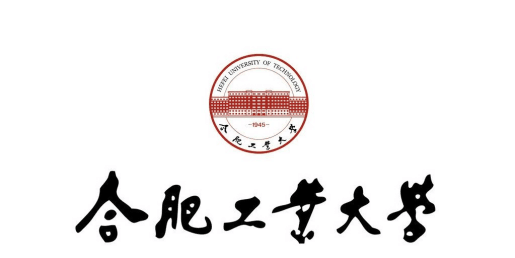
**数据结构课程设计**



|  |
| --- |
| **设计题目： 16 随机决策树构造** |
| **学生姓名： 马哲超** |
| **专 业： 计算机科学与技术** |
| **班 级： 计科21-3班** |
| **学 号： 2021212683** |
| **指导老师： 郭丹** |
| **完成日期： 2023年6月11日** |

**合肥工业大学计算机与信息学院**

# （一）需求和规格说明

**问题描述**

这是一个可以进行决策树生成与测试数据预测的程序



该程序可以利用读取的训练数据进行决策树的生成与可视化，并可以读取未分类的测试数据，并对测试数据进行分类显示在右侧表格中，如果遇到无法分类的测试数据则会显示类别为空。

该程序还可以保存生成的决策树为json格式，且可以读取已生成的决策树并利用生成的决策树对数据进行分类。

**编程任务：**

1. 利用Qt实现GUI可视化
2. 建立决策树数据结构
3. 用C++实现ID3算法
4. 利用ID3算法，对读取的数据进行决策树的生成，并进行可视化
5. 对读取的测试数据进行分类算法，并可视化分类结果
6. 将生成的决策树保存为json格式，并能从json格式读取保存的决策树

# （二）设计

**1.设计思想**

决策树算法较为简单，难点主要是决策树的可视化

可视化的时候，我利用了继承于QGraphicsItem的自建nodeItem类。需要注意的是，如果每次递归调用drawNode的时候再创建nodeItem，会导致程序死循环，所以需要在一开始先遍历树，并生成所有需要的nodeItem，这样就不会递归生成nodeItem导致死循环。但是由于类的特性，刷新的时候会重新画图，这是缺点之一。

为了美观，我加入了树的深度的判断，来使得深度不同的数节点之间的距离不同，但实现过程较为坎坷。为了测试程序的可泛化性，我利用了课设指导书中的例子以及鸢尾花数据集（iris）进行训练和测试。在利用鸢尾花数据集测试的时候发现，由于他每个特征的取值比较多，且测试数据中不一定存在所有的取值可能性，如果算法不更新的话，需要在测试的时候加入判断，只输出能判断出来的测试数据。且鸢尾花的树比较大，为了全面观察，我还实现了树的拖动事件，同样，实现过程较为困难。

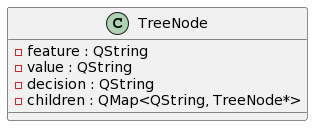
为了保证输入的文件是正确的csv格式，我进行了读取文件时的设置，保证程序的鲁棒性与实用性。

为了将决策树写入文件，我利用json进行保存，同理，利用json进行读取。

这里，鸢尾花数据集的学习可以发现我的决策树有很大的问题，因为他不能找到不同的类之间的联系，即不能进行多数值的决策，虽然可以进行测试数据的判断，但仍需要后续改进。

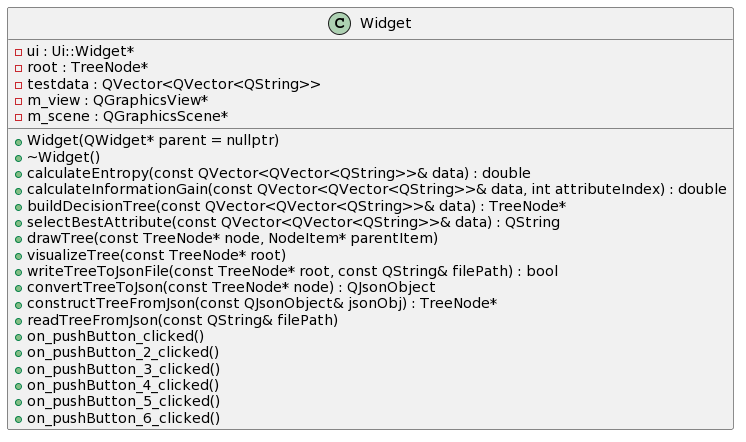
**2.设计表示**

（1）决策树节点结构：



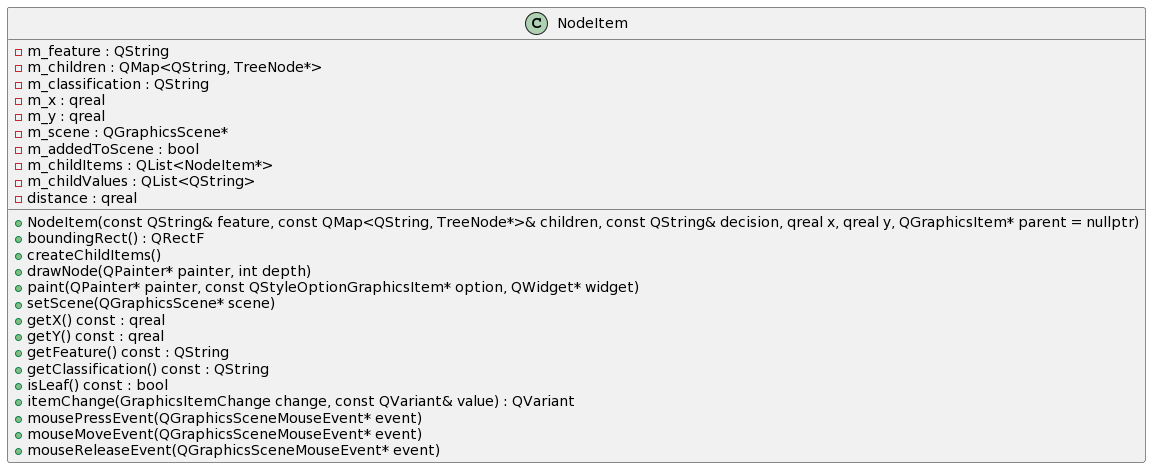
|  |  |  |  |
| --- | --- | --- | --- |
| 结构体名 | 成员数据类型 | 成员名 | 描述 |
| TreeNode | QString | feature | 内部节点对应的分类特征 |
| QString | value | 进行调试的时候使用 |
| QString | decision | 叶子节点对应的分类类别 |
| QMap<QString,TreeNode\*> | children | 特征不同选择对应的子节点 |

（2）主界面类



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类名 | 成员类别 | 成员数据类型 | 成员名 | 描述 |
| Widget | 数据 | UI::Widget\* | ui | 用于访问 UI 文件生成的对象和组件。 |
| TreeNode\* | root | 决策树的根节点 |
| QVector<QVector<QString> | testdata | 存储测试数据的二维向量 |
| QGraphicsView\* | m\_view | 用于绘制和显示决策树的可视化结果。 |
| QGraphicsScene\* | m\_scene | 用于绘制和显示决策树的可视化结果。 |
| 函数 | void | Widget(QWidget \*parent = nullptr) | 构造函数，用于创建 Widget 类的对象 |
| void | ~Widget() | 析构函数，用于销毁 Widget 类的对象。 |
| double | calculateEntropy(const QVector<QVector<QString>>& data) | 计算给定数据集的熵。 |
| double | calculateInformationGain(const QVector<QVector<QString>>& data, int attributeIndex) | 计算给定数据集在指定属性上的信息增益 |
| TreeNode\* | buildDecisionTree(const QVector<QVector<QString>>& data) | 构建决策树，返回根节点。 |
| QString | selectBestAttribute(const QVector<QVector<QString>>& data) | 选择数据集中最佳的属性作为节点分裂的依据。 |
| void | drawTree(const TreeNode\* node, NodeItem\* parentItem) | 根据决策树的节点信息绘制可视化的树形结构。 |
| void | visualizeTree(const TreeNode\* root) | 可视化整棵决策树。 |
| void | writeTreeToJsonFile(const TreeNode\* root, const QString& filePath) | 将决策树以 JSON 格式写入文件。 |
| QjsonObject | convertTreeToJson(const TreeNode\* node) | 将决策树转换为 JSON 对象 |
| void | readTreeFromJson(const QString& filePath) | 从文件中读取决策树 |
| void | on\_pushButton\_clicked() | 读取测试文件并执行决策树的构建 |
| void | on\_pushButton\_2\_clicked() | 可视化树 |
| void | on\_pushButton\_3\_clicked() | 读取测试文件 |
| void | on\_pushButton\_4\_clicked() | 对测试文件进行判断 |
| void | on\_pushButton\_5\_clicked() | 保存树为json格式 |
| void | on\_pushButton\_6\_clicked() | 读取json的树并加载到类的root中 |

（3）可视化画图类



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类名 | 成员类别 | 成员数据类型 | 成员名 | 描述 |
| NodeItem | 数据 | QString | m\_feature | 节点的属性。 |
| QMap<QString,TreeNode\*> | m\_children | 节点的子节点映射表，使用特征值作为键，对应的子节点 TreeNode 对象作为值。 |
| QString | m\_classification | 叶子节点的决策结果。 |
| qreal | m\_x | 节点的坐标。 |
| qreal | m\_y | 节点的坐标。 |
| QGraphicsScene\* | m\_scene | 节点所属的图形场景。 |
| bool | m\_addedToScene | 标志变量，表示节点是否已经添加到场景中 |
| QList<NodeItem\*> | m\_childItems | 子节点的列表，存储所有子节点的 NodeItem 对象 |
| Qlist<QString> | m\_childValues | 子节点的特征取值列表，存储子节点的特征取值。 |
| qreal | distance=120.0 | 节点之间的水平距离。 |
| 函数 | void | boundingRect() | 返回节点的边界矩形。 |
| void | createChildItems() | 根据节点的子节点信息创建所有子节点的 NodeItem 对象。 |
| double | drawNode(QPainter\* painter, int depth) | 绘制节点的形状和文本，并绘制子节点的连线和文本。 |
| double | paint(QPainter\* painter, const QStyleOptionGraphicsItem\* option, QWidget\* widget) | 绘制节点的外观。 |
| TreeNode\* | setScene(QGraphicsScene\* scene) | 设置节点所属的图形场景。 |
| QString | getX() | 返回节点的 x 坐标 |
| void | getY() | y 坐标。 |
| void | getFeature() | 返回节点的属性。 |
| void | getClassification() | 返回叶子节点的决策结果。 |
| QjsonObject | isLeaf() | 判断节点是否为叶子节点。 |
| void | itemChange(GraphicsItemChange change, const QVariant& value) | 重写该函数以捕获节点位置的变化，并限制节点的移动范围在场景内。 |
| void | mousePressEvent(QGraphicsSceneMouseEvent\* event) | 处理鼠标按下事件，初始化拖动操作。 |
| void | mouseMoveEvent(QGraphicsSceneMouseEvent\* event) | 处理鼠标移动事件，开始拖动操作。 |
| void | mouseReleaseEvent(QGraphicsSceneMouseEvent\* event) | 处理鼠标释放事件，完成拖动操作。 |

**3.核心算法**

（1）**决策树构造算法**

这里首先介绍ID3算法，本决策树算法是基于ID3算法而构建的

要介绍ID3算法，还需要介绍几个基本概念

<1>**信息熵**

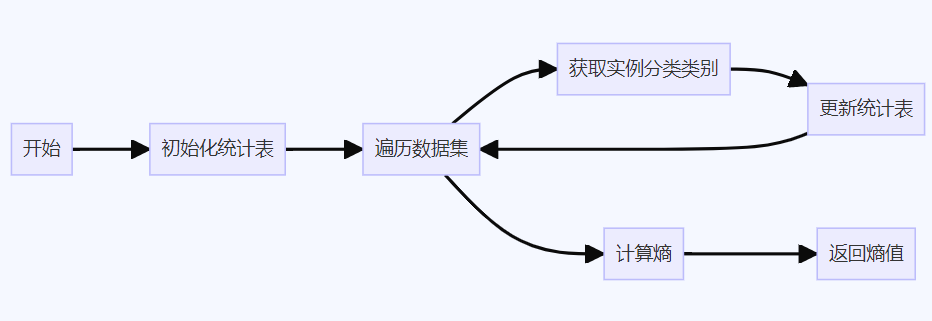
在信息论中，信息熵定义为信息的期望值。如果待分类的事物可能划分在多个类之中，则符号的信息定义为：，其中为选择该分类的概率。为了计算信息熵，还需要计算所有类别的所有可能值所包含的信息期望值：



其中的n为分类数目，由以上公式即可计算出一个数据集的信息熵。

在本次课设中，计算信息熵（经验熵）的函数calculateEntropy算法如下：

* 初始化一个空的哈希表 count，用于存储不同分类的数量。
* 遍历数据集 data 中的每一行（实例），从索引 1 开始（跳过属性列或标签行，所以此算法对于数据集本身需要处理，所计算的数据集需要第一例为序号列，第一行为标签行，且最后一列为分类列）：
* 获取当前实例的分类类别，即该行的最后一个元素。
* 在哈希表 count 中查找该分类类别是否已存在。
* 如果存在，将对应分类的数量加一。如果不存在，将该分类类别作为键，初始数量为 1 存入哈希表 count。
* 初始化经验熵 entropy 为 0.0。
* 计算总实例数 totalInstances，即数据集的行数减一（因为第一行是属性行）。
* 遍历哈希表 count 中的每个分类类别：
* 计算当前分类的概率 probability，即该分类的数量除以总实例数。
* 使用公式 ，其中 p 是当前分类的概率，更新经验熵 entropy。
* 返回计算得到的经验熵 entropy。



< 2>**信息增益**

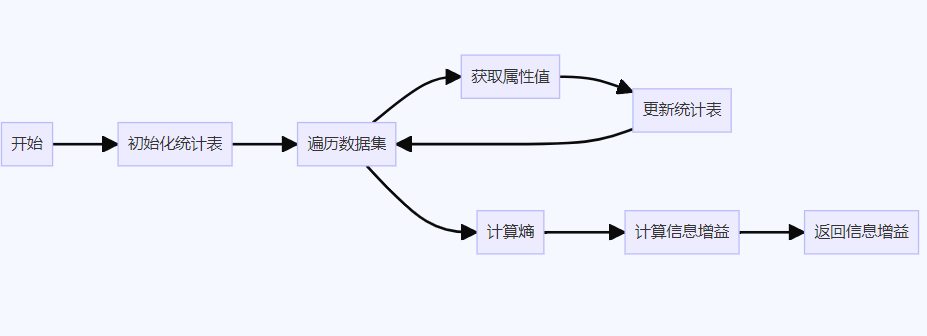
信息增益是相对于特征而言的。所以，特征A对训练数据集D的信息增，定义为集合D的经验熵H(D)与特征A给定条件下D的经验条件熵之差，即：



