

电子科技大学  
计算机科学与工程学院

标准实验报告

(实验) 课程名称 人工智能

电子科技大学教务处制表

# 电子科技大学

# 电子科技大学

## 实验报告

实验地点：主楼 A2-412

一、实验室名称：计算机学院实验中心

二、实验项目名称：A\*算法实验

三、实验学时：5 学时

四、实验原理：

A\*算法是一种启发式图搜索算法，其特点在于对估价函数的定义上。对于一般的启发式图搜索，总是选择估价函数 $f$ 值最小的节点作为扩展节点。因此， $f$ 是根据需要找到一条最小代价路径的观点来估算节点的，所以，可考虑每个节点 $n$ 的估价函数值为两个分量：从起始节点到节点 $n$ 的实际代价 $g(n)$ 以及从节点 $n$ 到达目标节点的估价代价 $h(n)$ ，且 $h(n) \leq h^*(n)$ ， $h^*(n)$ 为 $n$ 节点到目的结点的最优路径的代价。

八数码问题是在 $3 \times 3$ 的九宫格棋盘上，摆有8个刻有1~8数码的将牌。棋盘中有有一个空格，允许紧邻空格的某一将牌可以移到空格中，这样通过平移将牌可以将某一将牌布局变换为另一布局。针对给定的一种初始布局或结构（目标状态），问如何移动将牌，实现从初始状态到目标状态的转变。如下图表示了一个具体的八数码问题求解。

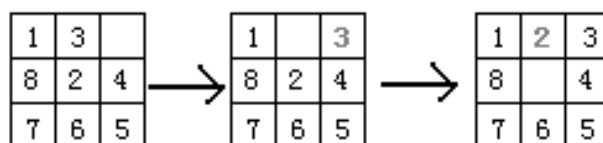


图2 八数码问题的求解

## 五、实验目的：

熟悉和掌握启发式搜索的定义、估价函数和算法过程，并利用 A\*算法求解 N 数码难题，理解求解流程和搜索顺序。

## 六、实验内容：

1. 以 8 数码问题为例实现 A\*算法的求解程序（编程语言不限），设计估价函数。

注：需在实验报告中说明估价函数，并附对应的代码。

2. 设置初始状态和目标状态，针对估价函数，求得问题的解，并输出移动过程。

要求：

（1）提交源代码及可执行文件。

（2）提交实验报告，内容包括：对代码的简单说明、运行结果截图及说明等。

## 七、实验器材（设备、元器件）：

PC 微机一台

## 八、实验步骤：

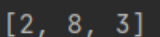
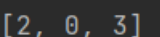
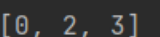
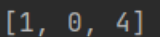
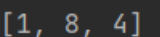
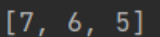
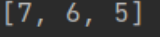
1. 估价函数设置为当前状态每个点与结束状态对应点之间的曼哈顿距离之和。

```

//对结束状态的每个点遍历
for (int row = 0; row < N; row++) {
    for (int col = 0; col < N; col++) {
        if (Main.end_state[row][col] != this.state[row][col]) {
            //如果和当前状态对应位置的点不一样,估价代价计算
            find:
            for (int temp_row = 0; temp_row < N; temp_row++) {
                for (int temp_col = 0; temp_col < N; temp_col++) {
                    if (this.state[temp_row][temp_col] == Main.end_state[row][col]) {
                        hcost += Math.abs(temp_col - col) + Math.abs(temp_row -
row));
                        break find;
                    }
                }
            }
        }
    }
}
}
}
}

```

2. 设置初始状态为 ，目标状态为 。解的过程为

第0次遍历	第1次遍历	第2次遍历	第3次遍历	第4次遍历
深度:0	深度:1	深度:2	深度:3	深度:4
预计cost:4	预计cost:4	预计cost:4	预计cost:2	预计cost:0
				
				
				

## 九、实验数据及结果分析：

关键代码说明：

node 类：此类代表了求解过程中的一个状态，这是包含的参数

```

static int N = 3; //问题大小
node parent;
int depth = 0;
int hcost = 0; //预计代价
int[][] state; //当前状态
int empty_row, empty_col; //空格所在位置

```

。初始化时传入 state 参数，自动计算预计代价。

`generateNode(String direction)`: 根据传入的方向, 生成空格向此方向走一步的孩子节点

`generateChildNode()`: 判断在当前状态下能生成的孩子节点并生成

Main 类: 首先建立 `open` 表和 `closed` 表, 构建初始状态。然后进入循环, 当 `open` 表为空时退出循环, 问题无解。循环中先从 `open` 表找到总花费最

```
node temp = open.get(0);
int min = open.get(0).getCost();
for (node i : open) {
    if (i.getCost() < min) {
        min = i.getCost();
        temp = i;
    }
}
```

小的节点

如果这个节点是最优解就输出

```
if (temp.hcost_ == 0) {
    //找到了最优解
    Stack<String> find_track = new Stack<>();
    find_track.push(temp.toString());
    while (temp.parent != null) {
        temp = temp.parent;
        find_track.push(temp.toString());
    }
    int i = 0;
    while (!find_track.empty()) {
        System.out.println("第" + i + "次遍历");
        i++;
        System.out.print(find_track.peek());
        find_track.pop();
    }
    break;
}
```

, 采用了一个栈存放从

解到初始状态的路径。

如果不是最优解, 就将节点移入 `closed` 表, 并且生成之后的节点

```

open.remove(index: 0);
closed.add(temp);
//生成接下来的节点
ArrayList<node> child = temp.generateChildNode();
//如果相同就不能加入
for (node i : child) {
    if (closed.contains(i)) {
        int pos = closed.indexOf(i);
        node pos_node = closed.get(pos);
        if (i.depth < pos_node.depth) {
            closed.remove(pos);
            closed.add(i);
        }
    } else
        open.add(i);
}

