# 可视化实验五报告

201900161140 张文浩

实验完成时间：10.30

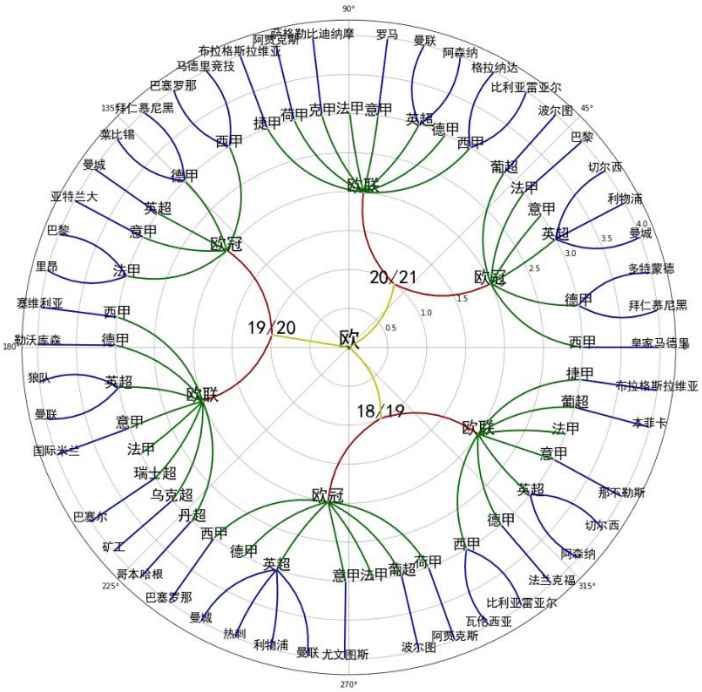
软件环境：spyder（anaconda）

## 实验要求：

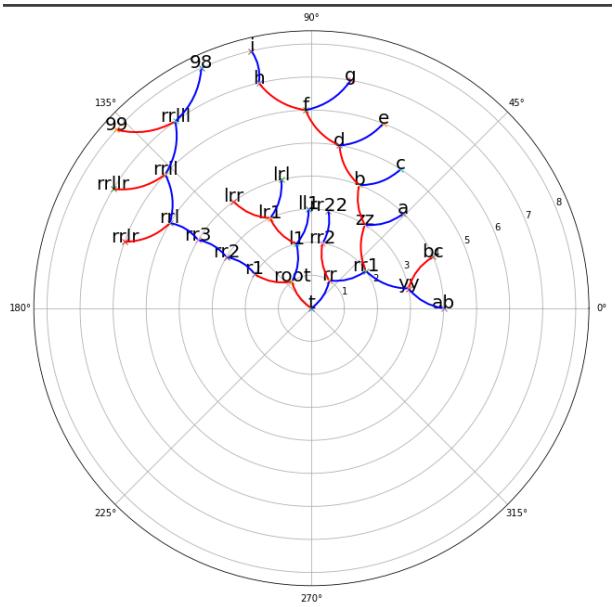
用 RT 算法实现 radial tree layout，提交代码和测试用例

## 最终效果：

n 叉树：




二叉树：




# 实验步骤：

说明：我本次实验很大程度上参考了老师发的网站 https://llimllib.github.io/pymag-trees/，实现的算法主要也是参考这篇文章提供的源代码来写的。

## 二叉树

一开始为了理解算法的思路，先根据网站上代码的思路实现了二叉树基本的构建，然后在实现的 n 叉树，下面是二叉树用到的函数的实现方法。

**1.**

定义树的类

```
class DrawTree:
    def __init__(self, tree, depth=-1):
        self.x = -1
        self.y = depth
        self.tree = tree
        self.children = []
        self.thread = None
        self.mod = 0

    def left(self):
        return self.thread or len(self.children) and self.children[0]

    def right(self):
        return self.thread or len(self.children) and self.children[-1]
```