在此程序中，有两个函数用到了信息增益，分别是信息增益计算函数以及求最优特征的函数，总流程如下：

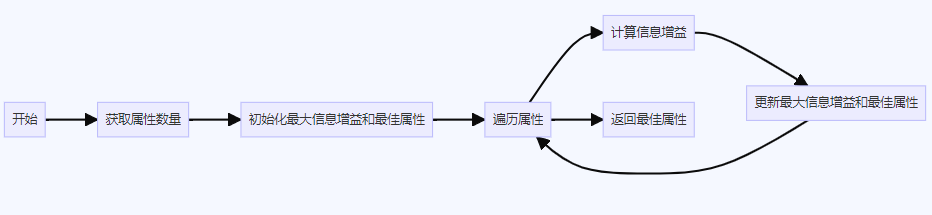
**calculateInformationGain函数：**

* 初始化一个空的哈希表 count，用于统计每个属性值的数量。
* 遍历数据集 data 中的每一行（实例），从索引 1 开始（跳过属性行）：
* 获取当前实例在给定属性索引 attributeIndex 处的属性值。
* 在哈希表 count 中查找该属性值是否已存在。
* 如果存在，将对应属性值的数量加一。
* 如果不存在，将该属性值作为键，初始数量为 1 存入哈希表 count。
* 初始化总实例数 totalInstances 为数据集的行数减一（因为第一行是属性列或标签行）。
* 计算初始信息增益 informationGain，通过调用 calculateEntropy 函数计算整个数据集的经验熵。
* 对哈希表 count 中的每个属性值进行迭代：
* 创建一个子集 subset，用于存储具有当前属性值的实例。
* 遍历数据集 data 中的每一行（实例），从索引 1 开始：
* 如果当前实例在给定属性索引 attributeIndex 处的属性值等于当前属性值，则将该实例添加到子集 subset 中。
* 计算当前属性值的概率 probability，即该属性值的数量除以总实例数。
* 计算当前属性值的信息增益，并通过调用 calculateEntropy 函数计算子集 subset 的经验熵，用于更新信息增益 informationGain。
* 返回计算得到的信息增益 informationGain。



**selectBestAttribute函数：**

* 获取数据集中属性的数量 numAttributes，通过数据集的第一行的大小减去 2（因为减去决策结果列和编号列）得到。
* 初始化最大信息增益 maxInformationGain 为 0.0，并初始化最佳属性 bestAttribute 为空。
* 对每个属性进行迭代，从索引 1 开始到 numAttributes：
* 调用 calculateInformationGain 函数计算当前属性的信息增益 informationGain。
* 如果当前信息增益大于最大信息增益 maxInformationGain，则更新最大信息增益和最佳属性。
* 返回具有最大信息增益的属性 bestAttribute



有了上述的铺垫，现在可以进行决策树的构建了，假设数据已经每行读取到了QVector<QVector<QString>>中，具体的读取函数算法在后文会介绍。

* 创建根节点：

创建一个新的TreeNode对象作为根节点。

* 检查数据集中的决策结果是否一致：

初始化布尔变量sameDecision为true。

获取数据集中第一个实例的决策结果，存储在变量decision中。

遍历数据集中的每个实例，从索引1开始：

获取当前实例的决策结果，如果与变量decision不相等，则将sameDecision设置为false并跳出循环。

如果sameDecision为true，即所有实例的决策结果一致，将决策结果赋值给根节点的decision属性，并返回根节点。

* 检查是否所有属性都已经用完：

检查数据集中的属性列数减去决策结果列和编号列的结果是否为1。

如果为1，表示所有属性都已经用完，没有可分割的属性，将决策结果中出现最频繁的值赋值给根节点的decision属性，并返回根节点。

* 选择最佳属性作为当前节点的属性：

调用selectBestAttribute函数选择具有最大信息增益的属性，并将结果赋值给根节点的feature属性。

* 获取最佳属性的所有可能取值：

初始化一个字符串列表attributeValues用于存储最佳属性的所有可能取值。

通过遍历数据集中第一行的属性，找到最佳属性的索引attributeIndex。

遍历数据集中的每个实例，从索引1开始：

获取当前实例在最佳属性索引处的属性值，如果attributeValues中不包含该值，则将该值添加到attributeValues中。

* 递归构建子节点：

对于最佳属性的每个可能取值value：

创建一个空的子数据集subset，将数据集的第一行去除最佳属性的列后添加到subset中。

遍历数据集中的每个实例：

如果实例在最佳属性索引处的属性值等于当前取值value，则将去除最佳属性列后的实例添加到subset中。

检查subset的长度：

如果长度为1，表示子集为空，创建一个叶子节点：

创建一个新的TreeNode对象作为叶子节点。

统计数据集中各个决策结果的数量，找到出现最频繁的决策结果，并将其赋值给叶子节点的decision属性。

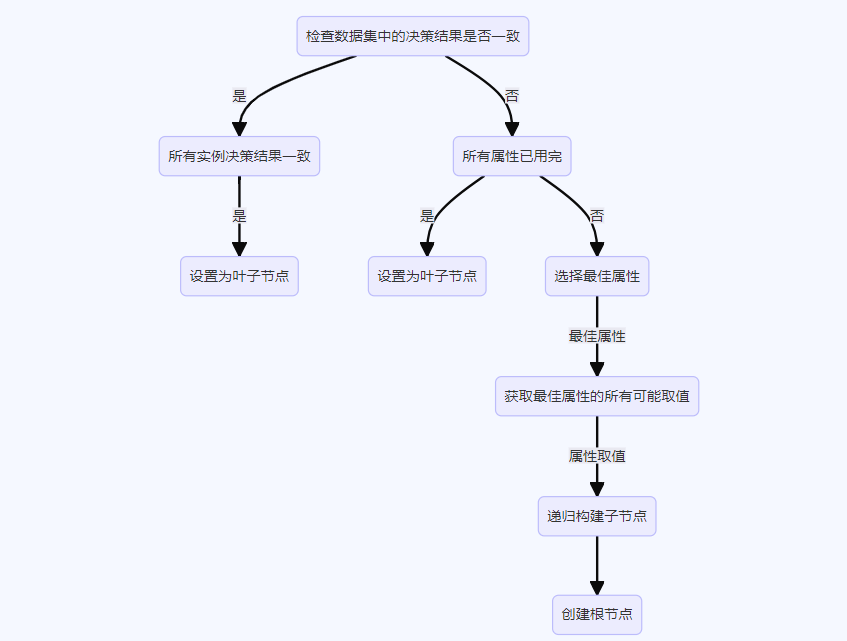
将叶子节点添加到根节点的children字典中，使用当前取值value作为键。

否则，递归构建子节点：

调用buildDecisionTree函数，传递子集subset作为参数，获得返回的子节点。

将子节点添加到根节点的children字典中，使用当前取值value作为键。

* 返回根节点。



由上述函数，即可构建决策树，并将根节点保存在类的成员中，以供后续使用。

（2）**决策树可视化算法**

这个算法应该是此程序中最为难实现的算法，由于实现的前辈较少，资料相当的稀少。

为了实现可视化，主要利用了Qt的graphicsView，具体通过以下函数（drawTree）实现：

* 检查节点是否为空：

如果节点为空，直接返回。

* 创建节点对象：

使用节点的feature、value和decision属性创建一个新的NodeItem对象。

将节点对象的位置设置为相对于父节点的位置，默认为 (0, 0)。

将节点对象添加到场景中。

计算子节点的间距：

获取子节点的个数n。

根据场景的宽度为800，将其平均分为n段，计算每段的长度。

将长度调整为100的倍数。(为了更美观)

* 绘制子节点：

初始化子节点的起始位置childX为 -100 - (子节点个数 - 1) \* 100，用于调整子节点的水平位置。

初始化子节点的垂直位置childY为200。

对于节点的每个子节点：

获取子节点的引用和键值value。

创建子节点的NodeItem对象，并设置其位置为(childX, childY)。

将子节点的NodeItem对象添加到场景中。（很重要）

将子节点的NodeItem对象添加为父节点的子节点，并使用value作为键。

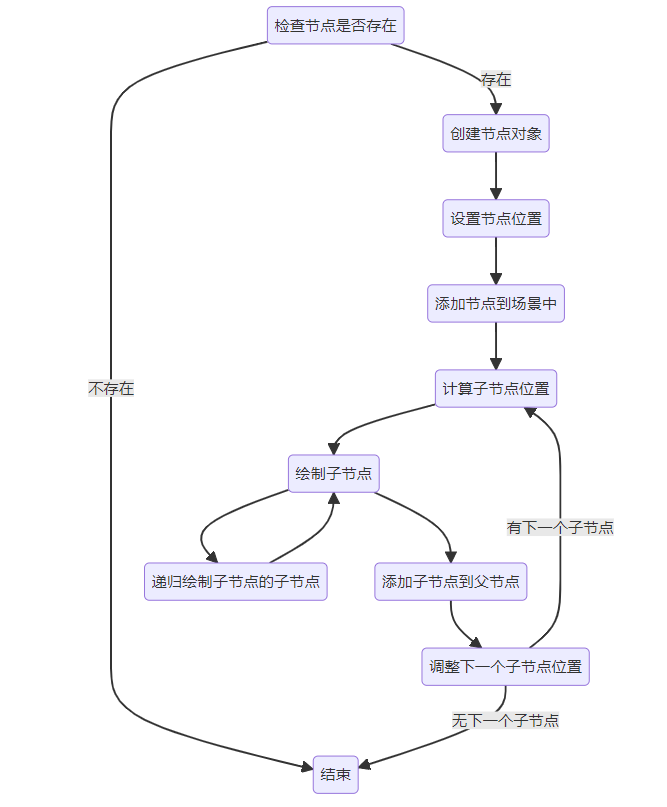
递归调用drawTree函数绘制子节点的子节点。

调整下一个子节点的位置，将childX增加子节点间距length。

这样，整个drawTree函数完成了对决策树的可视化绘制。绘制过程是递归进行的，从根节点开始，依次绘制每个节点及其子节点，直到叶子节点为止。节点的位置是根据父节点的位置和子节点间距进行计算和调整的。

这里用到一个关键的类NodeItem，继承于graphicsView，其功能如下：

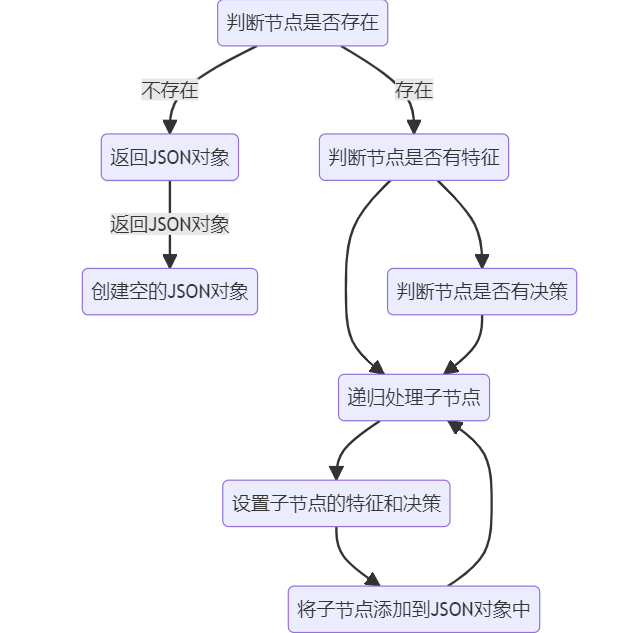
* 绘制决策树的节点项，包括节点的矩形框、决策文本、特征信息和连线特征值。
* 允许节点项在场景中拖动，并自动更新子节点的位置。
* 可以添加子节点，并在节点间创建连线。