```

，要注意的是对于生

成的节点要有选择性的加入 open 表，如果在 closed 表中已经存在了，就不加入 open 表，而是修改 closed 表中对应项为花费最小的。

## 十、实验结论：

成功实现了用 A\*算法求解 8 数码难题。

## 十一、总结及心得体会：

深刻理解了 A\*算法。

## 十二、对本实验过程及方法、手段的改进建议：

无。

报告评分：

指导教师签字：

# 电 子 科 技 大 学

# 实 验 报 告

**实验地点：主楼 A2-412**

**一、实验室名称： 计算机学院实验中心**

**二、实验项目名称：决策树实验**

**三、实验学时：5 学时**

**四、实验原理：**

**(1) ID3 算法**

ID3 算法的核心思想就是以信息增益度量属性选择，选择分裂后信息增益最大的属性进行分裂。下面先定义几个要用到的概念。设 D 为用类别对训练元组进行的划分，则 D 的熵（entropy）表示为：

$$\inf o(D) = - \sum_{i=1}^m p_i \log_2(p_i)$$

其中  $p_i$  表示第  $i$  个类别在整个训练元组中出现的概率，可以用属于此类别元素的数量除以训练元组元素总数量作为估计。熵的实际意义表示是 D 中元组的类标号所需要的平均信息量。现在我们假设将训练元组 D 按属性 A 进行划分，则 A 对 D 划分的期望信息为：

$$\inf o_A(D) = \sum_{j=1}^v \frac{|D_j|}{|D|} \inf o(D_j)$$

而信息增益即为两者的差值：

$$gain(A) = \inf o(D) - \inf o_A(D)$$

ID3 算法就是在每次需要分裂时，计算每个属性的增益率，然后选择增益率最大的属性进行分裂。

对于特征属性为连续值，可以如此使用 ID3 算法：先将 D 中元素按照特征属性排序，则每两个相邻元素的中间点可以看做潜在分裂点，从第一个潜在分裂点开始，分裂 D 并计算两个集合的期望信息，具有最小期望信息的点称为这个属性的最佳分裂点，其信息期望作为此属性的信息期望。

**五、实验目的：**

编程实现决策树算法 ID3；理解算法原理。

## 六、实验内容：

利用 traindata.txt 的数据（75\*5，第 5 列为标签）进行训练，构造决策树；  
利用构造好的决策树对 testdata.txt 的数据进行分类，并输出分类准确率。

