可视化实验六报告

201900161140 张文浩

实验时间：10.25

软件环境：vscode

**1.实验要求：**

阅读https://jheer.github.io/barnes-hut/，理解force-directed layout的算法，

构建用例跑一下d3的fore-directed layout

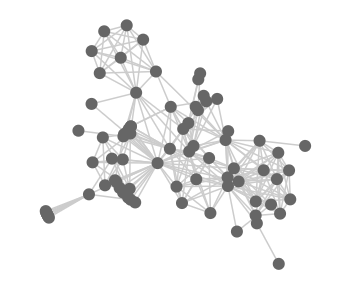
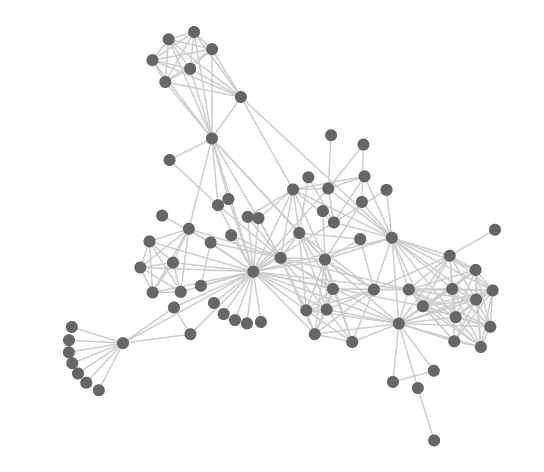
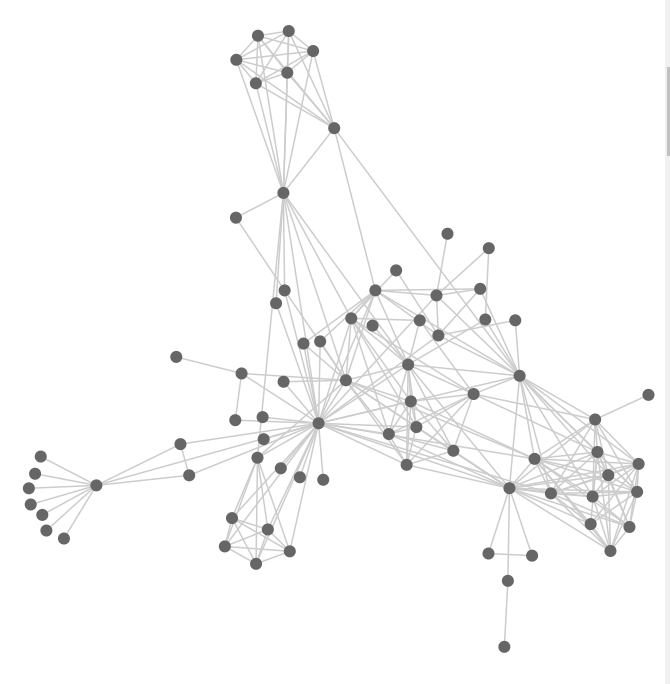
**2.实验步骤：**

**第一步：**

阅读文章<https://jheer.github.io/barnes-hut/>，理解force-directed layout的算法原理。

**force strength：**

调整force strength力的强度，力越大，代表点与点之间的引力越大。负数代表斥力。

force strength = -30 force strength = -10 force strength = 0 force strength = 10  
 

现在这种naïve的计算方法的时间复杂度为n2，随着点数n的增加，运行时间与n 2成比例地增长，无法处理大数据量。

**Barnes-Hut 近似：**

为了加速计算并使大规模模拟成为可能，天文学家 Josh Barnes 和 Piet Hut 设计了一个巧妙的方案。关键思想是通过用它们的**质心**替换一组远距离点来近似远程力。该方案显着加快了计算速度，复杂度为n log n而不是n 2。