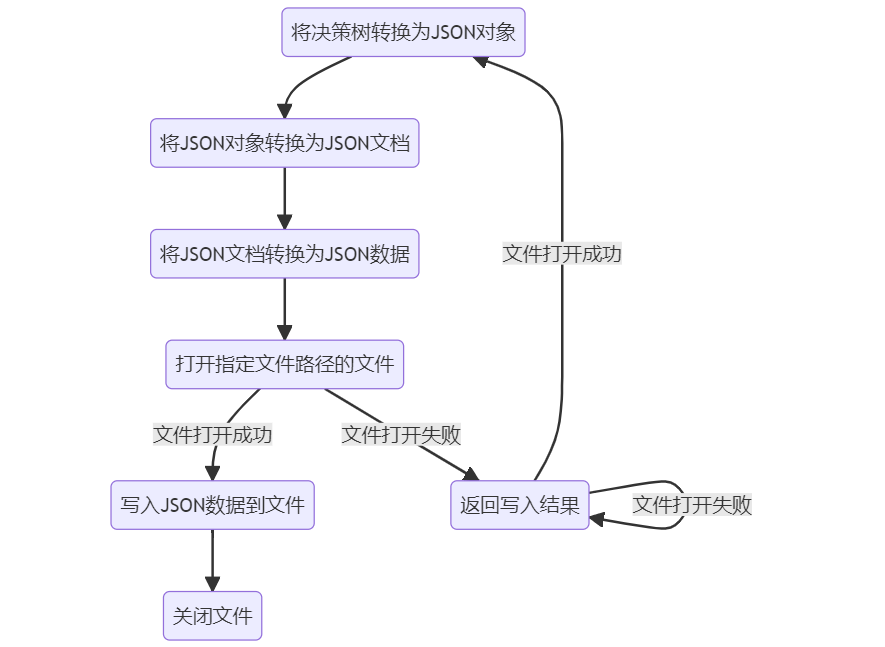
**（3）json树生成与读取算法**

这两个算法比较简单，这里不详细解释，详见流程图：

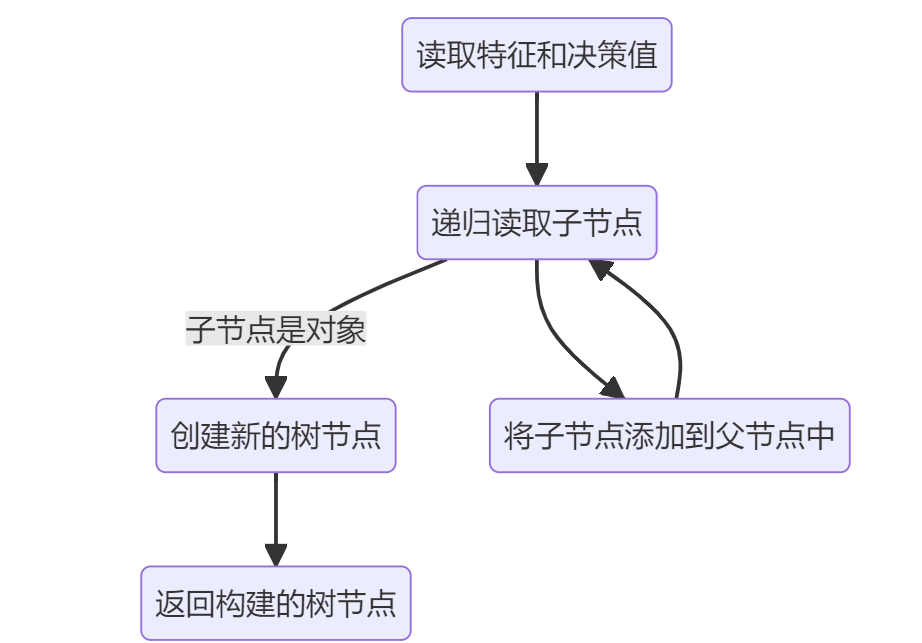
1. convertTreeToJson



2. writeTreeToJsonFile

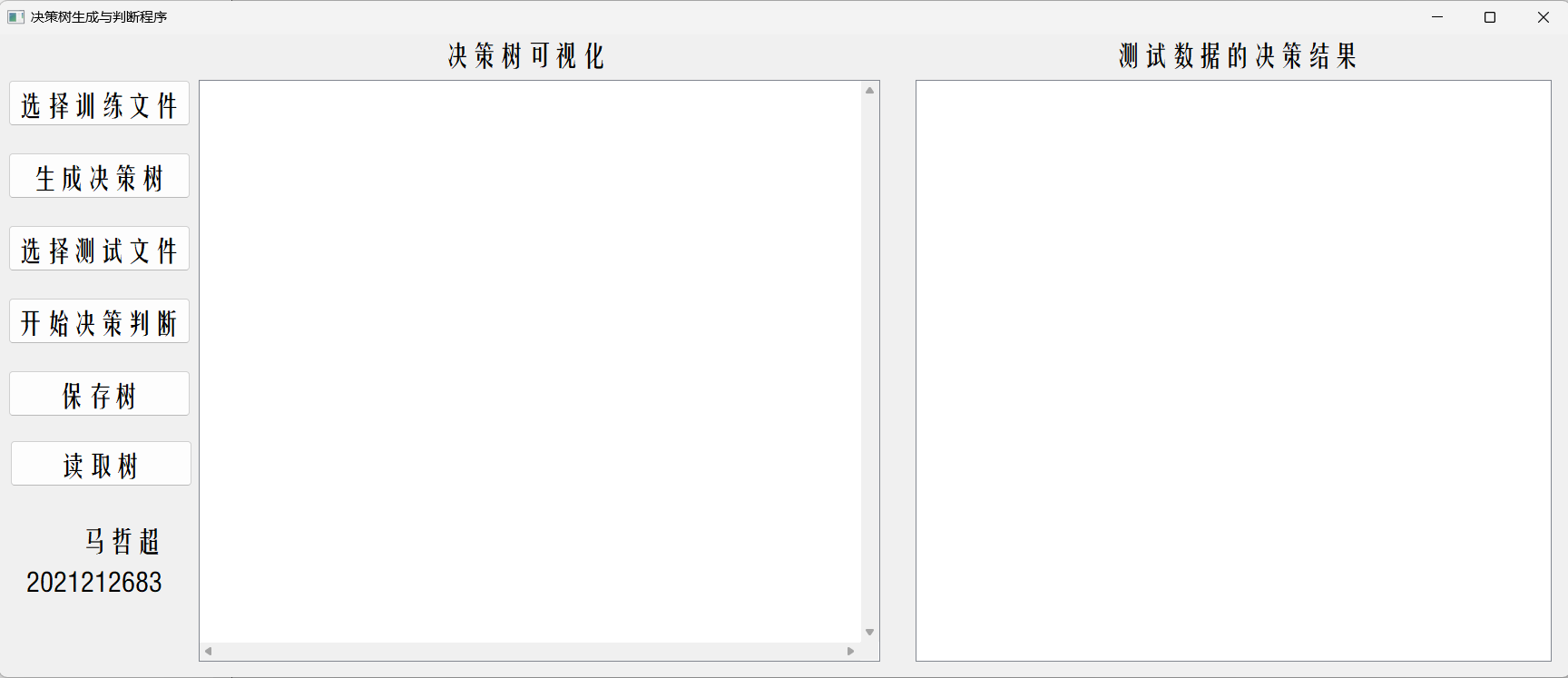


3. constructTreeFromJson



# （三）用户手册

打开此程序，界面如下：



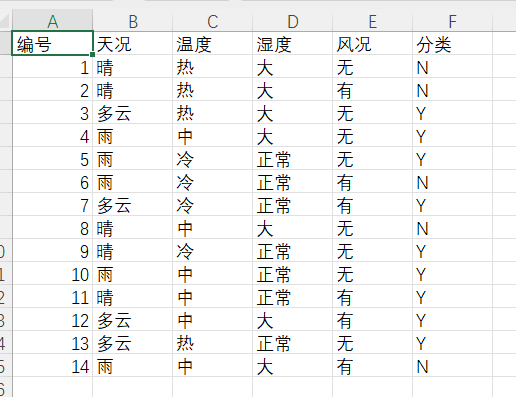
若是第一次使用，请点击“选择训练文件”选择您需要进行训练的数据集，这里对于数据集有一定要求：

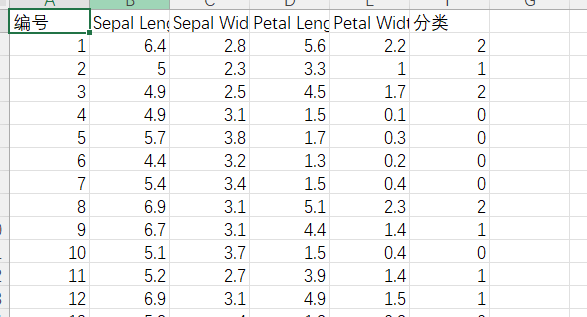
1. 第一行需为列标签，且最后一列为分类结果

2. 第一列需为编号

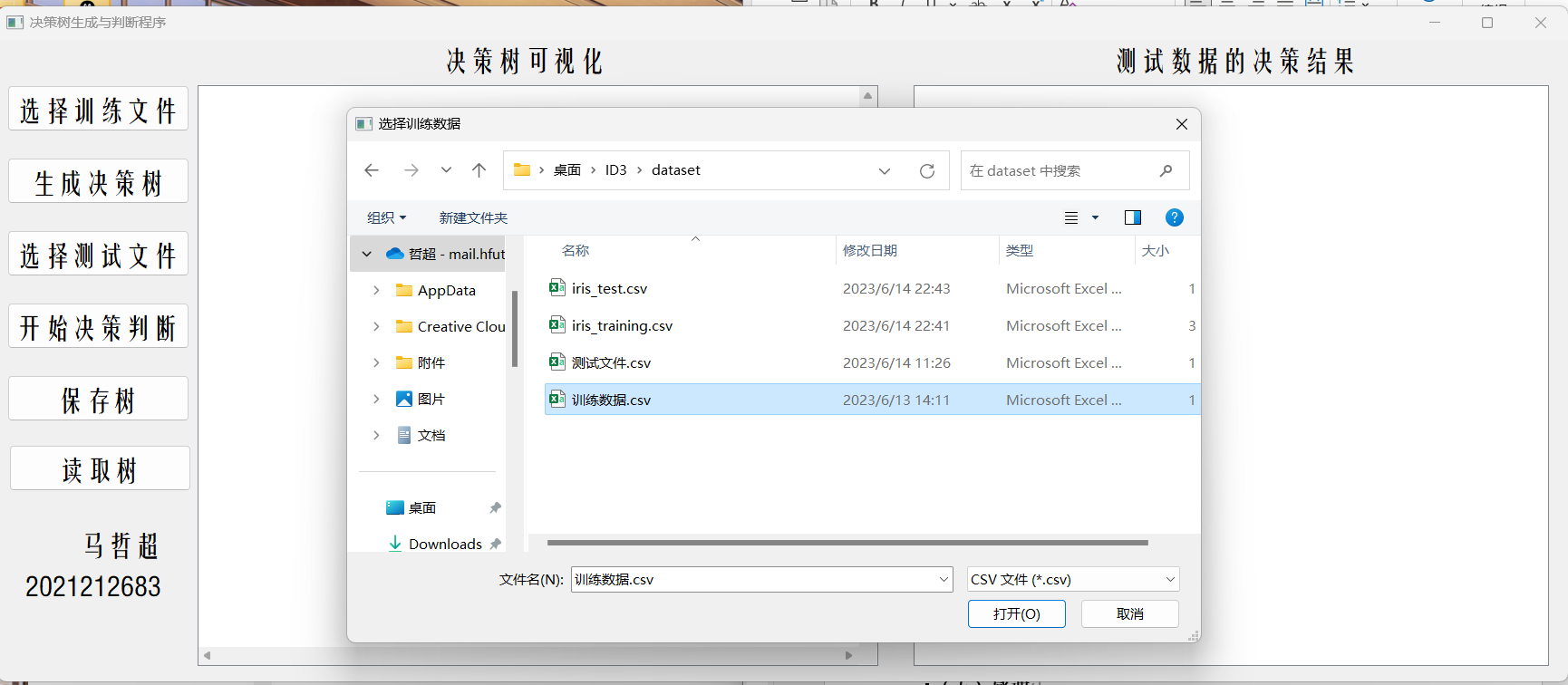
3. 文件需为.csv格式，包括后续的测试文件也需要为.csv格式

具体见如下两个例子：

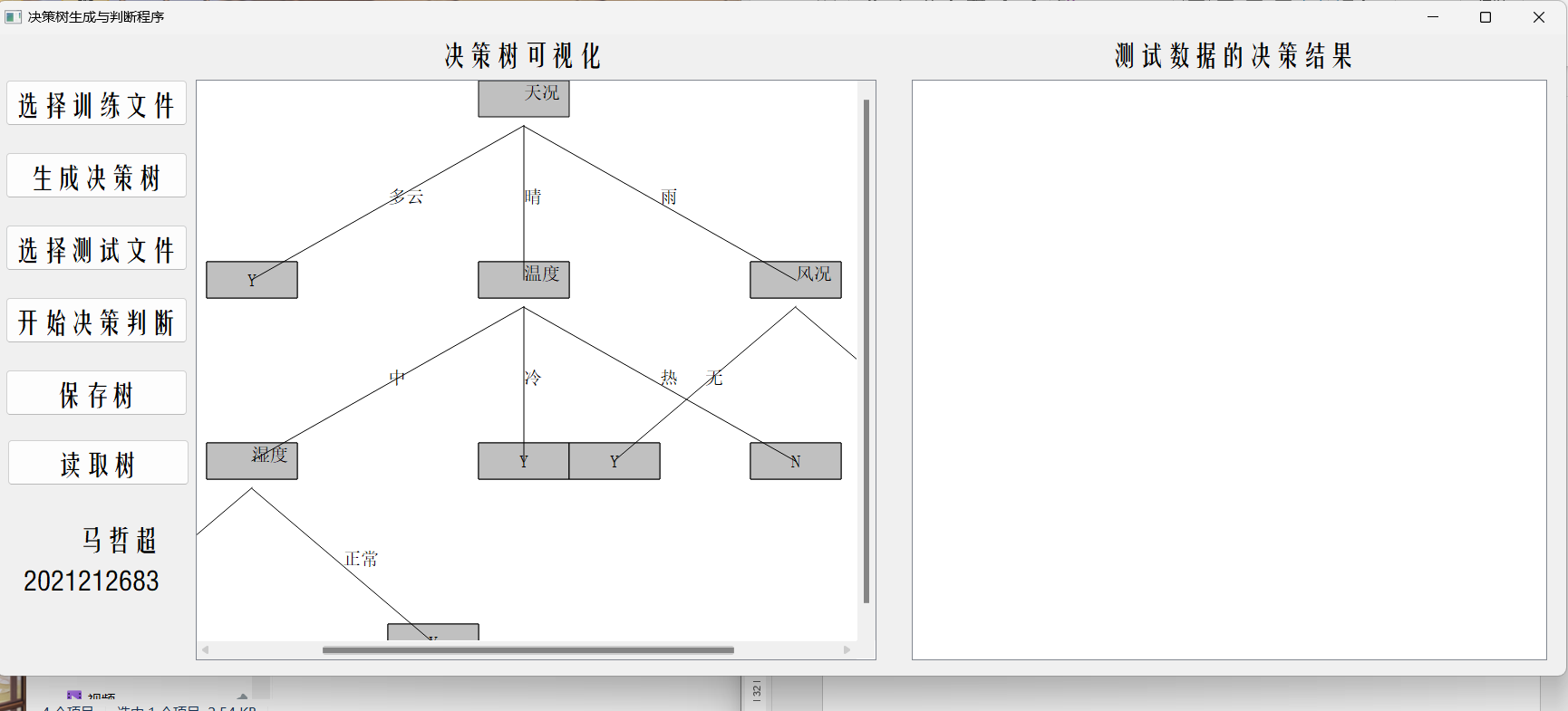




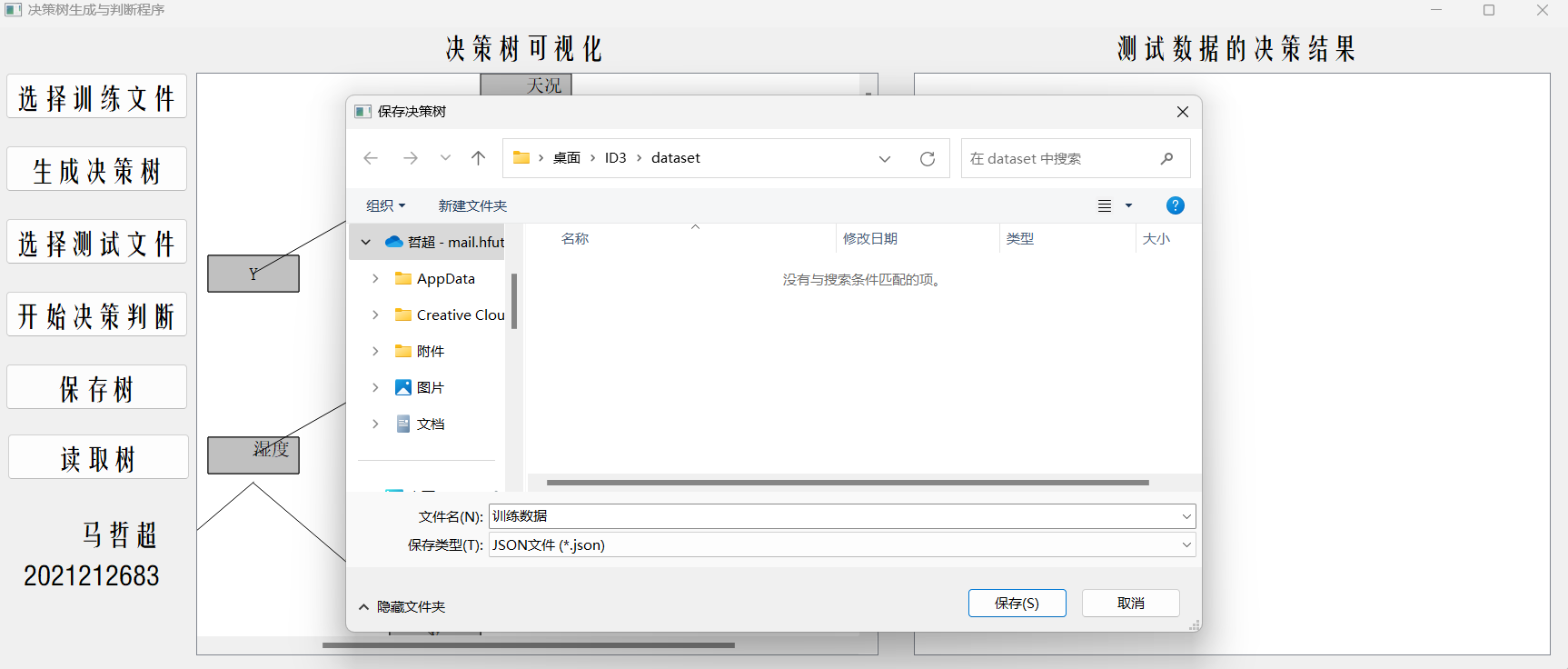
点击“选择训练文件”按钮后，选择您的csv文件进行训练，选择后程序会自动训练：



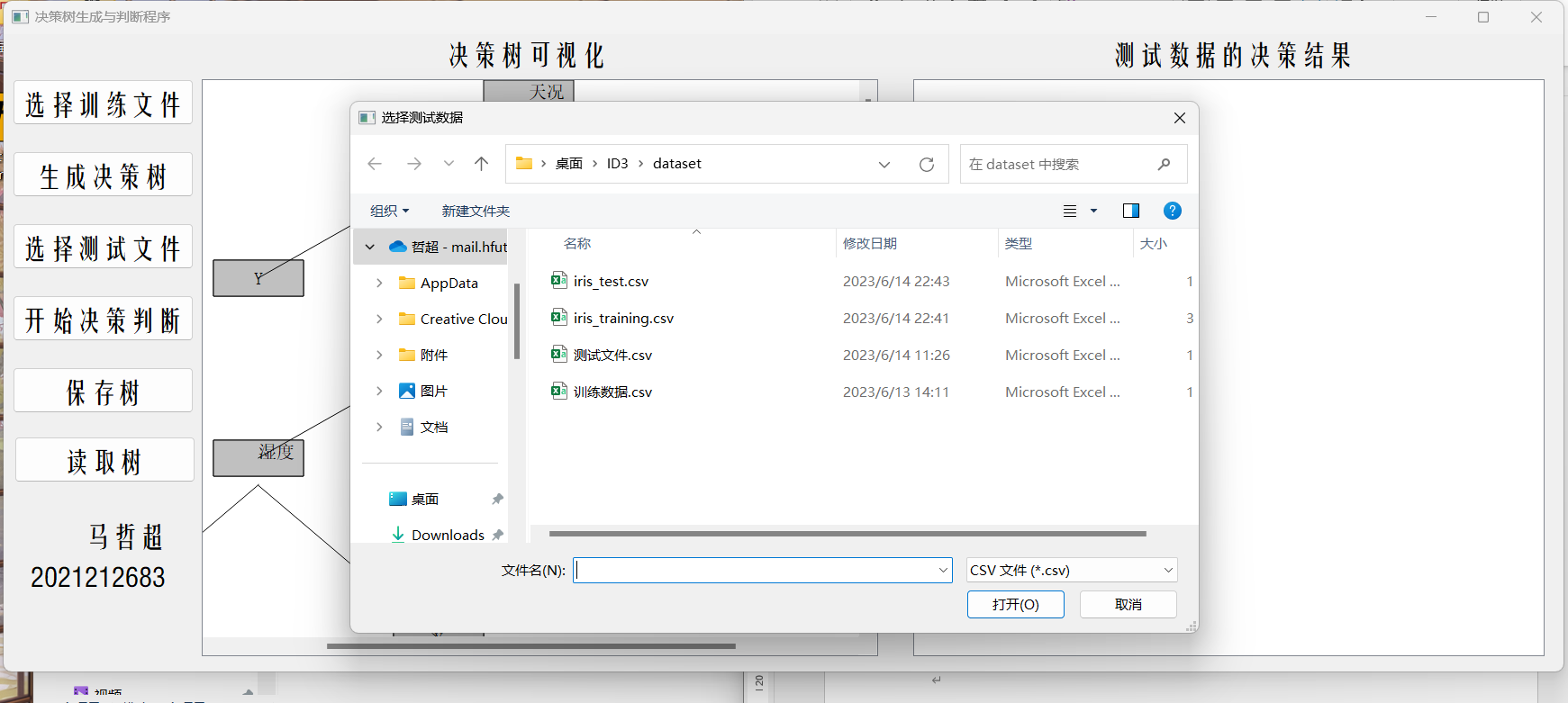
有了训练数据，用户可以选择可视化决策树和保存决策树，如果点击“生成决策树”，则会在“决策树可视化”板块显示用户的决策树：