要求：

（1）提交源代码及可执行文件。

（2）提交实验报告，内容包括：对代码的简单说明、运行结果的截图及说明等。

（3）需画出决策树，指明每个分支所对应的特征/属性，以及分裂值。

注：如用到了剪枝、限定深度等技巧（加分项），请加以说明。

## 七、实验器材（设备、元器件）：

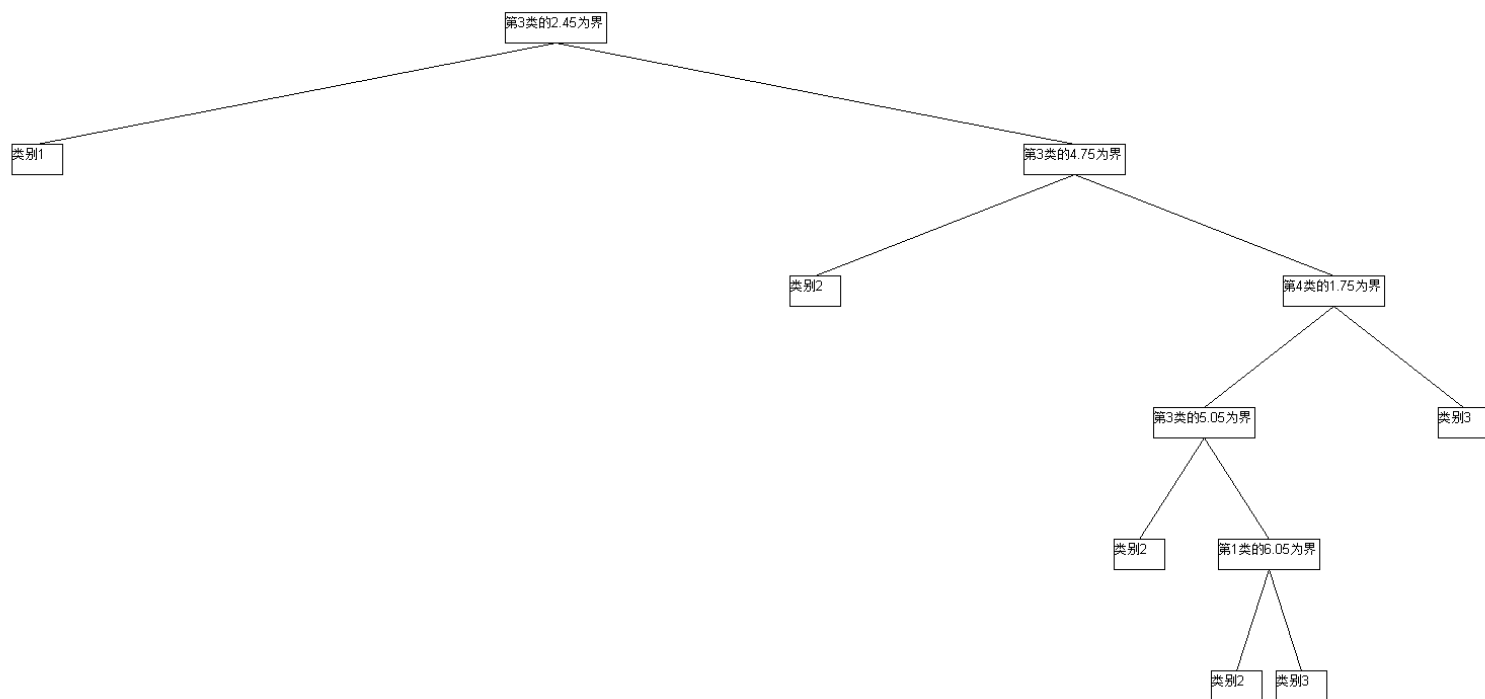
PC 微机一台

## 八、实验步骤：

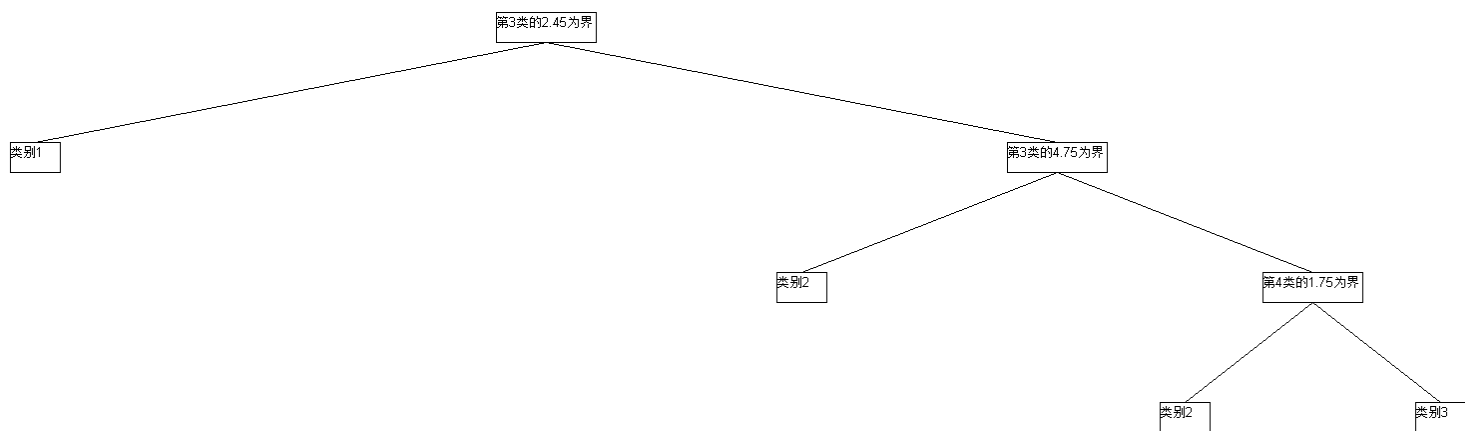


1.剪枝前的决策树（源图片：）





## 2. 剪枝后的决策树（源图片：剪枝后.jpg）



3.决策树在测试集上的准确率，剪枝前后的准确率没有区别，错误的节点也

```
错误数据：
种类:2
数据:[5.9, 3.2, 4.8, 1.8]
误分类为:3
错误数据：
种类:3
数据:[4.9, 2.5, 4.5, 1.7]
误分类为:2
错误数据：
种类:3
数据:[6.0, 2.2, 5.0, 1.5]
误分类为:2
准确率:96.0%
```

一样，这说明这是一个好的剪枝。

## 九、实验数据及结果分析：

剪枝方法：采用后剪枝的方法精简决策树。从后往前对每个非叶节点进行判断，对比保持原样和剪去此分支改为数量最多的类别，根据在测试集中的正确率来判断是否剪枝。根据奥卡姆剃刀原则，如果准确率没有下降就应该剪枝。

```

static public void post_pruning(node root) {
    //若此节点的孩子不是叶节点，就递归调用
    if (!root.child_left.is_leaf()) {
        post_pruning(root.child_left);
    }
    if (!root.child_right.is_leaf()) {...}
    //此节点的孩子都是叶节点
    if (root.child_left.is_leaf() && root.child_right.is_leaf()) {
        //统计错误情况
        int wrong_without_pruning = 0;
        int wrong_with_pruning = 0;
        ArrayList<data> after_pruning_test_data = new ArrayList<>();
        after_pruning_test_data.addAll(root.child_left.test_data);
        after_pruning_test_data.addAll(root.child_right.test_data);
        root.test_data = after_pruning_test_data;
        for (data test : root.child_left.test_data) {...}
        for (data test : root.child_right.test_data) {...}
        for (data test : after_pruning_test_data) {...}
        if (wrong_without_pruning >= wrong_with_pruning) {
            //剪枝
            System.out.println("剪枝了"+root.classify+root.flag);
            root.classify = -1;
            root.flag = 0;
            ArrayList<data> list = new ArrayList<>();
            list.addAll(Arrays.asList(root.child_left.train_data));
            list.addAll(Arrays.asList(root.child_right.train_data));
            data[] temp = new data[list.size()];
            list.toArray(temp);
            root.train_data = temp;
            root.child_left = null;
            root.child_right = null;
        }
    }
}
}

```

关键代码说明：

node 类： 此类代表决策树上的节点。参数为

```

node parent, child_left, child_right;
int classify = -1; //分类的种类
double flag; //分类的值
data[] train_data;
ArrayList<data> test_data = new ArrayList<>();
double entropy; //信息熵

```

在新建节点时自动计算信息熵

```
private double calculate_entropy() {
    int total = train_data.length;
    Map<Integer, Integer> count = get_count();
    double entropy = 0;
    for (int value : count.values()) {
        double p = (double) value / total;
        entropy -= p * (log(p) / log(2));
    }
    return entropy;
}
```

有方法 `get_max_type()` 获取节点中最大的种类。

`Data` 类：代表数据集中的一行数据

`draw_tree` 类：用于绘制决策树，与构建决策树相同的思想，从根节点开始递归调用绘制各个节点，最后将结果保存到文件。

```
public void draw_node(node root, int x, int y, int depth) {
    StringBuilder sb = new StringBuilder();
    if (root.classify == -1) {
        sb.append("类别").append(root.get_max_type());
        node_width = 50;
    } else {
        sb.append("第").append(root.classify + 1).append("类的").append(root.flag).append("为界");
        node_width = 100;
    }
    //绘制此节点
    g2.drawRect(x - node_width / 2, y, node_width, node_height);
    g2.drawString(sb.toString(), x - node_width / 2, y + node_height / 2);
    int start_point_x = x;
    int start_point_y = y + node_height;
    if (root.child_left != null) {
        offset_x = (int) (width / Math.pow(2, depth + 1));
        g2.drawLine(start_point_x, start_point_y, x2: x - offset_x, y2: y + node_height + offset_y);
        draw_node(root.child_left, x - offset_x, y + node_height + offset_y, depth + 1);
    }
    if (root.child_right != null) {...}
}
```

`Main` 类：因为特征值为连续的，因此要先构造分类点。对训练集的每个特征的每个值进行遍历，排序，去除重复值，就可以得到分类的方法。并且因为是连续值，同一种分类的方法可以重复使用。

```

//计算分类点
for (int i = 0; i < 4; i++) {
    ArrayList<Double> temp_classify_flag = new ArrayList<>();
    ArrayList<Double> temp = new ArrayList<>();
    for (data t : train_data)
        if (!temp.contains(t.classify[i]))
            temp.add(t.classify[i]);
    temp.sort(Comparator.naturalOrder());
    for (int t = 0; t < temp.size() - 1; t++) {
        temp_classify_flag.add((temp.get(t) + temp.get(t + 1)) / 2);
    }
    double[] toArray = new double[temp_classify_flag.size()];
    for (int j = 0; j < temp_classify_flag.size(); j++)
        toArray[j] = temp_classify_flag.get(j);
    classify_flag[i] = toArray;
}

```

之后手动构建一个根节点后开始构建决策树，并且测试，输出图片，再剪枝进行对比。

```

data[] train_data_array = new data[train_data.size()];
train_data.toArray(train_data_array);
//构建决策树
node root = new node(train_data_array);
root.parent = null;
create_tree(root);
new draw_tree(root, path: "剪枝前.jpg");

//测试决策树
data[] test_data_array = new data[test_data.size()];
test_data.toArray(test_data_array);
test_tree(root, test_data_array);

//剪枝
post_pruning(root);
new draw_tree(root, path: "剪枝后.jpg");
test_tree(root, test_data_array);