其中 x 是最终要显示在图片上的横坐标，y 是纵坐标即深度，tree 是节点的名称，child 是子树的集合，mod 是 x 要的偏移量

**2.**

计算树的轮廓

```
def contour(left,right,max_offset=None,loffset=0,
    roffset=0,left_outer=None,right_outer=None):
    if not max_offset \
    or left.x + loffset - (right.x + roffset) > max_offset:
        max_offset = left.x + loffset - (right.x + roffset)

    if not left_outer:
        left_outer = left
    if not right_outer:
        right_outer = right

    lo = left_outer.left()
    li = left.right()
    ri = right.left()
    ro = right_outer.right()

    if li and ri:
        loffset += left.mod
        roffset += right.mod
        return contour(li, ri, max_offset, loffset, roffset, lo, ro)

    return li, ri, max_offset, loffset, roffset, left_outer, right_outer
```

我们需要做的第一件事是维护两个额外的变量，左子树上的 modifiers 的总和和右子树上的 modifiers 的总和。这些总和对于计算轮廓上每个节点的实际位置是必要的，以便我们可以检查它是否与另一侧的节点冲突。

**3.**

为了做到保证树结构尽量紧凑，所以我们需要一个函数 fix_substrees 来设置每个树节点偏移量，偏移量可以利用上一步的计算轮廓函数 contour 加以计算得到。

```
def fix_subtrees(left, right):
    li, ri, diff, loffset, roffset, lo, ro \
        = contour(left, right)
    diff += 1
    diff += (right.x + diff + left.x) % 2       #stick to the integers

    right.mod = diff
    right.x += diff
    if right.children:
        roffset += diff

    #right was deeper than left
    if ri and not li:
```

```
        lo.thread = ri
        lo.mod = roffset - loffset
    #left was deeper than right
    elif li and not ri:
        ro.thread = li
        ro.mod = loffset - roffset

    return (left.x + right.x) / 2
```

在我们运行轮廓程序后，我们将左右树之间的最大差值加 1，这样它们就不会相互冲突，如果它们之间的中点是奇数，则再添加一个。这让我们保持一个方便的测试属性 - 所有节点都有完整的 x 坐标，不会损失精度。

然后我们将右边的树向右移动规定的量。请记住，我们都将 diff 添加到 x 坐标并将其保存到 mod 值的原因是 mod 值仅适用于当前节点下方的节点。如果右子树有多个节点，我们将 diff 添加到 roffset，因为右节点的所有子节点都将向右移动那么远。

如果树的左侧比右侧深，反之亦然，我们需要设置一个线程。我们只需检查一侧的节点指针是否比另一侧的节点指针进展得更远，如果是，则将线程从较浅的树的外部设置到较深的树的内部。

为了正确处理我们之前谈到的 mod 值，我们需要在线程节点上设置一个特殊的 mod 值。由于我们已经更新了右侧偏移值以反映右侧树的向右移动，因此我们在这里需要做的就是将线程节点的 mod 值设置为更深的树的偏移量与其自身之间的差值。