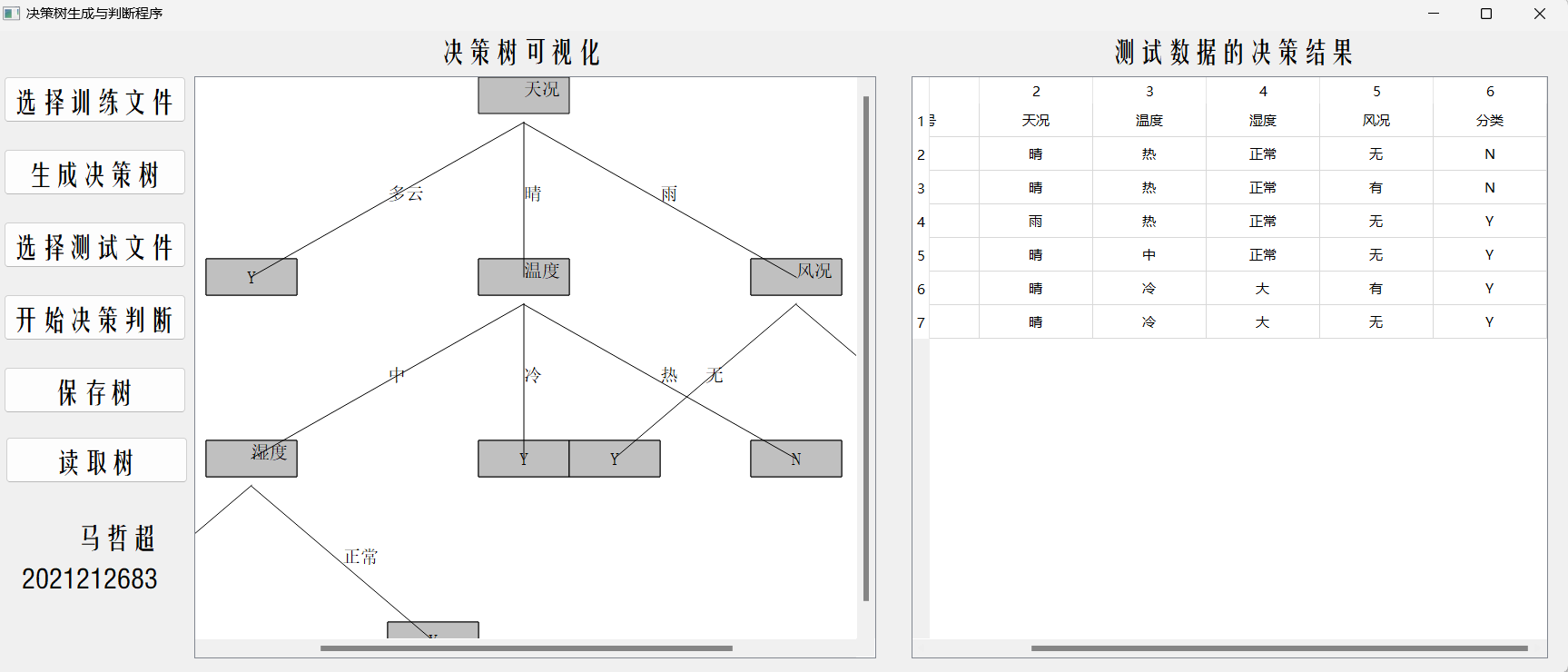


用户可以通过左右上下拖动来进行查看，如果点击“保存树”，则用户可以定义一个保存的树文件名并保存（.json）：

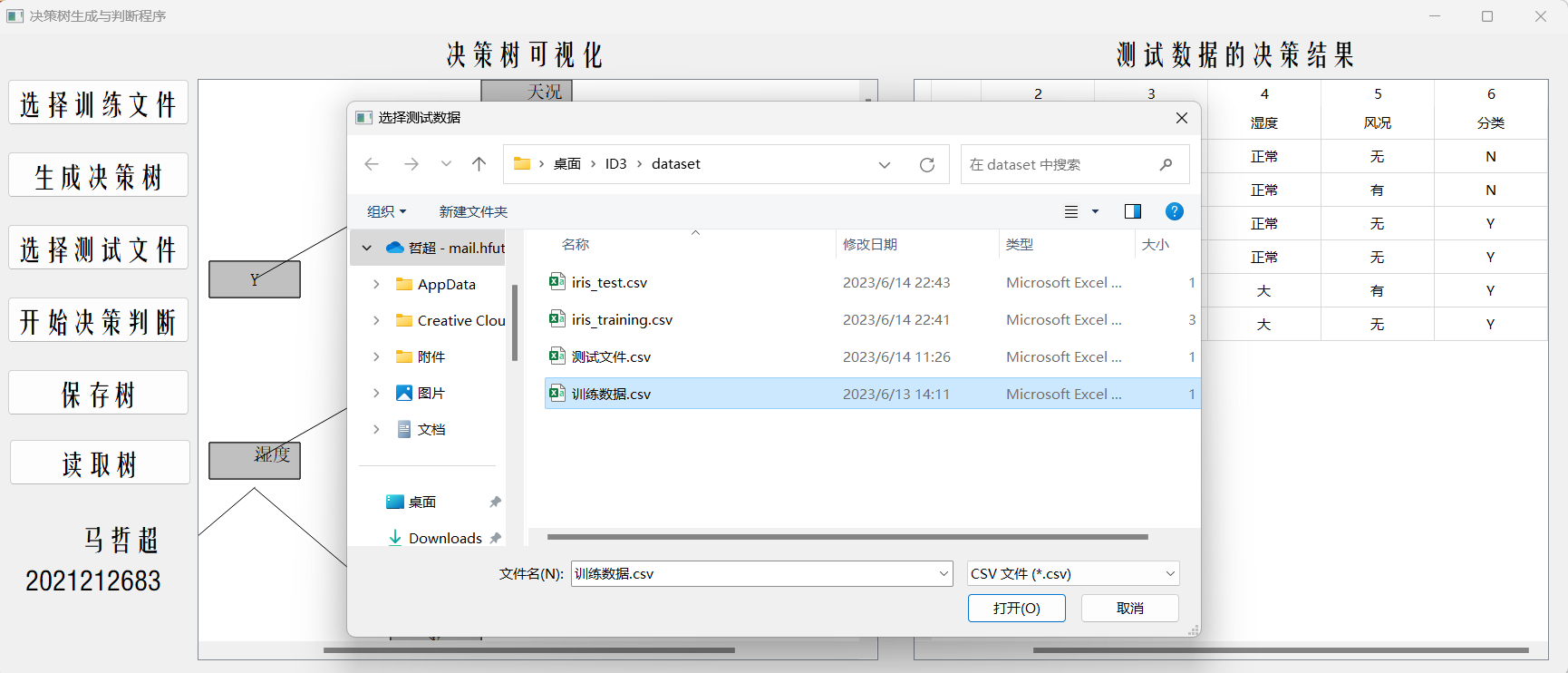


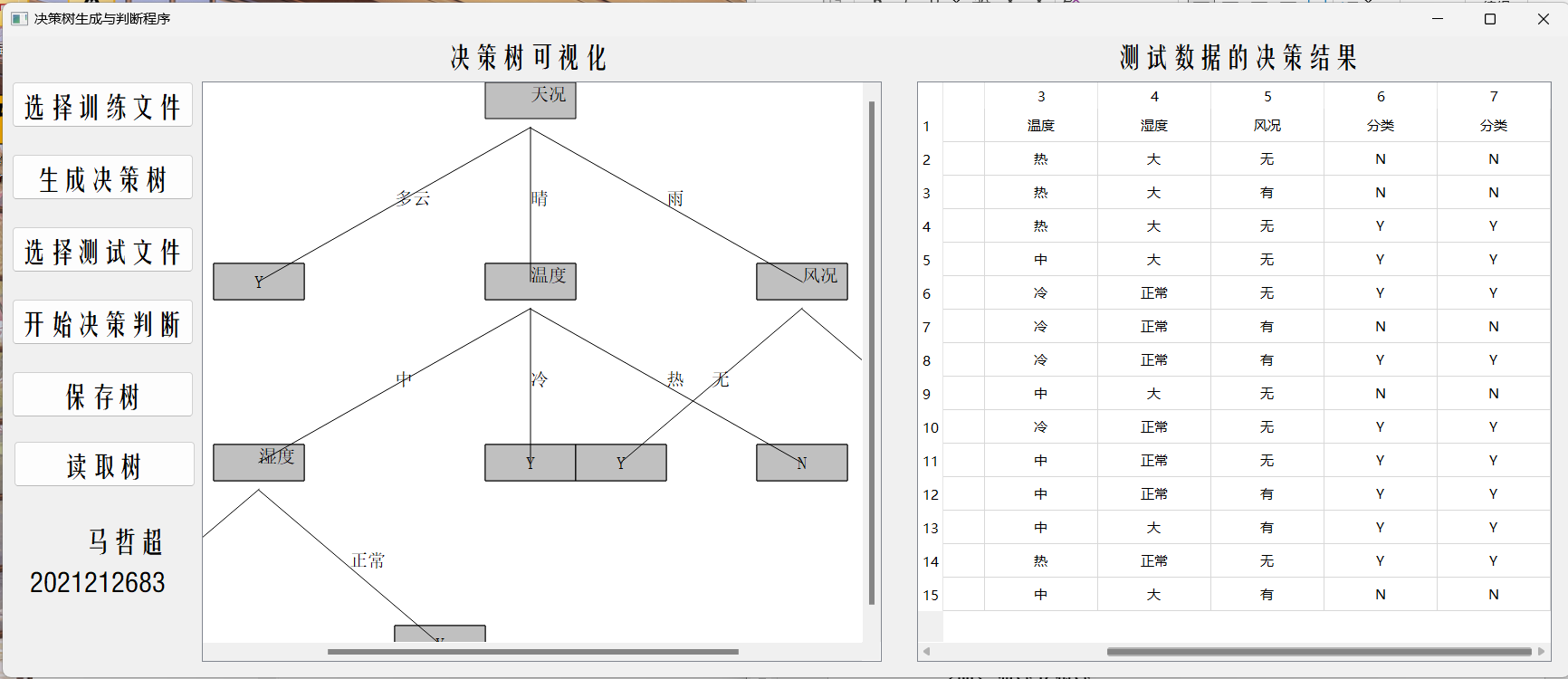
有了训练数据后，可以选择没有标签的测试数据（目标数据集）进行分类判断，且会在“测试数据的决策结果”一栏显示表格类型的决策结果，用户只需要点击“选择测试文件”选择文件后，点击“开始决策判断”即可：