```

计算信息增益：对于一个父节点，根据一个已知的划分方式，可以生成两个孩子节点，然后运用信息增益的公式就可以求出。

```

static public double caculate_gain(node father_node, int classify, double flag) {
    node[] temp = generate_child(father_node, classify, flag);
    int total = father_node.train_data.length;
    double gain = father_node.entropy;
    for (node i : temp) {
        gain -= ((double) i.train_data.length / total) * i.entropy;
    }
    return gain;
}

```

生成孩子节点：父节点的 train\_data 包含了此节点下的所有训练数据，因此对这些数据遍历可以很轻松构建 2 个孩子节点。

```

for (data i : father_node.train_data) {
    if (i.classify[classify] <= flag)
        left_data.add(i);
    else
        right_data.add(i);
}

```

构建决策树：如果节点的数据都是同一种类，就不需要继续划分。不然对于每种分类方式进行遍历，寻找信息增益最大的一种。然后对以此构建的孩子节点继续构建决策树。

```

static public void create_tree(node root) {
    double max_gain = -99999;
    int max_classify = -1;
    double max_flag = -1;
    if (root.is_one_type())
        return;
    //找到信息增益最大的分类方式
    for (int i = 0; i < 4; i++) {
        for (double flag : classify_flag[i]) {
            double gain = caculate_gain(root, i, flag);
            if (gain > max_gain) {
                max_gain = gain;
                max_classify = i;
                max_flag = flag;
            }
        }
    }
    node[] temp = generate_child(root, max_classify, max_flag);
    create_tree(temp[0]);
    create_tree(temp[1]);
}

```

测试决策树：对于测试集中的每一个数据，走一遍决策树既可知道是否正确。同时将测试数据保存到节点中，为剪枝做准备。

```

int wrong = 0, right = 0;
for (data test : test_data) {
    node state = root;
    while (state.classify != -1) {
        if (test.classify[state.classify] < state.flag)
            state = state.child_left;
        else
            state = state.child_right;
    }
    state.test_data.add(test);
    if (state.get_max_type() != test.type) {
        wrong++;
        System.out.println("错误数据:\n" + test + "误分类为:" + state.get_max_type());
    } else
        right++;
}
}

```

## 十、实验结论：

完成了 ID3 算法构建决策树，并且应用了后剪枝的方法将决策树精简，且正确率没有下降。

**十一、总结及心得体会：**

对于 ID3 算法有了更加深刻的认识。

**十二、对本实验过程及方法、手段的改进建议：**

无。

**报告评分：**

**指导教师签字：**



# 电子科技大学

## 实验报告

二、实验项目名称：强化学习实验

三、实验学时：6 学时

四、实验原理：

### (1) Q-Learning 算法

Q-Learning 是一种记录行为值 (Q value) 的方法，每种行为在一定的状态都会有一个值  $Q(s, a)$ ，就是说行为  $a$  在  $s$  状态的值是  $Q(s, a)$ 。对于迷宫游戏， $s$  就是当前 agent 所在的地点了。每一步，智能体可以选择四种动作，所以动作  $a$  有四种可能性。

在本实验里，已经提供了强化学习基本的训练接口，只需要实现 Q 表格的强化学习方法即可。算法框架如下：

```
Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
  Initialize  $s$ 
  Repeat (for each step of episode):
    Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Take action  $a$ , observe  $r, s'$ 
     $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
     $s \leftarrow s'$ 
  until  $s$  is terminal
```

图 1 强化学习算法框架

可以看到，算法的核心部分是更新 Q 表格。训练目标是在评价阶段，智能体的平均奖励达到最高。

## (2) 迷宫游戏说明

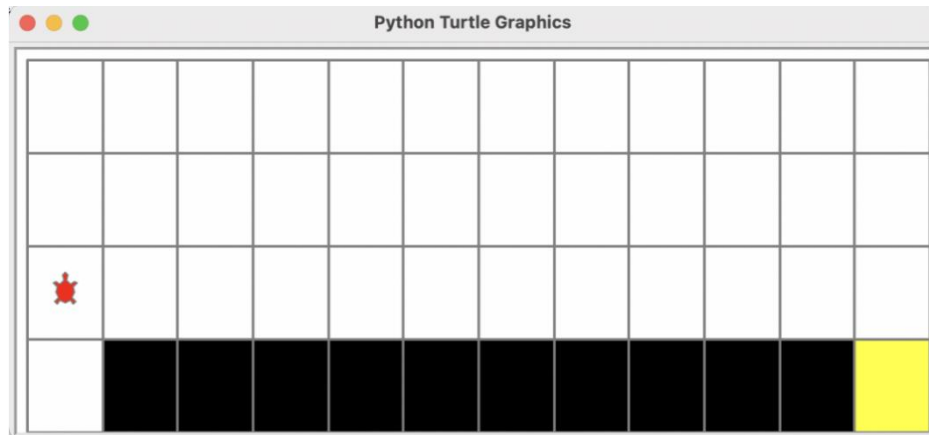


图 2 迷宫游戏示例

如上图所示，本实验所研究的迷宫由一个二维表格构成。其中白色区域是可行走区域，黑色区域是陷阱区域，黄色区域是终点。规则如下：

- 1) 智能体从左下角出发，到达黄色区域即游戏成功。
- 2) 智能体每次可选择上、下、左、右四种移动动作，每次动作得到-1 奖励。
- 3) 智能体不能移动出网络，如果下一步的动作命令会让智能体移动出边界，那么这一步将不会执行，即智能体原地不动，得到-1 奖励。
- 4) 智能体移动到黑色区域，得到-100 奖励。
- 5) 智能体移动到黄色区域，该回合结束。

由图可知，最优的路线需要 13 步，所以最后智能体获得的奖励指在-13 左右为最佳结果。

## 五、实验目的：

使用 Q 表格方法解决迷宫寻路问题，理解强化学习算法原理。

## 六、实验内容：

结合给定的程序代码框架，补全代码，并达到实验目的。鼓励探索新的迷宫布局 and 任务。

要求:

