下面是路径中每次移动的数字：
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
```

2. 评测指标展示及分析

	压缩包 1	压缩包 2	压缩包 3	压缩包 4	ppt1	ppt2
A*时间(s)	0.14501	0.05782	0.02488	0.45491	22.62713	138.74059
IDA*时间(s)	0.20472	0.09510	0.03785	1.03916	47.13114	199.25497
A*结点数	7196	2926	1182	22834	845467	4549898
IDA*结点数	4632	2277	985	22433	818186	2826082

（由于本人电脑为内存 8G 的轻薄本，ppt1 和 ppt3 所需内存过大所以暂时没有数据）

可以看到，IDA*搜索算法消耗的时间比 A*搜索算法多，但是 IDA*搜索产生的结点数比 A*搜索少。并且多数情况下，结点数越多，与 A*相比，IDA*能够减少的结点数也越多。其结点数减少的倍数大于其耗时增加的倍数。在压缩包 4 案例中，IDA*能够减少的结点较少，其耗时也为 A*的两倍以上。可以得知，IDA*更适用于结点数更多的情况，其能够减少的减点数越多，其性能越好。IDA*时间换空间的效果能够帮助我们在内存有限的情况下运行结点数更多的案例。

四、 参考资料

课件及 ppt