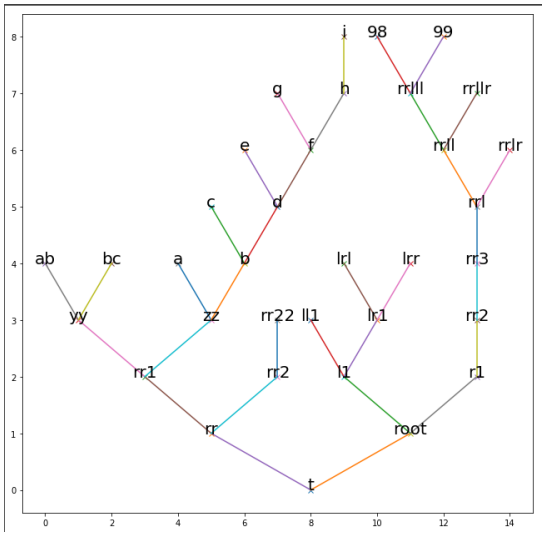
**4.**

基本的函数构建完毕，现在只需要利用 RT 算法，将它们整合起来

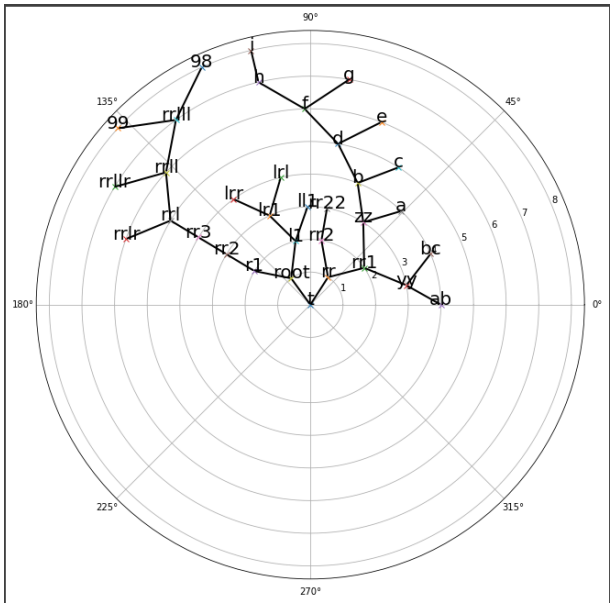
```
def layout(tree):
    dt = reingold_tilford(tree)
    return addmods(dt)

def addmods(tree, mod=0):
    tree.x += mod
    for c in tree.children:
        addmods(c, mod+tree.mod)
    return tree

def reingold_tilford(tree, depth=0):
    dt = DrawTree(tree, depth)
    if len(tree) == 0:
        dt.x = 0
        return dt

    if len(tree) == 1:
        dt.children = [reingold_tilford(tree[0], depth+1)]
        dt.x = dt.children[0].x
        return dt
```

```
    left = reingold_tilford(tree[0], depth+1)
    right = reingold_tilford(tree[1], depth+1)

    dt.children = [left, right]
    dt.x = fix_subtrees(left, right)
    return dt
```

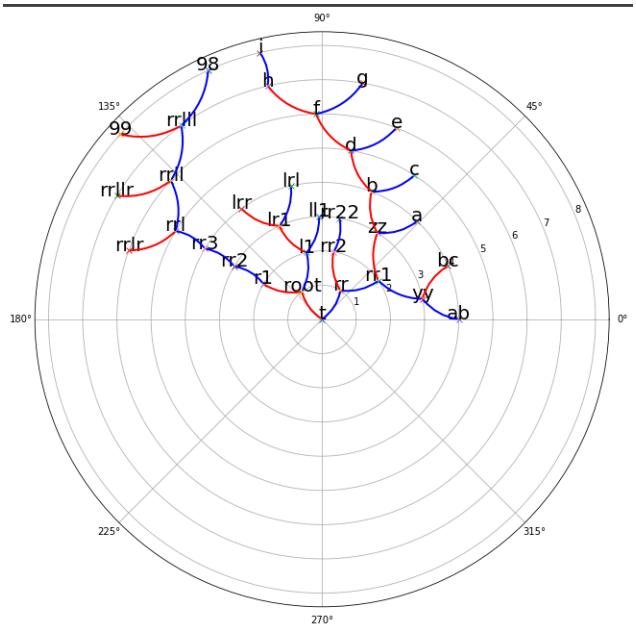
到这里我们的二叉树是这个样的：




但本次实验要求的是实现辐射型的树，所以我的想法是，将 x 转换为弧度，y 值（depth）转换为半径。




虽然现在实现了辐射型的二叉树，但不是特别美观，所以将直线换成曲线，且要区分出左右子树的曲率，再加上颜色，蓝色表示左子树，红色表示右子树。效果如下：




到这里二叉树构造完毕，下面再构造 n 叉树。

## n 叉树

现在已经得到了一个绘制二叉树的算法，考虑如何将其扩展到具有任意数量子树的树。相比于二叉树来说，n 叉树在构建时要满足一个新的原则：*父节点的子节点应该均匀分布。*

为了快速对称地绘制 n 叉树，老师给的参网站上采用了 Christoph Buchheim 等人提出的算法。

为了修改上面的算法以满足原则 6，我们需要一种方法来分隔两棵冲突的较大树之间的树。最简单的方法是，每次两棵树发生冲突时，将可用空间除以树的数量，然后移动每棵树，使其与其兄弟姐妹分开该数量。例如，在图 7 中，左右大树之间有一些距离 n，它们之间有三棵树。如果我们简单地将中间的第一棵树 n/3 与左边的树隔开，将下一棵树与左边的树隔开 n/3，依此类推，我们就会得到一棵树满足原则 6。