- (1) 提交源代码及可执行文件。
- (2) 提交实验报告, 内容包括: 对代码的简单说明、运行结果的截图及说明等。

## 七、实验器材 (设备、元器件):

PC 微机一台

## 八、实验步骤:

1. 训练数据, 可以明显看到随着轮次的增加 reward 逐渐变大:

```
Start to train !
Env:CliffWalking-v0, Algorithm:
Episode:1/200: reward:-7063.0
Episode:2/200: reward:-2843.0
Episode:3/200: reward:-396.0
Episode:4/200: reward:-1088.0
Episode:5/200: reward:-76.0
Episode:6/200: reward:-429.0
Episode:7/200: reward:-437.0
Episode:8/200: reward:-481.0
Episode:9/200: reward:-452.0
Episode:10/200: reward:-336.0
Episode:11/200: reward:-157.0
Episode:12/200: reward:-769.0
Episode:13/200: reward:-247.0
Episode:14/200: reward:-252.0
Episode:15/200: reward:-432.0
Episode:185/200: reward:-20.0
Episode:186/200: reward:-15.0
Episode:187/200: reward:-24.0
Episode:188/200: reward:-129.0
Episode:189/200: reward:-23.0
Episode:190/200: reward:-17.0
Episode:191/200: reward:-360.0
Episode:192/200: reward:-231.0
Episode:193/200: reward:-13.0
Episode:194/200: reward:-13.0
Episode:195/200: reward:-27.0
Episode:196/200: reward:-353.0
Episode:197/200: reward:-14.0
Episode:198/200: reward:-25.0
Episode:199/200: reward:-21.0
Episode:200/200: reward:-26.0
Complete training!
```

2. 测试数据 reward 都为-13, 是最优解 (有时会出现卡在两个节点之间的情况, 说明经过那两个节点的次数太少, 通过增加训练轮次既可解决):

```
Start to eval !
Env:CliffWalking-v0, Algorithm:
Episode:1/30, reward:-13.0
Episode:2/30, reward:-13.0
Episode:3/30, reward:-13.0
Episode:4/30, reward:-13.0
Episode:5/30, reward:-13.0
Episode:26/30, reward:-13.0
Episode:27/30, reward:-13.0
Episode:28/30, reward:-13.0
Episode:29/30, reward:-13.0
Episode:30/30, reward:-13.0
Complete evaling!
results saved!
```

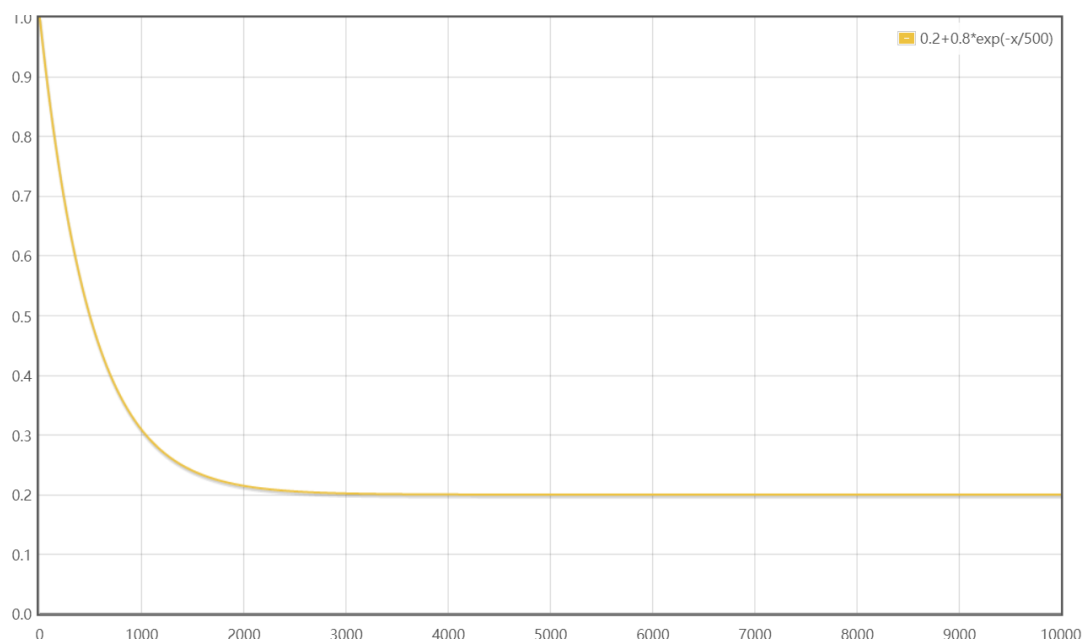
## 九、实验数据及结果分析:

关键代码说明:

predict 函数根据现在的状态输出最优解，如果 Q\_table 的值一样就随机选一个

```
def predict(self, state):
    max_num = np.max(self.Q_table[state])
    action = np.where(self.Q_table[state] == max_num)[0]
    action = np.random.choice(action)
    return action
```

choose\_action 函数采用了  $\epsilon$ -greedy 方法，并且随着训练步数的增多将  $\epsilon$  值减少，公式为  $\epsilon = 0.2 + 0.8 \times e^{-\frac{x}{500}}$ ，图像为



每次产生状态都有  $\epsilon$  的概率随机选择探索一个方向，并且越来越趋向于选择已知的最优解

```
def choose_action(self, state):
    self.sample_count += 1
    self.epsilon = 0.2 + 0.8 * math.exp(-1.0 * self.sample_count / 500)
    if np.random.uniform(0, 1) > self.epsilon:
        action = self.predict(state)
    else:
        action = np.random.choice(self.action_dim) # 随机探索选取一个动作
    return action
```

update 函数用来更新 Q\_table 的值，如果已经完成了就只用更新为此次的 reward，否则需要搭配衰减系数和学习率平衡当前和历史的 reward。

```
def update(self, state, action, reward, next_state, done):
    self.Q_table[state, action] += self.lr * (
        reward + self.gamma * np.max(self.Q_table[next_state]) - self.Q_table[state, action])
    if done:
        self.Q_table[state, action] = reward
```