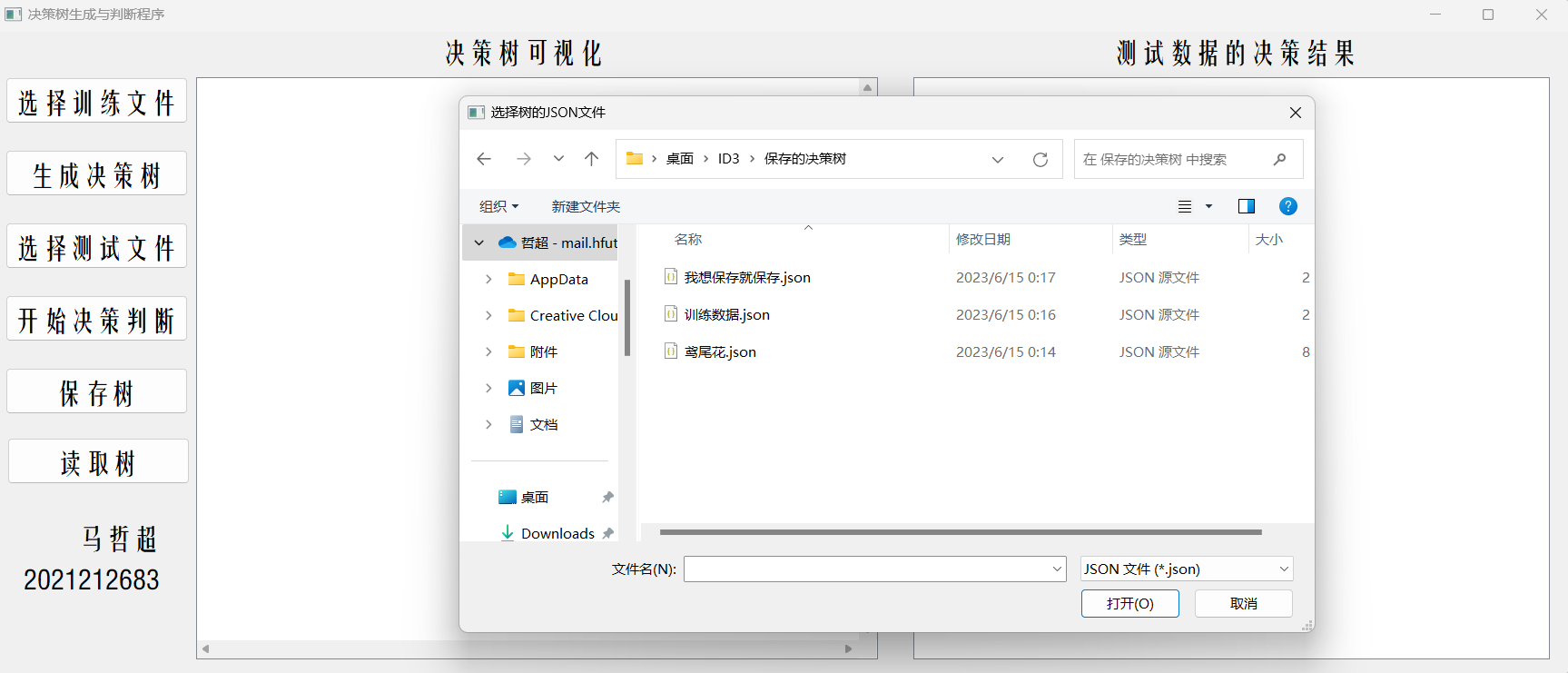
用户可以在选择训练数据的时候，选择用于测试的源文件进行树的测试，如下：



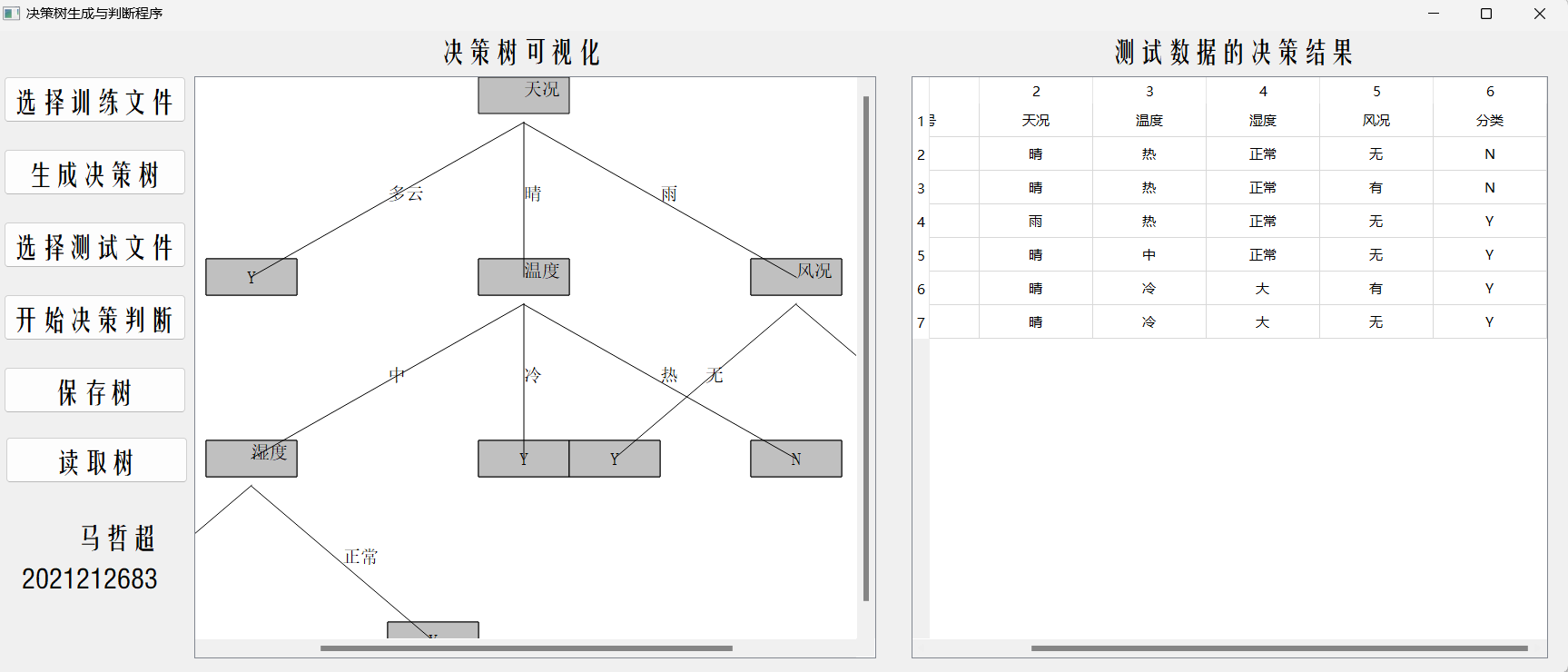


此时可以看出，会在最后新增一列分类列，用于与源数据的分类结果进行对比。

有了保存的树后，用户还可以直接打开软件，选择“读取树”：



读取后，也可以直接生成决策树与进行决策判断，如下：



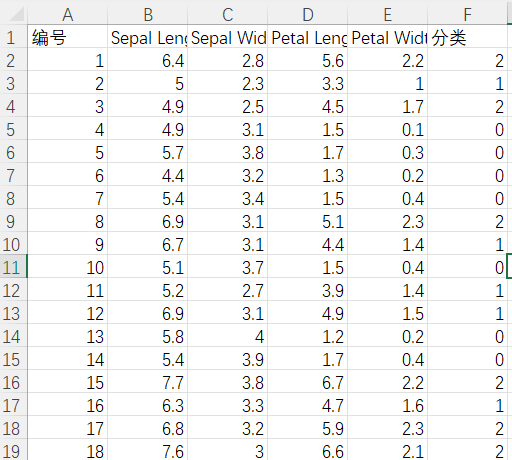
# （四）调试及测试

**1.测试数据**

对于普通数据集的测试结果可以参考用户手册，这里选择了鸢尾花数据集进行测试，其各参数如下：

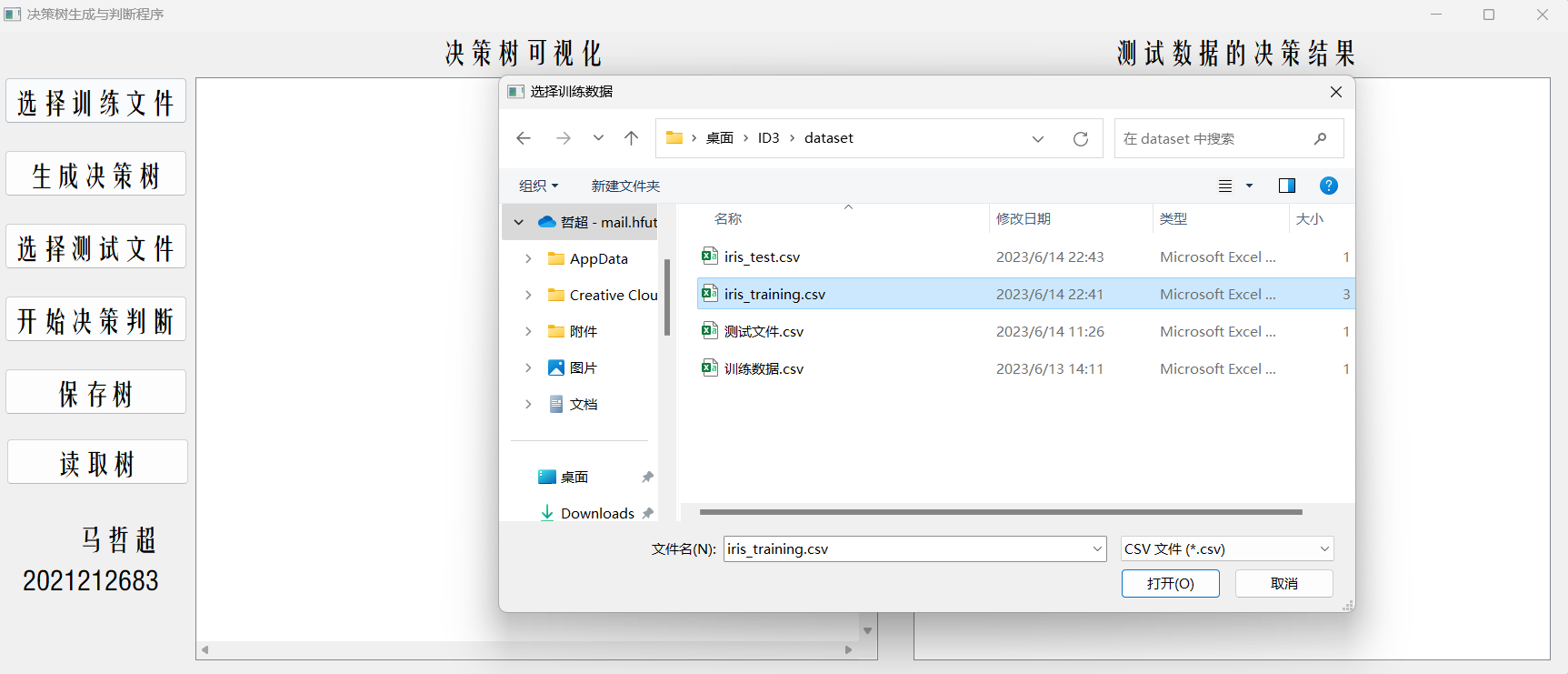
|  |  |
| --- | --- |
| 参数 | 值 |
| 训练数据集大小 | 120 |
| 测试数据集大小 | 30 |
| 特征个数 | 4 |
| 类别 | 3 |