到目前为止，我们每次查看本文中的简单算法时，都会发现它不够用，这次也不例外。如果我们必须在每两棵冲突的树之间移动所有树，我们就有可能在算法中引入 O(n^2) 操作。

此问题的修复类似于我们之前遇到的移位问题的修复，为此我们引入了 mod。不是每次发生冲突时都将每个子树移到中间，而是保存将树移到中间所需的值，然后在放置节点的所有子节点后应用移位。

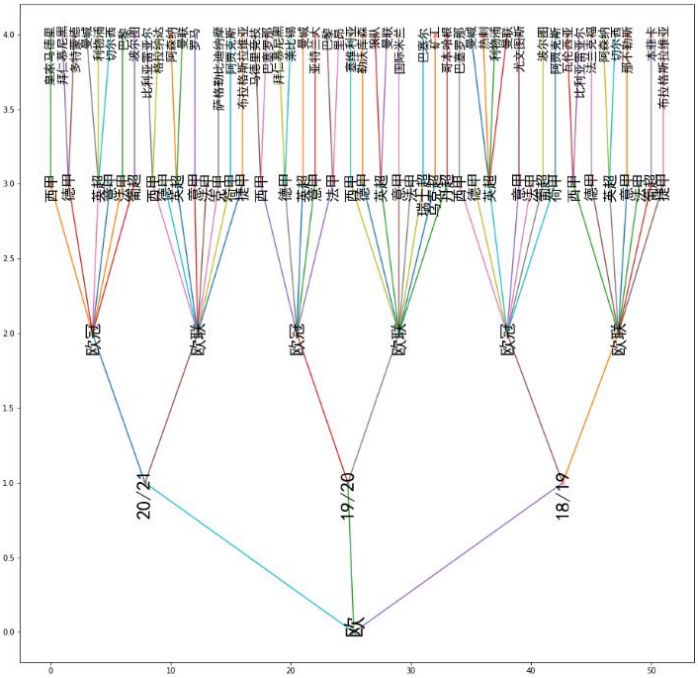
为了找出我们想要移动中间节点的正确值，我们需要能够找到冲突的两个节点之间的树数。当我们只有两棵树时，很明显，发生的任何冲突都发生在左树和右树之间。当可能有任意数量的树时，找出导致冲突的树成为一个挑战。

为了应对这一挑战，我们将引入一个 default_ancestor 变量并向我们的树数据结构中添加另一个成员，我们将其称为 ancestor. 祖先节点要么指向自身，要么指向它所属树的根。当我们需要找到一个节点属于哪棵树时，如果它被设置，我们将使用祖先成员，但回退到由指向的树 default_ancestor。

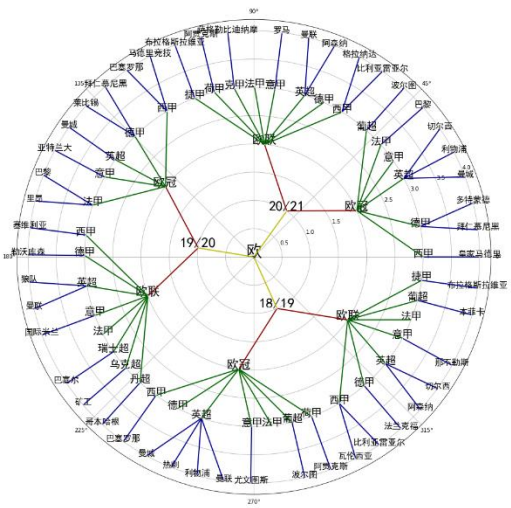
当我们放置节点的第一个子树时，我们简单地设置 default_ancestor 指向该子树，并假设由下一棵树引起的任何冲突都与第一棵树发生冲突。放置第二个子树后，我们区分两种情