## 十、实验结论：

成功实现了用 Q\_learning 算法进行强化学习，解决了迷宫寻路问题。

## 十一、总结及心得体会：

深刻理解了 Q\_learning 算法已经强化学习的思路。

## 十二、对本实验过程及方法、手段的改进建议：

对于  $\epsilon$  值的更新并没有知道理论上的指导，只是自己随意写了一个单调递减函数，并且对于参数的确定是试出来的。如果有理论依据将会更好。

报告评分：

指导教师签字：

# 附件

## 实验一代码：

### Main.java

```
1. import java.util.ArrayList;
2. import java.util.Stack;
3.
4. public class Main {
5.     static final int[][] start_state = {{2, 8, 3}, {1, 0, 4}, {7, 6,
6.     5}};
7.     static final int[][] end_state = {{1, 2, 3}, {8, 0, 4}, {7, 6, 5}
8.     };
9.
10.    public static void main(String[] args) {
11.        ArrayList<node> open = new ArrayList<>();
12.        ArrayList<node> closed = new ArrayList<>();
13.        node start_node = new node(start_state);
14.        start_node.parent = null;
15.        open.add(start_node);
16.        while (open.size() > 0) {
17.            //sort 太慢了
18.            //open.sort(Comparator.naturalOrder());
19.            node temp = open.get(0);
20.            int min = open.get(0).getCost();
21.            for (node i : open) {
```

```

20.         if (i.getCost() < min) {
21.             min = i.getCost();
22.             temp = i;
23.         }
24.     }
25. //         System.out.println("深度"+temp.depth);
26. //         System.out.println("hcost"+temp.hcost);
27.     if (temp.hcost == 0) {
28.         //找到了最优解
29.         Stack<String> find_track = new Stack<>();
30.         find_track.push(temp.toString());
31.         while (temp.parent != null) {
32.             temp = temp.parent;
33.             find_track.push(temp.toString());
34.         }
35.         int i = 0;
36.         while (!find_track.empty()) {
37.             System.out.println("第" + i + "次遍历");
38.             i++;
39.             System.out.print(find_track.peek());
40.             find_track.pop();
41.         }
42.         break;
43.     }
44.     open.remove(0);
45.     closed.add(temp);
46.     //生成接下来的节点
47.     ArrayList<node> child = temp.generateChildNode();
48.     //如果相同就不能加入
49.     for (node i : child) {
50.         if (closed.contains(i)) {
51.             int pos = closed.indexOf(i);
52.             node pos_node = closed.get(pos);
53.             if (i.depth < pos_node.depth) {
54.                 closed.remove(pos);
55.                 closed.add(i);
56.             }
57.         } else
58.             open.add(i);
59.     }
60. }
61. if (open.size() == 0) {
62.     System.out.println("无解");
63.     System.out.println(closed.size());

```

```

64.     }
65.
66.     }
67. }

```

## Node.java

```

1. import java.util.ArrayList;
2. import java.util.Arrays;
3.
4.
5. public class node implements Comparable<node> {
6.     static int N = 3; //问题大小
7.     node parent;
8.     int depth = 0;
9.     int hcost = 0; //预计代价
10.    int[][] state; //当前状态
11.    int empty_row, empty_col; //空格所在位置
12.
13.    public node(int[][] state) {
14.        this.state = state;
15.        for (int row = 0; row < N; row++) {
16.            for (int col = 0; col < N; col++) {
17.                if (state[row][col] == 0) {
18.                    empty_row = row;
19.                    empty_col = col;
20.                }
21.                if (Main.end_state[row][col] != this.state[row][col])
22.                {
23.                    //寻找不同的节点在哪,估价代价计算
24.                    find:
25.                    for (int temp_row = 0; temp_row < N; temp_row++)
26.                    {
27.                        for (int temp_col = 0; temp_col < N; temp_col
28.                        ++){
29.                            if (this.state[temp_row][temp_col] == Mai
30.                            n.end_state[row][col]) {
31.                                hcost += (Math.abs(temp_col - col) +
32.                                Math.abs(temp_row - row));
33.                                break find;
34.                            }
35.                        }
36.                    }
37.                }
38.            }
39.        }
40.    }
41. }

```

```

34.     }
35. }
36.
37. private node generateNode(String direction) {
38.     int swap_row = empty_row, swap_col = empty_col;
39.     switch (direction) {
40.         case "left" -> swap_col--;
41.         case "right" -> swap_col++;
42.         case "up" -> swap_row--;
43.         case "down" -> swap_row++;
44.     }
45.     //要用深拷贝
46.     int[][] temp_state = new int[N][N];
47.     for (int i = 0; i < N; i++) {
48.         System.arraycopy(state[i], 0, temp_state[i], 0, N);
49.     }
50.     int temp_swap = temp_state[swap_row][swap_col];
51.     temp_state[swap_row][swap_col] = 0;
52.     temp_state[empty_row][empty_col] = temp_swap;
53.     node temp = new node(temp_state);
54.     temp.depth = this.depth + 1;
55.     temp.parent = this;
56.     return temp;
57. }
58.
59. public ArrayList<node> generateChildNode() {
60.     if (this.hcost == 0) {
61.         return null;
62.     }
63.     ArrayList<node> result = new ArrayList<>();
64.     //left
65.     if (empty_col > 0) {
66.         result.add(generateNode("left"));
67.     }
68.     //right
69.     if (empty_col < N - 1) {
70.         result.add(generateNode("right"));
71.     }
72.     //up
73.     if (empty_row > 0) {
74.         result.add(generateNode("up"));
75.     }
76.     //down
77.     if (empty_row < N - 1) {

```



```

78.         result.add(generateNode("down"));
79.     }
80.     return result;
81. }
82.
83. public int getCost() {
84.     return depth + hcost;
85. }
86.
87. @Override
88. public int compareTo(Node o) {
89.     return Integer.compare(this.getCost(), o.getCost());
90. }
91. @Override
92. public String toString() {
93.     StringBuilder temp = new StringBuilder();
94.     temp.append("深度:").append(depth).append("\n");
95.     temp.append("预计 cost:").append(hcost).append("\n");
96.     for(int[] i:state){
97.         temp.append(Arrays.toString(i));
98.         temp.append("\n");
99.     }
100.    return temp.toString();
101. }
102. @Override
103. public boolean equals(Object obj) {
104.     if(obj instanceof Node){
105.         return Arrays.deepEquals(((Node) obj).state,this.state
106.     );
107.     }
108.     return obj.equals(this);
109. }

```

实验二代码：

## Main.java

```

1. import java.io.IOException;
2. import java.nio.file.Files;
3. import java.nio.file.Paths;
4. import java.util.ArrayList;
5. import java.util.Arrays;
6. import java.util.Comparator;
7. import java.util.List;