部分如下（训练数据）：

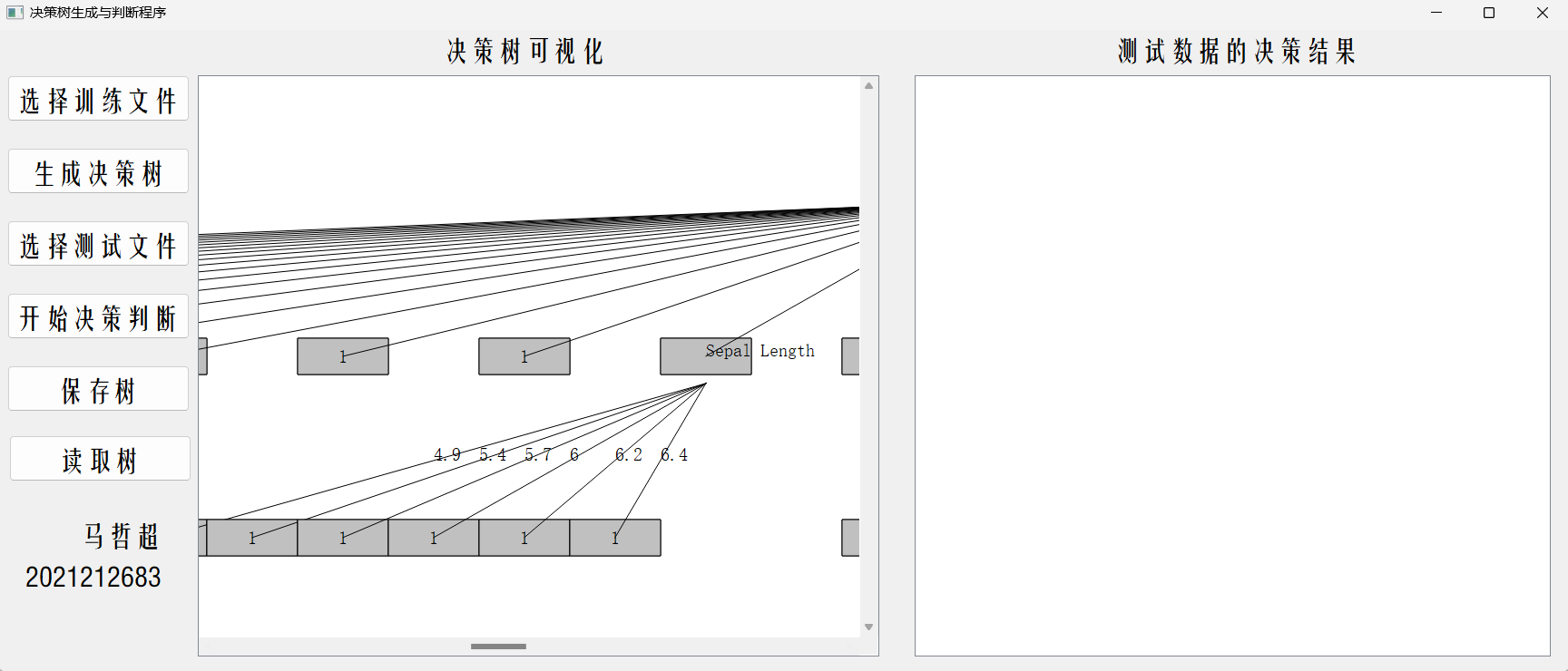


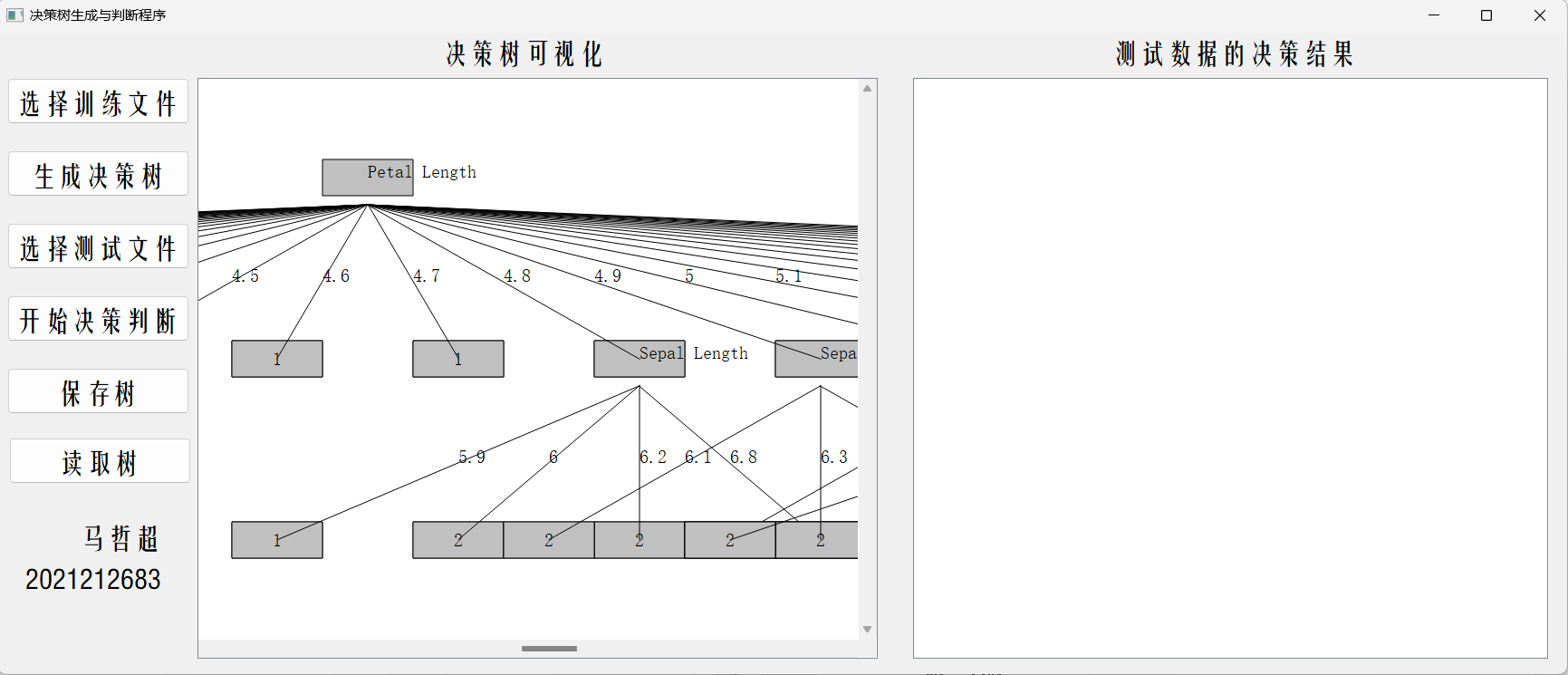
**2.测试结果**

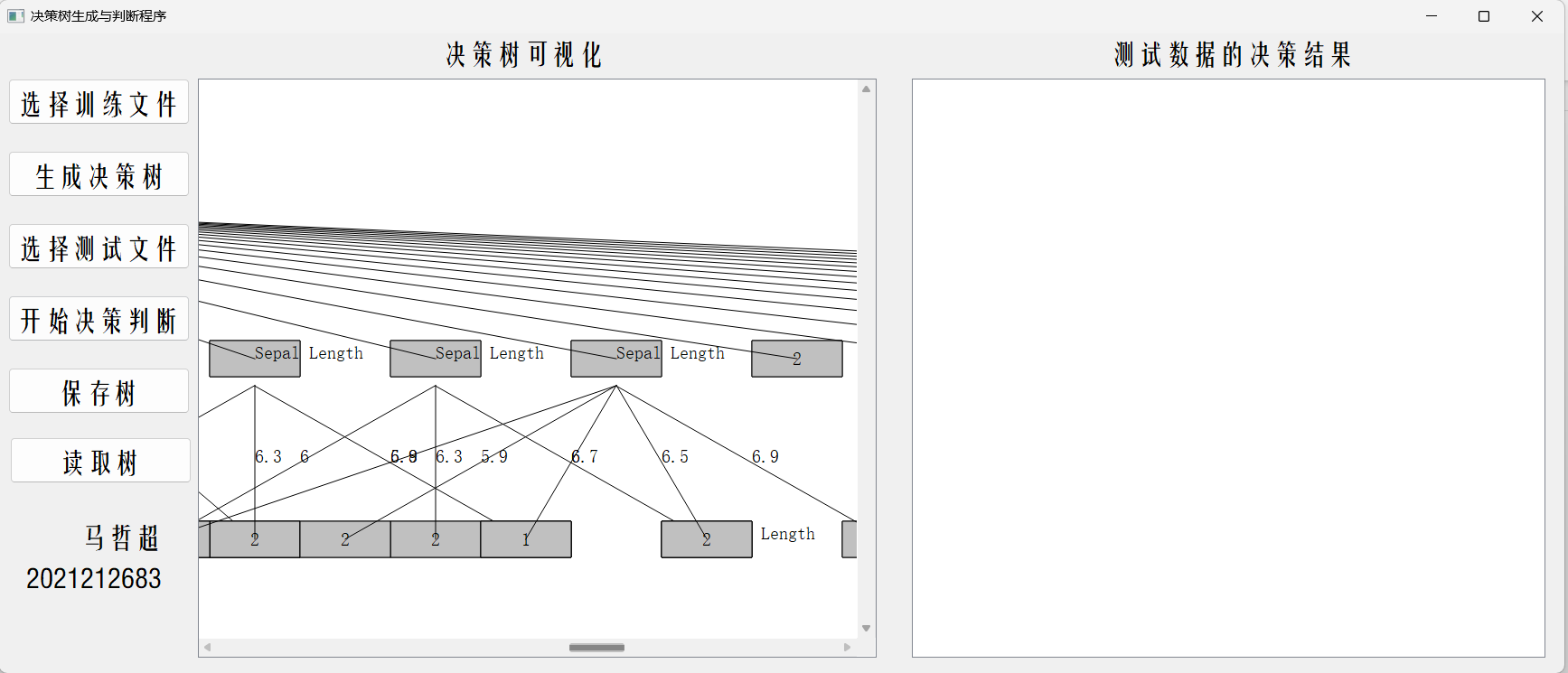
选择数据集：

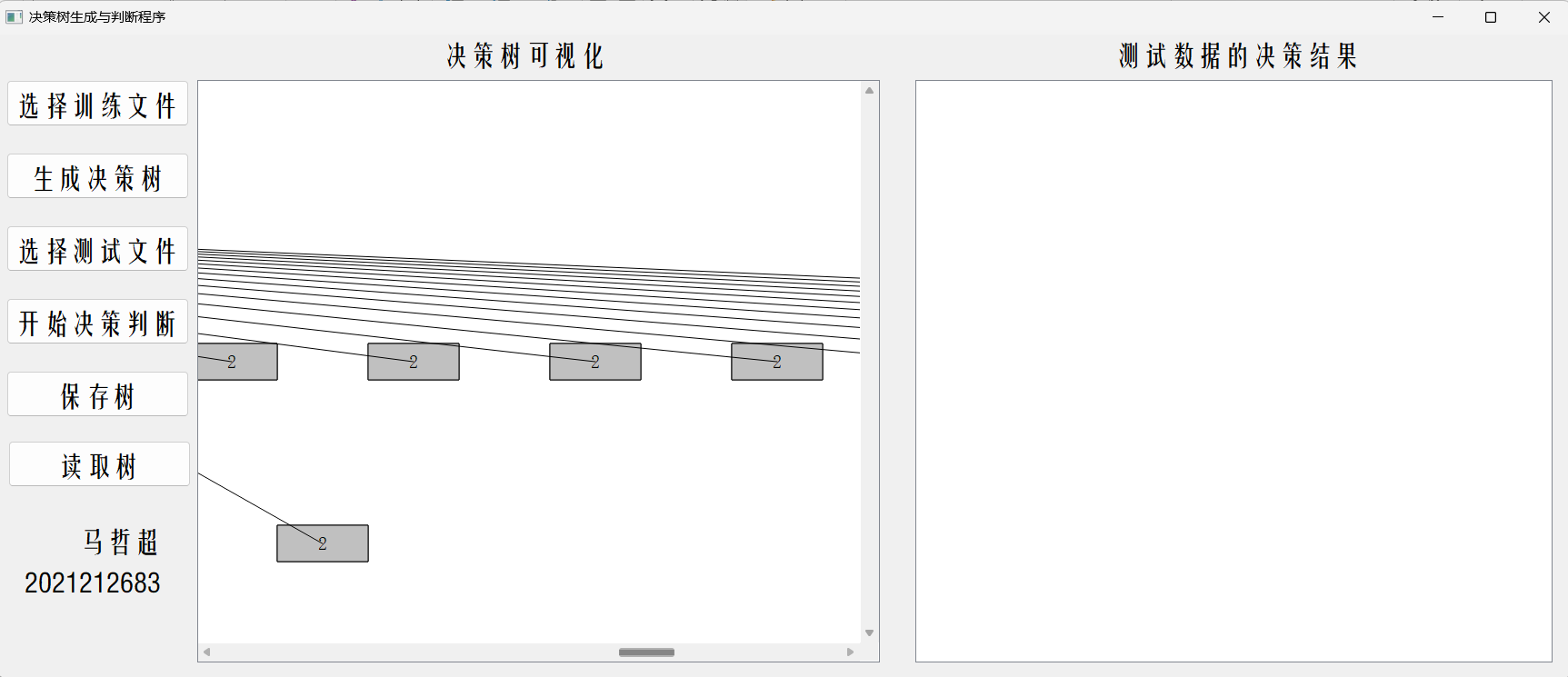


生成树：



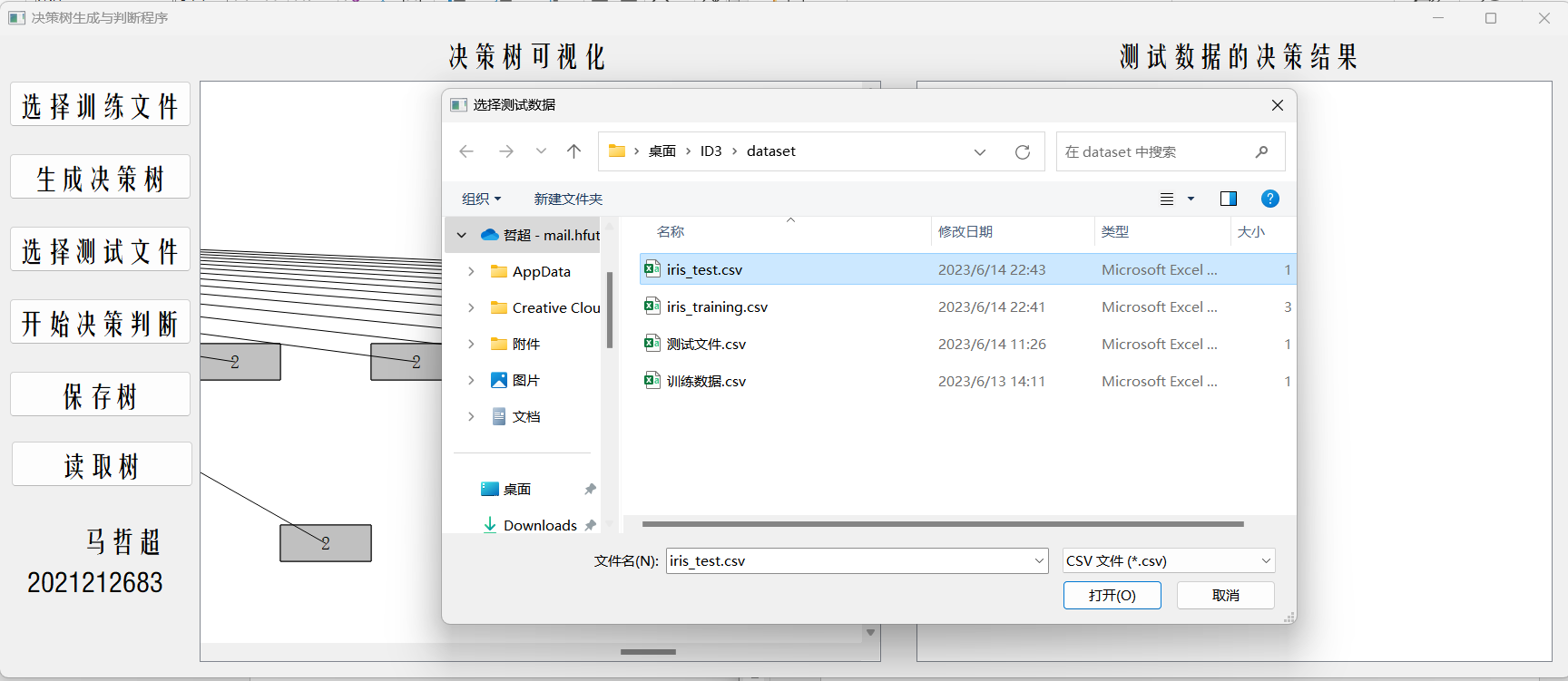




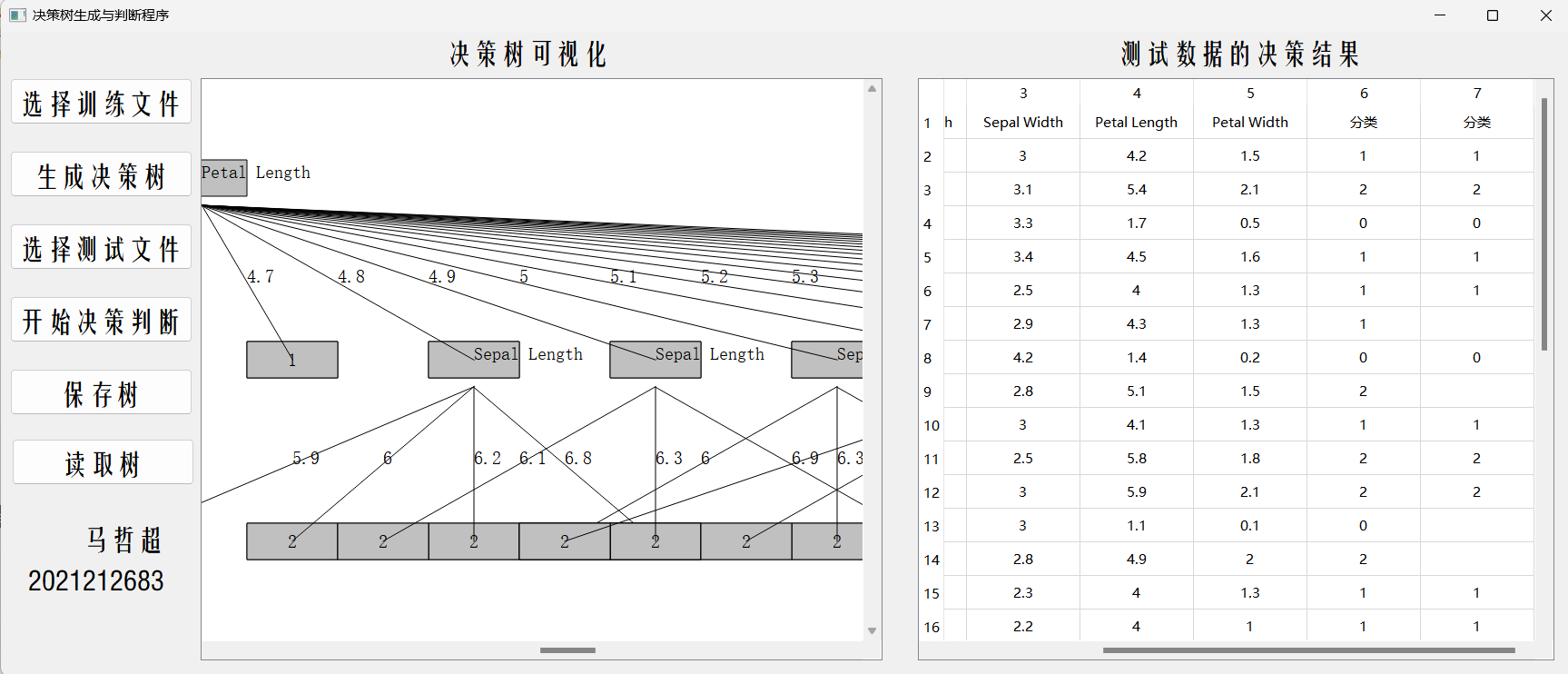


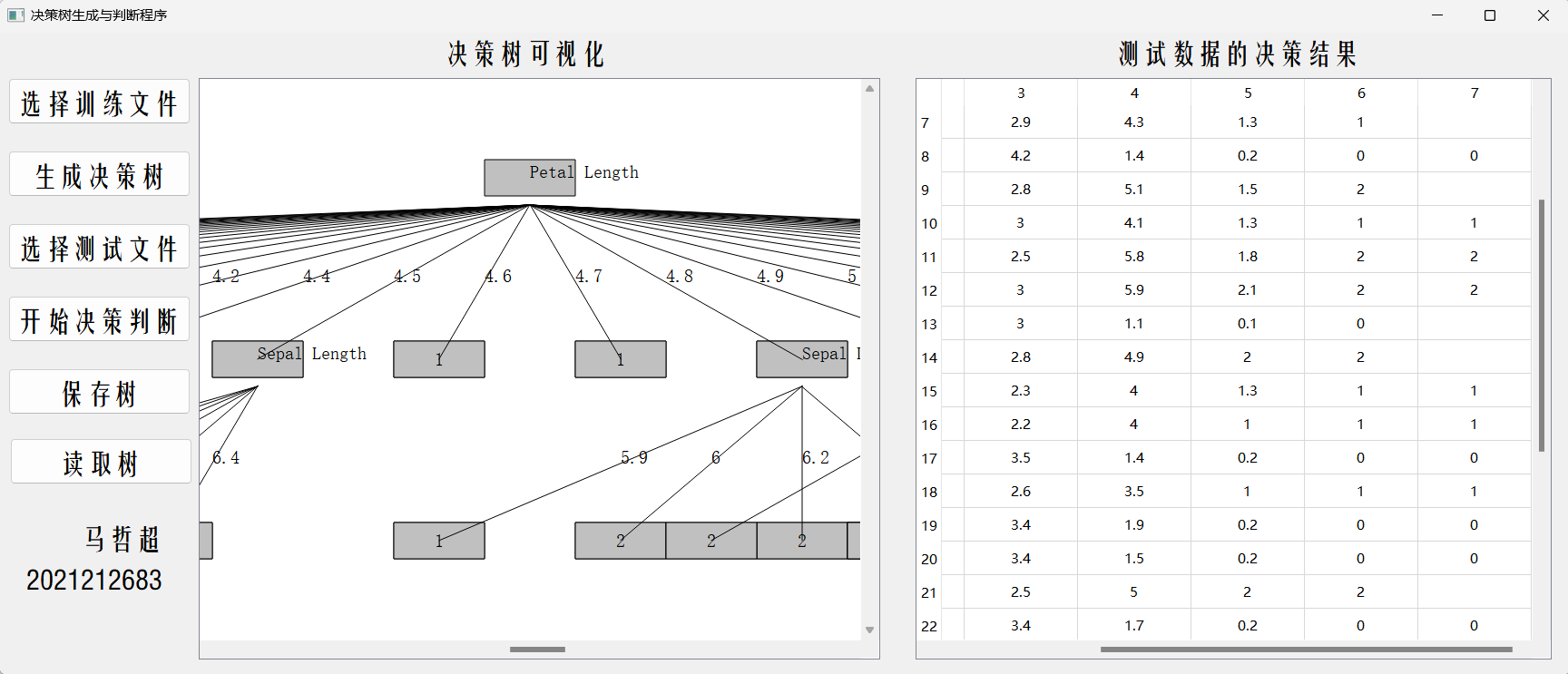
树比较大，未显示的部分均分类为2

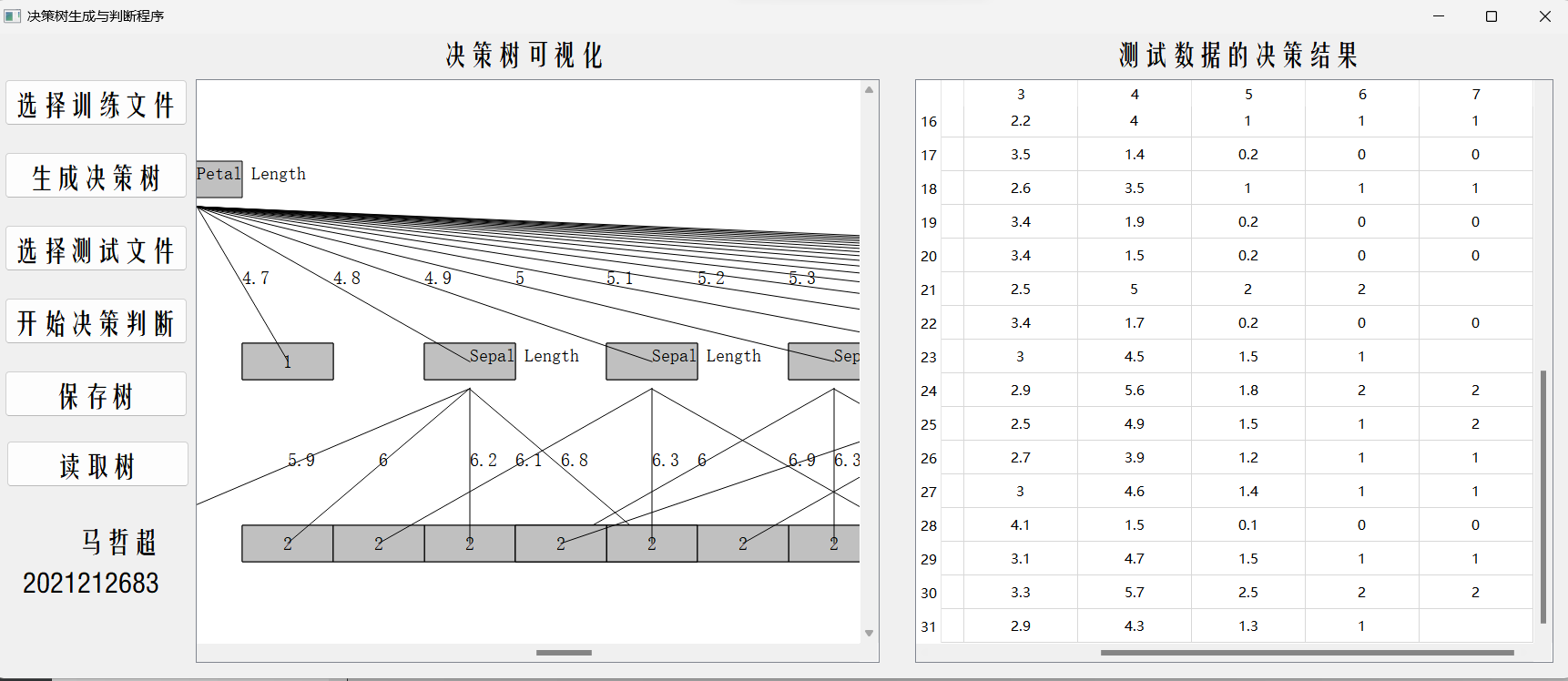
选择测试数据：



进行决策判断：

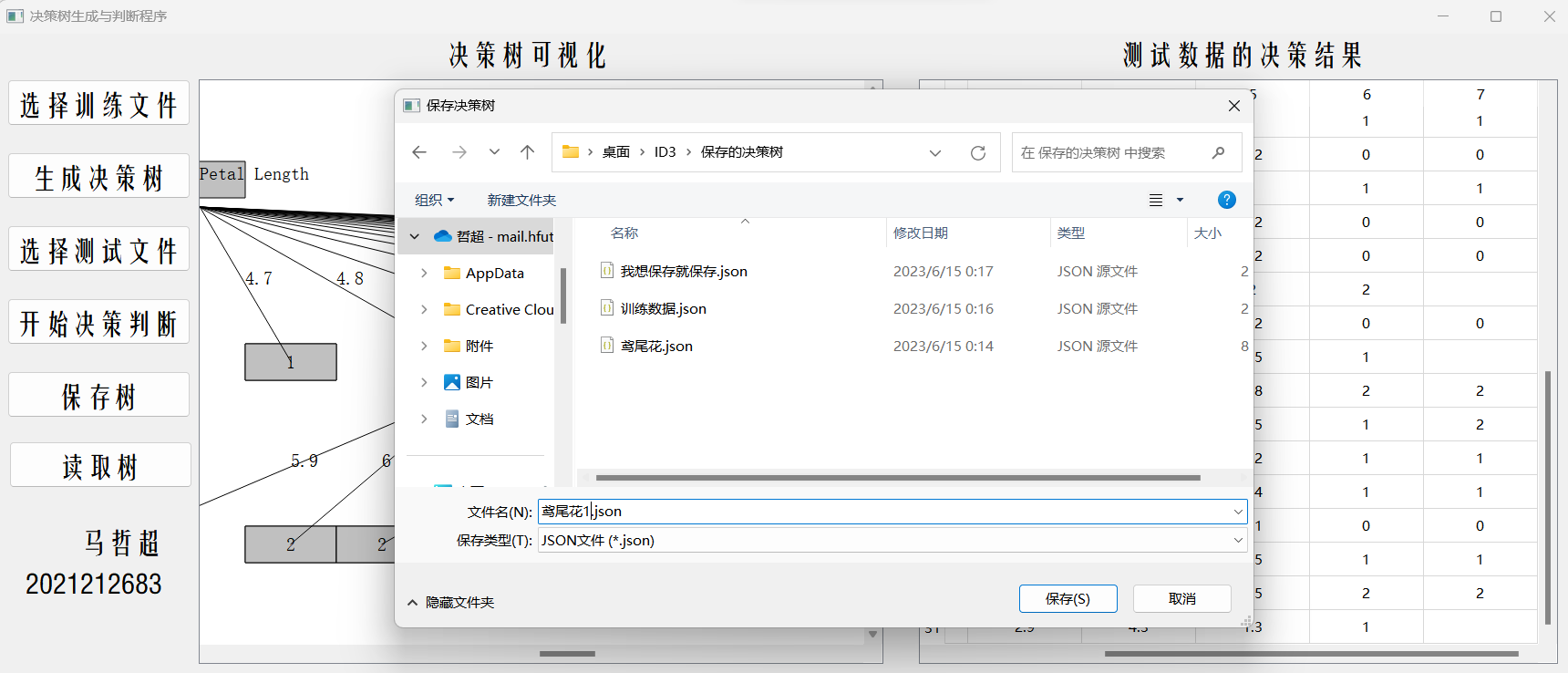


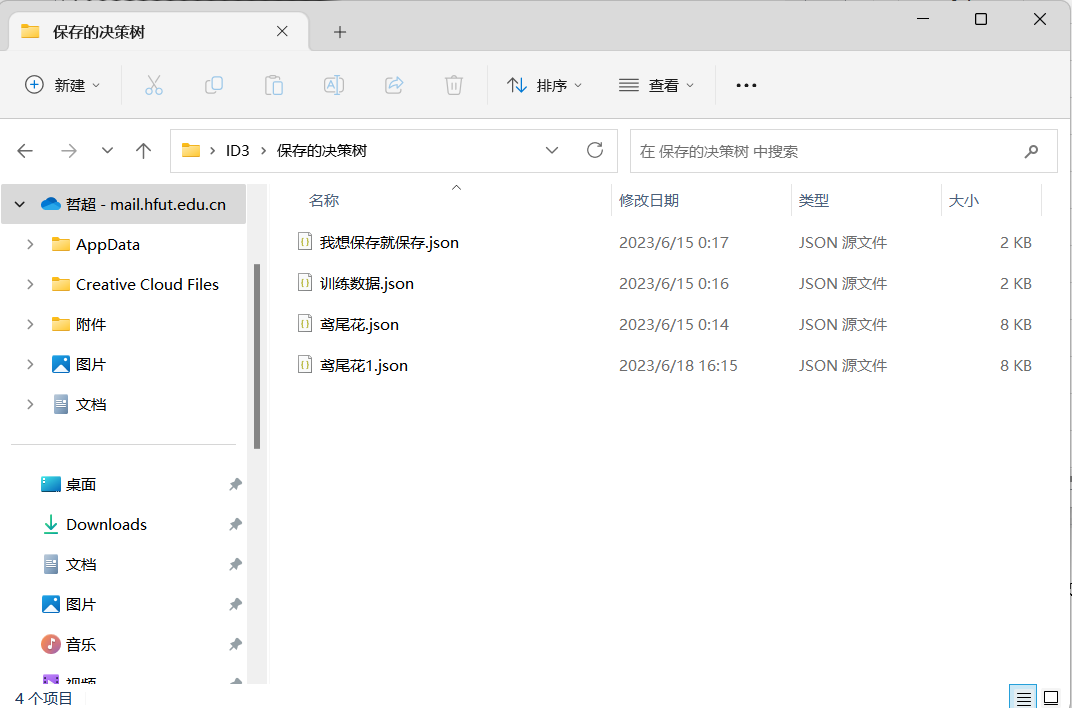


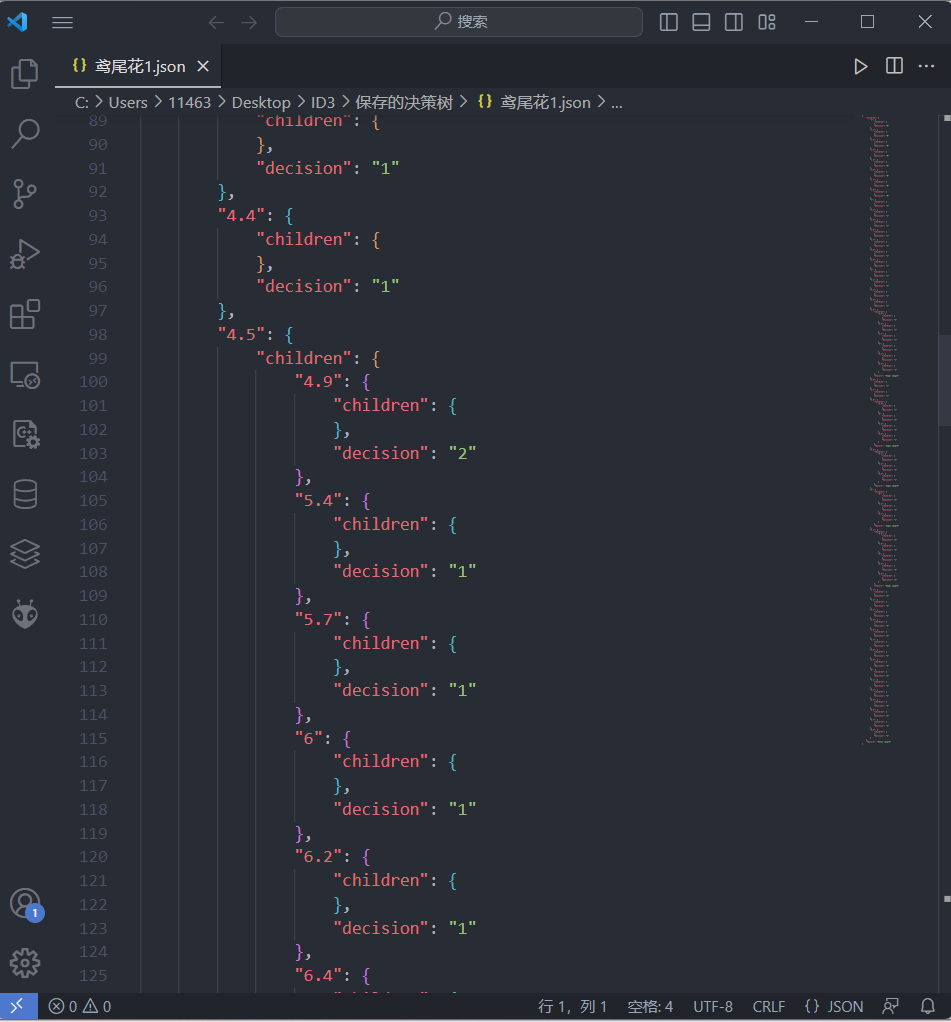


发现有部分数据无法进行判断，原因是训练数据集中没有此行数据所选择的特征值

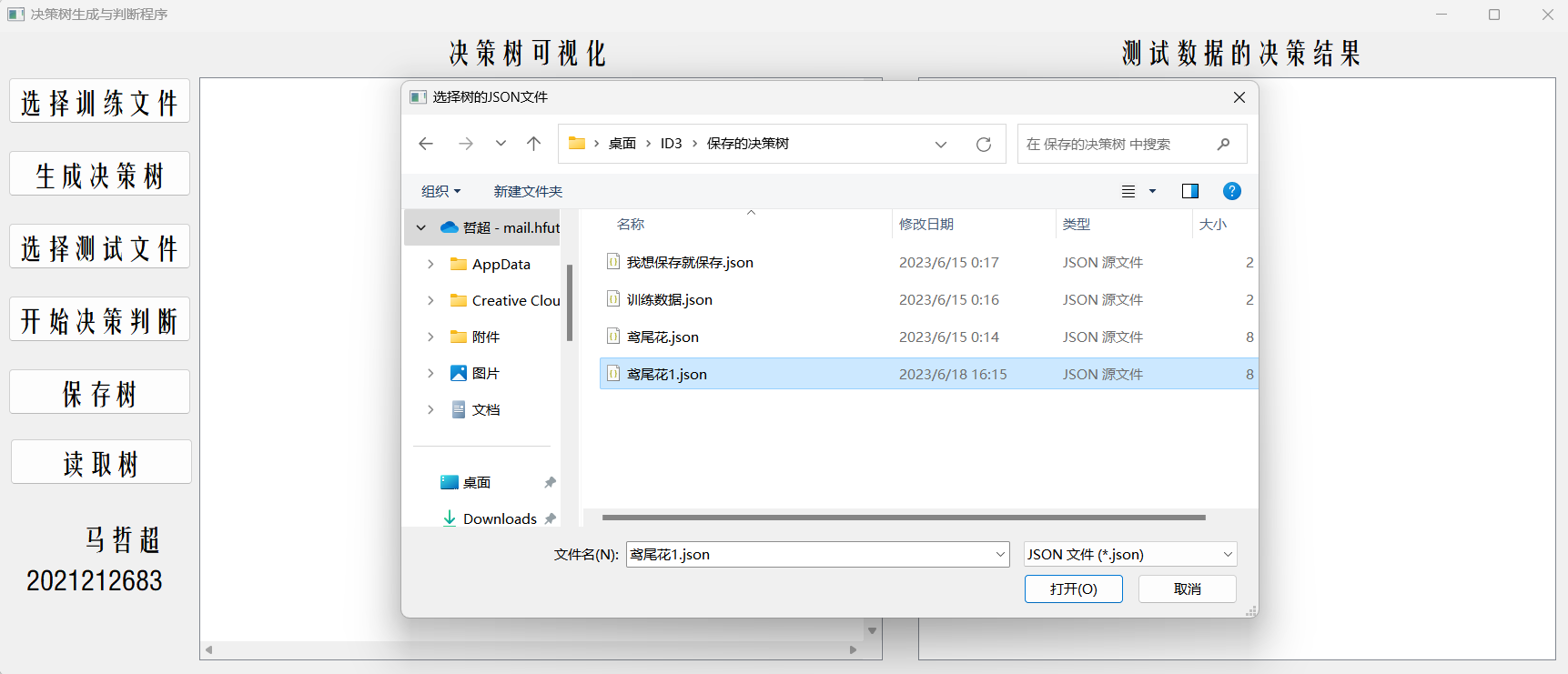
保存树：



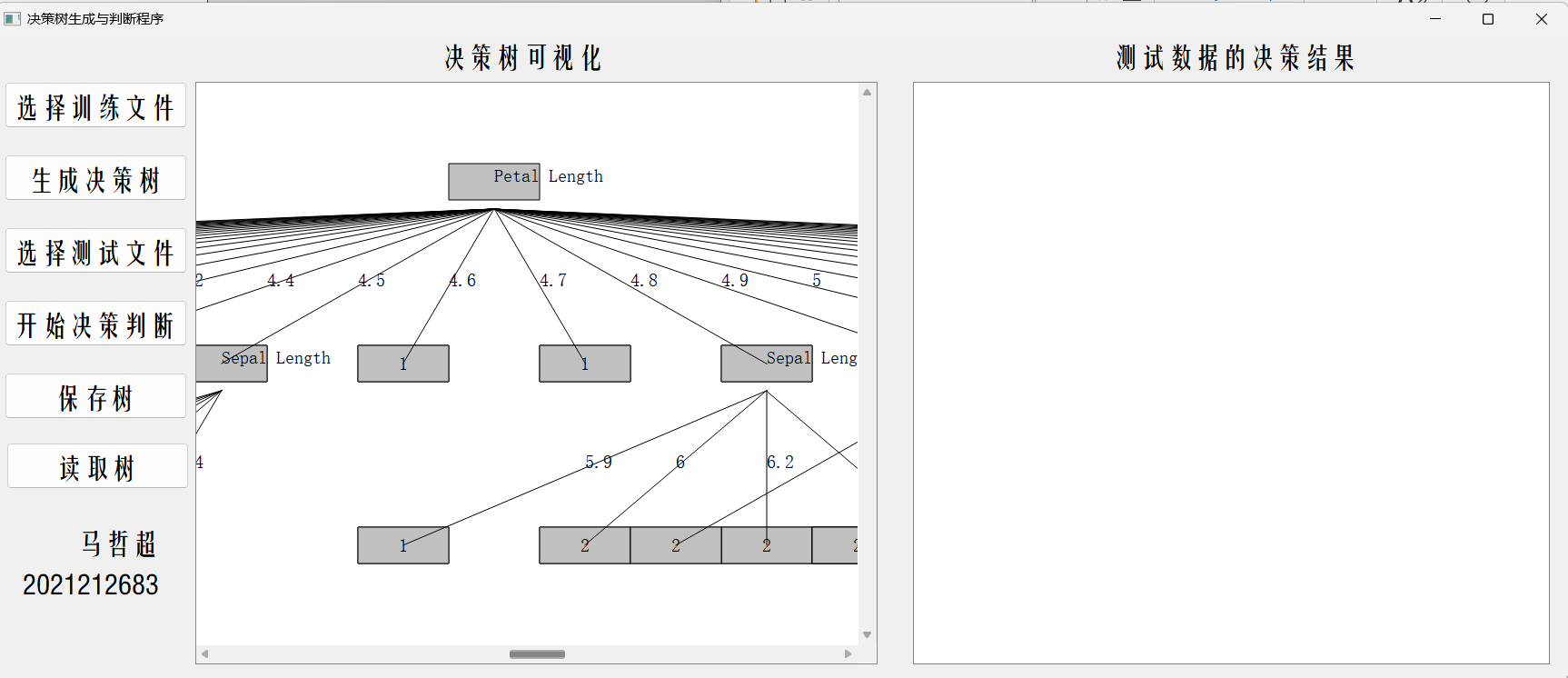




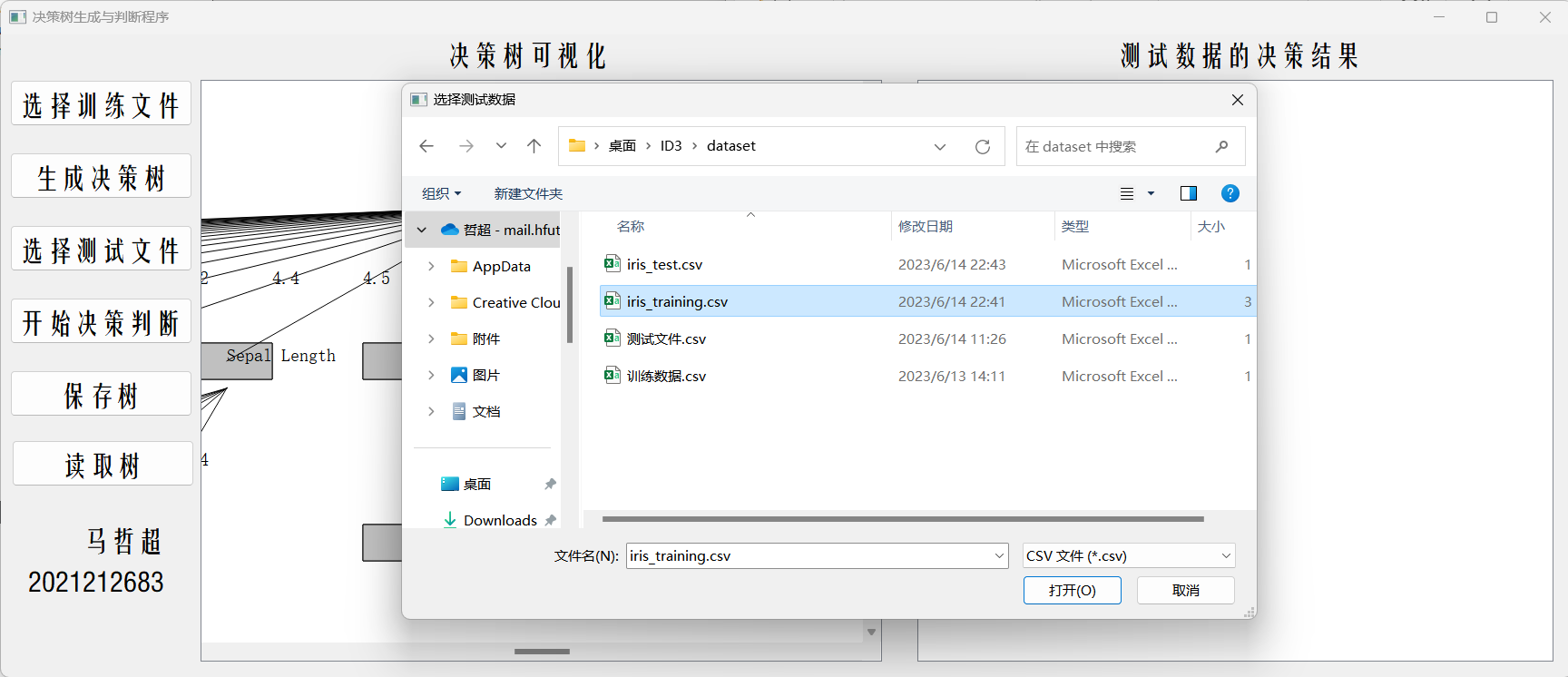
读取决策树：

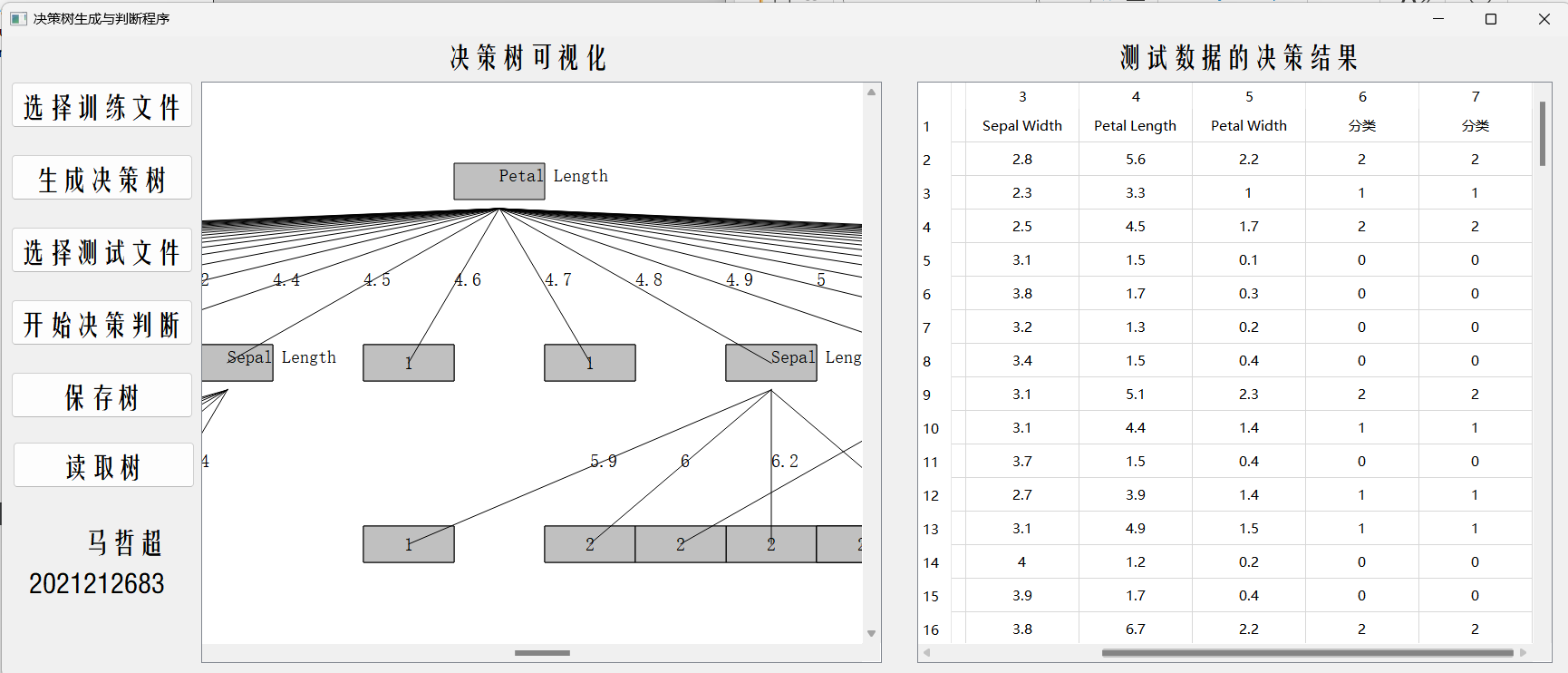


利用读取的树生成树：



进行源数据集的决策：





测试结束

**3.进一步改进**

在测试中以及用户手册中可以看出，此算法（ID3）算出来的决策树的过拟合性较为容易，泛化能力低，对于实型数据不是很好适用（需要极大数据集）