况。如果第二个子树比第一个子树深，我们遍历它的右轮廓，设置祖先成员等于第二个树的根。否则，第二棵树比第一棵树大，这意味着与要放置的下一棵树的任何冲突都将与第二棵树发生冲突，因此我们只需将 default_ancestor 设置为指向它。

n 叉树的代码比较长，就不在实验报告中展示了。
非辐射型的树




接下来我们变为辐射型的树。实现方法与二叉树一样。



现在发现不够美观，我也想像二叉树一样将直线换为弧线，但是又有一个问题，二叉树可以分别设置左右子树的弧度，但 n 叉树怎么办呢。

我的解决方法是根据当前是根节点的第几个子树动态调整弧度的大小，距离中间子树越远，弧度越大，最中间子树的弧度为 0

实现函数如下：

画点：

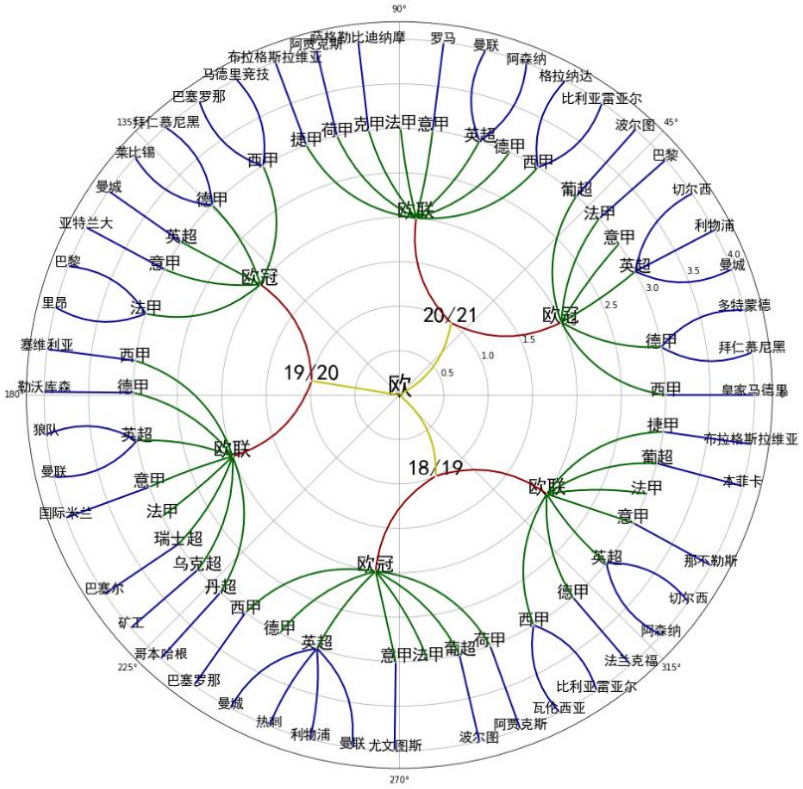
```
def drawt(root, depth):
    global r
    plt.polar(tran(root.x), depth, ',')
    plt.text(tran(root.x),depth,root.tree,fontsize=30-3.5*depth,
horizontalalignment="center",family="SimHei")
    for child in root.children:
        drawt(child, depth+1)
```

画弧线

```
def drawconn(root, depth):
    num = len(root.children)
    colors = ['y','darkred','darkgreen','darkblue']
    i=0
    for child in root.children:
        #plt.plot([tran(root.x),tran(child.x)],[depth,depth+1])
        if num==1:
            plt.annotate("",xy=(tran(root.x),depth),xytext=(tran(child.x),depth+1),
                    arrowprops=dict(color=colors[depth],
                                    arrowstyle="-",
                                    connectionstyle='arc3,rad=0',
                                    linewidth=2))
        else:
            plt.annotate("",xy=(tran(root.x),depth),xytext=(tran(child.x),depth+1),
                    arrowprops=dict(color=colors[depth],
                                    arrowstyle="-",
                                    connectionstyle='arc3,rad={}'.format((i/(num-1)-0.5)/2),
                                    linewidth=2))
        i = i + 1
        drawconn(child, depth+1)
```

另外，为了使可视化的数据更加清晰，还根据深度调整字体大小，父节点比子节点的文字大一点。

最终效果：




数据集是自己造的，表示了近三个赛季（18/19、19/20、20/21）欧冠和欧联的八强球队以及它们分别来自哪个联赛。