```

```

8.
9.  public class Main {
10.     static ArrayList<node> tree = new ArrayList<>();
11.     static double[][] classify_flag = new double[4][];
12.
13.     public static void main(String[] args) {
14.         ArrayList<data> train_data = new ArrayList<>();
15.         ArrayList<data> test_data = new ArrayList<>();
16.         //读取文件
17.         try {
18.             List<String> train_lines = Files.readAllLines(Paths.get("
traindata.txt"));
19.             List<String> test_lines = Files.readAllLines(Paths.get("t
estdata.txt"));
20.             for (String e : train_lines) {
21.                 train_data.add(new data(e));
22.             }
23.             for (String e : test_lines) {
24.                 test_data.add(new data(e));
25.             }
26.         } catch (IOException e) {
27.             e.printStackTrace();
28.         }
29.         //计算分类点
30.         for (int i = 0; i < 4; i++) {
31.             ArrayList<Double> temp_classify_flag = new ArrayList<>();
32.
33.             ArrayList<Double> temp = new ArrayList<>();
34.             for (data t : train_data)
35.                 if (!temp.contains(t.classify[i]))
36.                     temp.add(t.classify[i]);
37.             temp.sort(Comparator.naturalOrder());
38.             for (int t = 0; t < temp.size() - 1; t++) {
39.                 temp_classify_flag.add((temp.get(t) + temp.get(t + 1)
) / 2);
40.             }
41.             double[] toArray = new double[temp_classify_flag.size()];
42.
43.             for (int j = 0; j < temp_classify_flag.size(); j++)
44.                 toArray[j] = temp_classify_flag.get(j);
45.             classify_flag[i] = toArray;
46.         }
47.
48.         data[] train_data_array = new data[train_data.size()];

```

```
47.         train_data.toArray(train_data_array);
48.         //构建决策树
49.         node root = new node(train_data_array);
50.         root.parent = null;
51.         create_tree(root);
52.         new draw_tree(root, "剪枝前.jpg");
53.
54.         //测试决策树
55.         data[] test_data_array = new data[test_data.size()];
56.         test_data.toArray(test_data_array);
57.         test_tree(root, test_data_array);
58.
59.         //剪枝
60.         post_pruning(root);
61.         new draw_tree(root, "剪枝后.jpg");
62.         test_tree(root, test_data_array);
63.
64.     }
65.
66.     static public node[] generate_child(node father_node, int classify
        y, double flag) {
67.         node[] result = new node[2];
68.         father_node.flag = flag;
69.         father_node.classify = classify;
70.         ArrayList<data> left_data = new ArrayList<>();
71.         ArrayList<data> right_data = new ArrayList<>();
72.         for (data i : father_node.train_data) {
73.             if (i.classify[classify] <= flag)
74.                 left_data.add(i);
75.             else
76.                 right_data.add(i);
77.         }
78.         data[] left_data_array = new data[left_data.size()];
79.         data[] right_data_array = new data[right_data.size()];
80.         left_data.toArray(left_data_array);
81.         right_data.toArray(right_data_array);
82.         node left_node = new node(left_data_array);
83.         node right_node = new node(right_data_array);
84.         left_node.parent = father_node;
85.         right_node.parent = father_node;
86.         father_node.child_left = left_node;
87.         father_node.child_right = right_node;
88.         result[0] = left_node;
89.         result[1] = right_node;
```

```

90.         return result;
91.     }
92.
93.     static public double caculate_gain(node father_node, int classify
, double flag) {
94.         node[] temp = generate_child(father_node, classify, flag);
95.         int total = father_node.train_data.length;
96.         double gain = father_node.entropy;
97.         for (node i : temp) {
98.             gain -
= ((double) i.train_data.length / total) * i.entropy;
99.         }
100.        return gain;
101.    }
102.
103.    static public void create_tree(node root) {
104.        double max_gain = -99999;
105.        int max_classify = -1;
106.        double max_flag = -1;
107.        if (root.is_one_type())
108.            return;
109.        //找到信息增益最大的分类方式
110.        for (int i = 0; i < 4; i++) {
111.            for (double flag : classify_flag[i]) {
112.                double gain = caculate_gain(root, i, flag);
113.                if (gain > max_gain) {
114.                    max_gain = gain;
115.                    max_classify = i;
116.                    max_flag = flag;
117.                }
118.            }
119.        }
120.        node[] temp = generate_child(root, max_classify, max_flag)
;
121.        create_tree(temp[0]);
122.        create_tree(temp[1]);
123.    }
124.
125.    static public void post_pruning(node root) {
126.        //若此节点的孩子不是叶节点，就递归调用
127.        if (!root.child_left.is_leaf()) {
128.            post_pruning(root.child_left);
129.        }
130.        if (!root.child_right.is_leaf()) {

```

```

131.         post_pruning(root.child_right);
132.     }
133.     //此节点的孩子都是叶节点
134.     if (root.child_left.is_leaf() && root.child_right.is_leaf(
        )) {
135.         //统计错误情况
136.         int wrong_without_pruning = 0;
137.         int wrong_with_pruning = 0;
138.         ArrayList<data> after_pruning_test_data = new ArrayLis
            t<>();
139.         after_pruning_test_data.addAll(root.child_left.test_da
                ta);
140.         after_pruning_test_data.addAll(root.child_right.test_d
                ata);
141.         root.test_data = after_pruning_test_data;
142.         for (data test : root.child_left.test_data) {
143.             if (test.type != root.child_left.get_max_type())
144.                 wrong_without_pruning++;
145.         }
146.         for (data test : root.child_right.test_data) {
147.             if (test.type != root.child_right.get_max_type())

148.                 wrong_without_pruning++;
149.         }
150.         for (data test : after_pruning_test_data) {
151.             if (test.type != root.get_max_type())
152.                 wrong_with_pruning++;
153.         }
154.         if (wrong_without_pruning >= wrong_with_pruning) {
155.             //剪枝
156.             // System.out.println("剪枝了
                "+root.classify+root.flag);
157.             root.classify = -1;
158.             root.flag = 0;
159.             ArrayList<data> list = new ArrayList<>();
160.             list.addAll(Arrays.asList(root.child_left.train_da
                ta));
161.             list.addAll(Arrays.asList(root.child_right.train_d
                ata));
162.             data[] temp = new data[list.size()];
163.             list.toArray(temp);
164.             root.train_data = temp;
165.             root.child_left = null;
166.             root.child_right = null;

```

```

167.         }
168.     }
169. }
170.
171.     static public void test_tree(node root, data[] test_data) {
172.         int wrong = 0, right = 0;
173.         for (data test : test_data) {
174.             node state = root;
175.             while (state.classify != -1) {
176.                 if (test.classify[state.classify] < state.flag)
177.                     state = state.child_left;
178.                 else
179.                     state = state.child_right;
180.             }
181.             state.test_data.add(test);
182.             if (state.get_max_type() != test.type) {
183.                 wrong++;
184.                 System.out.println("错误数据:\n" + test + "误分类
为:" + state.get_max_type());
185.             } else
186.                 right++;
187.         }
188.         System.out.println("准确
率:" + (double) right / (right + wrong) * 100 + "%");
189.     }
190. }