改进方法：

* 使用剪枝技术: 剪枝是一种常用的改进决策树算法的方法。它可以通过修剪决策树的一些分支和叶子节点来降低模型的复杂度，减少过拟合的风险。剪枝技术包括预剪枝和后剪枝两种方式，预剪枝在构建决策树时提前终止分裂过程，而后剪枝则先构建完整的决策树再进行修剪。
* 引入正则化方法: 正则化方法可以在构建决策树的过程中引入惩罚项，限制模型的复杂度，从而避免过拟合。例如，可以在计算信息增益时引入正则项，对属性的取值个数或分裂后的子节点个数进行惩罚，使得模型倾向于选择简单的分裂方式。
* 集成学习方法: 集成学习是通过结合多个弱分类器来构建一个强分类器的技术。可以尝试将多个决策树进行集成，例如随机森林和梯度提升树等方法，以提高泛化能力和抑制过拟合。
* 数据预处理和特征选择: 在使用决策树算法之前，进行数据预处理和特征选择可以提高算法的性能。可以考虑对数据进行归一化、标准化或离散化等处理，以及使用特征选择方法选择最具代表性的特征，减少冗余和噪声特征的影响。
* 使用其他决策树算法: ID3算法是决策树算法中的一种，还有其他改进的算法可以尝试，例如C4.5、CART等。这些算法对于过拟合和泛化能力的问题可能有更好的处理能力。

# （五）感想

这次实验对我来说是一次非常有益的学习经历。通过亲自实现和深入理解ID3算法构建决策树的过程，我对决策树算法的原理和实现细节有了更深入的认识。

在实验过程中，我发现ID3算法在处理实型数据和过拟合方面存在一些挑战。实型数据的连续特征在信息增益计算和属性选择上带来了一些困难，而且ID3算法对训练数据过于敏感，容易导致过拟合问题的发生。

为了改进算法的性能，我积极学习了一些常用的改进方法。其中，剪枝技术是一种有效的策略，可以减少决策树的复杂度，提高泛化能力。另外，正则化方法如C4.5算法和CART算法引入了属性划分的度量指标，使算法更加灵活适应不同类型的数据。此外，集成学习方法如随机森林和梯度提升树通过组合多个决策树模型，可以显著提高整体的预测性能。

通过亲自编程实现，我更深入地理解了算法的细节。从数据集的处理、信息增益的计算、属性选择，到递归构建决策树和绘制决策树的过程，我逐步掌握了决策树算法的执行流程和关键步骤。

这次实验让我深刻认识到机器学习算法的优点和局限性，并激发了我进一步探索和学习的兴趣。我相信这次实验对我的机器学习之路具有重要意义，为我打下了坚实的基础。我将继续学习和应用机器学习算法，不断提升自己在这个领域的能力。

这次实验中，我还面临了一个额外的挑战，就是将决策树可视化并利用Qt进行展示。在开始之前，我需要学习Qt库的基本知识和图形绘制的相关概念。了解了如何使用QGraphicsItem类创建自定义的图形项，并通过重写paint()函数来绘制图形，还学会了如何处理图形项的位置和拖动操作。

在实现可视化过程中，我遇到了一些困难。其中一个挑战是确定每个节点的位置，使得决策树可以清晰地展示出来。我需要考虑节点的相对位置和层次结构，以便正确绘制连线和布局。另一个挑战是将特征信息和连线特征值显示在节点上，这要求我对坐标的计算和绘制文本有一定的了解。

然而，通过面对这些挑战并努力解决它们，我获得了很多收获。首先，我学会了如何将抽象的决策树转化为可视化的图形，从而更直观地理解决策树的结构和决策过程。其次，通过与Qt库的交互，我深入了解了图形绘制的基本原理和技巧，对图形界面的开发有了更深入的理解。

在这个过程中，我也注意到了图形可视化对于解释和传达复杂算法的结果和决策是多么重要。通过可视化，我可以更好地理解决策树的运行机制，同时也能够向他人清晰地展示我的实现和结果。

总的来说，虽然在利用Qt进行可视化方面遇到了一些困难，但通过努力和学习，我成功地将决策树可视化并展示出来。这使我对算法的理解更加深入，并增加了我在图形界面开发方面的技能。这次实验为我提供了一个宝贵的机会，让我学习到了更多的知识和技能，对于今后的学习和实践将会产生积极的影响。

通过这次实验，我对决策树算法的原理、实现和改进方法有了更全面的了解。我非常感谢这个实验给予我的学习机会，并期待在未来的学习和实践中取得更好的成果！