```

## Node.java

```

1. import java.util.ArrayList;
2. import java.util.HashMap;
3. import java.util.Map;
4.
5. import static java.lang.Math.log;
6.
7.
8. public class node {
9.     node parent, child_left, child_right;
10.    int classify = -1; //分类的种类
11.    double flag; //分类的值
12.    data[] train_data;
13.    ArrayList<data> test_data = new ArrayList<>();
14.    double entropy; //信息熵
15.
16.    public node(data[] train_data) {

```

```

17.         this.train_data = train_data;
18.         this.entropy = calculate_entropy();
19.     }
20.
21.     public boolean is_leaf() {
22.         return child_left == null && child_right == null;
23.     }
24.
25.     private double calculate_entropy() {
26.         int total = train_data.length;
27.         Map<Integer, Integer> count = get_count();
28.         double entropy = 0;
29.         for (int value : count.values()) {
30.             double p = (double) value / total;
31.             entropy -= p * (log(p) / log(2));
32.         }
33.         return entropy;
34.     }
35.
36.     public boolean is_one_type() {
37.         int classify = train_data[0].type;
38.         for (data t : train_data) {
39.             if (t.type != classify)
40.                 return false;
41.         }
42.         return true;
43.     }
44.     //获取节点中最多的种类
45.     public int get_max_type() {
46.         int total = train_data.length;
47.         int max_count = -1;
48.         int max_type = -1;
49.         Map<Integer, Integer> count = get_count();
50.
51.         for (Integer key : count.keySet()) {
52.             if (count.get(key) >= total / 2)
53.                 return key;
54.             if (max_count < count.get(key)) {
55.                 max_count = count.get(key);
56.                 max_type = key;
57.             }
58.         }
59.         return max_type;
60.     }

```

```

61.
62.
63.    //种类-数量映射
64.    private Map<Integer, Integer> get_count() {
65.        Map<Integer, Integer> count = new HashMap<>();
66.        for (data i : train_data) {
67.            if (count.containsKey(i.type)) {
68.                int temp = count.get(i.type);
69.                count.put(i.type, ++temp);
70.            } else {
71.                count.put(i.type, 1);
72.            }
73.        }
74.        return count;
75.    }
76.
77.
78. }

```

## Draw\_tree.java

```

1. import javax.imageio.ImageIO;
2. import java.awt.*;
3. import java.awt.image.BufferedImage;
4. import java.io.File;
5. import java.io.IOException;
6.
7. public class draw_tree {
8.     int width = 2048;
9.     int node_width = 50;
10.    int node_height = 30;
11.    int offset_x = 150, offset_y = 100;
12.    node root;
13.    Graphics2D g2;
14.
15.    public draw_tree(node root, String path) {
16.        this.root = root;
17.        final BufferedImage image = new BufferedImage(2048, 1024, BufferedImage.TYPE_INT_ARGB);
18.        g2 = image.createGraphics();
19.        g2.setFont(new Font(null, Font.PLAIN, 13));
20.        g2.setPaint(Color.white);
21.        g2.fillRect(0, 0, 2048, 1024);
22.        g2.setPaint(Color.black);
23.        draw_node(root, width / 2, 10, 1);

```



```

24.         g2.dispose();
25.
26.         try {
27.             ImageIO.write(image, "png", new File(path));
28.         } catch (IOException e) {
29.             e.printStackTrace();
30.         }
31.     }
32.
33.     public void draw_node(node root, int x, int y, int depth) {
34.         StringBuilder sb = new StringBuilder();
35.         if (root.classify == -1) {
36.             sb.append("类别").append(root.get_max_type());
37.             node_width = 50;
38.         } else {
39.             sb.append("第").append(root.classify + 1).append("类的")
40.             sb.append(root.flag).append("为界");
41.             node_width = 100;
42.         }
43.         //绘制此节点
44.         g2.drawRect(x - node_width / 2, y, node_width, node_height);
45.
46.         g2.drawString(sb.toString(), x - node_width / 2, y + node_height / 2);
47.
48.         int start_point_x = x;
49.         int start_point_y = y + node_height;
50.         if (root.child_left != null) {
51.             offset_x = (int) (width / Math.pow(2, depth + 1));
52.             g2.drawLine(start_point_x, start_point_y, x - offset_x, y
53.             + node_height + offset_y);
54.             draw_node(root.child_left, x - offset_x, y + node_height
55.             + offset_y, depth + 1);
56.         }
57.         if (root.child_right != null) {
58.             offset_x = (int) (width / Math.pow(2, depth + 1));
59.             g2.drawLine(start_point_x, start_point_y, x + offset_x, y
60.             + node_height + offset_y);
61.             draw_node(root.child_right, x + offset_x, y + node_height
62.             + offset_y, depth + 1);
63.         }
64.     }
65. }

```

**Data.java**

```

1. import java.util.Arrays;
2.
3. public class data {
4.     int type;//种类
5.     double[] classify;//特征
6.
7.     public data(String s) {
8.         String[] temp = s.split("\t");
9.         classify = new double[4];
10.        for (int i = 0; i < 4; i++) {
11.            classify[i] = Double.parseDouble(temp[i]);
12.        }
13.        type = Integer.parseInt(temp[4]);
14.    }
15.
16.    @Override
17.    public String toString() {
18.        return "种类:" + type + "\n数
据:" + Arrays.toString(classify) + "\n";
19.    }
20. }

```

### 实验三代码:

```

1. def choose_action(self, state):
2.     self.sample_count += 1
3.     self.epsilon = 0.2 + 0.8 * math.exp(-
4.         1.0 * self.sample_count / 500)
5.     if np.random.uniform(0, 1) > self.epsilon:
6.         action = self.predict(state)
7.     else:
8.         action = np.random.choice(self.action_dim) # 随机探索选取一个
          动作
9.     return action
10.
11. def update(self, state, action, reward, next_state, done):
12.     self.Q_table[state, action] += self.lr * (
13.         reward + self.gamma * np.max(self.Q_table[next_state]) -
14.         self.Q_table[state, action])
15.     if done:
16.         self.Q_table[state, action] = reward
17.
18. def predict(self, state):
19.     max_num = np.max(self.Q_table[state])
20.     action = np.where(self.Q_table[state] == max_num)[0]

```

```
19.     action = np.random.choice(action)
20.     return action
```