Barnes-Hut算法包括三个步骤：

1. 构建空间索引（例如，四叉树）
2. 计算质心
3. 估计force strength

**①：**

构建四叉树，没插入一个点，就细分空间扩展树，保证每个点都在单独的一个单元格中。

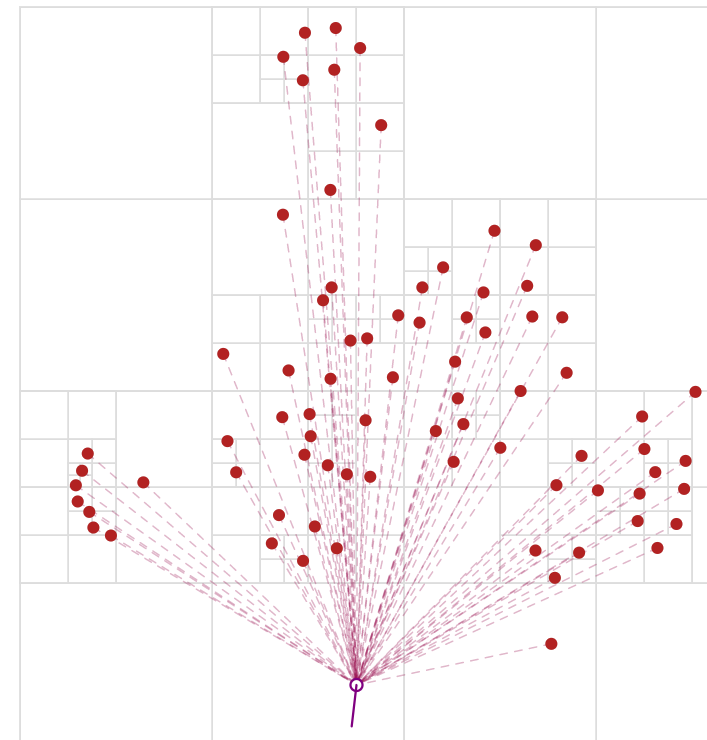
**②：**

计算质心，四叉树构建后，我们计算树的每个单元的质心。四叉树单元的质心只是其四个子单元中心的加权平均值。

我们首先访问叶节点单元格，然后访问后续的父单元格，在我们向上通过树时合并数据。遍历完成后，每个单元格都更新了其质心的位置和force strength。

**③：**

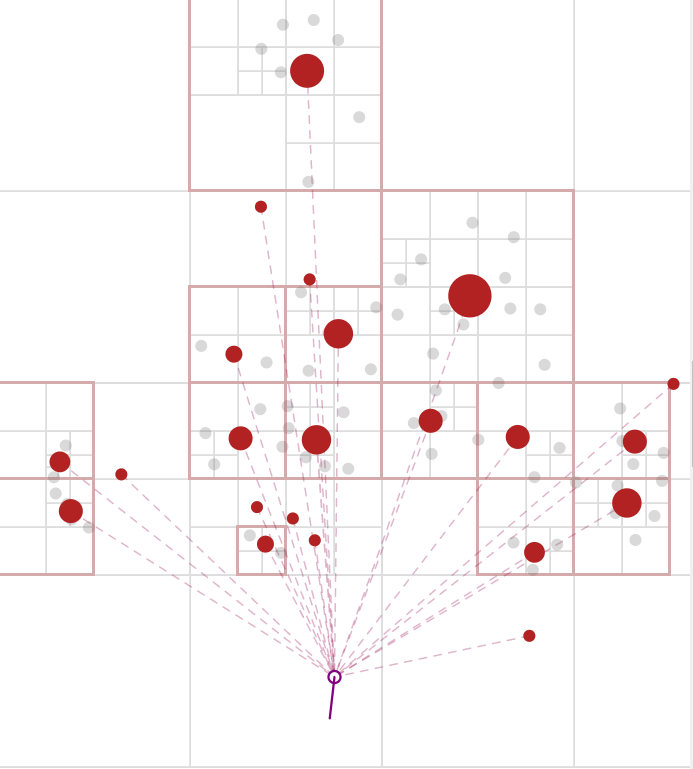
估计force，在图中加一个“探针”，计算每个每个四叉树单元对“探针”位置的force的作用。



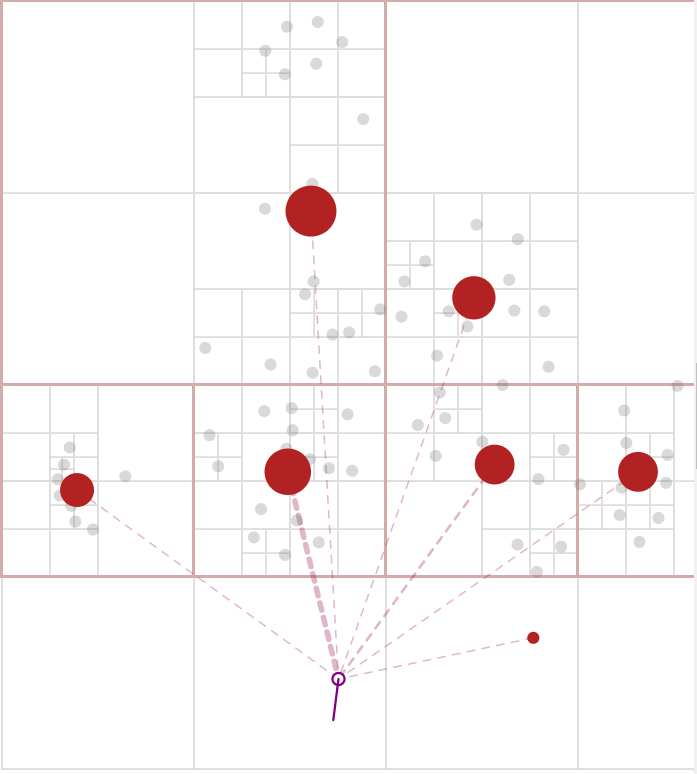
为了进一步简化计算，我们对于比较远的点，可以用更大的四叉树单元进行代替。实现的方式是设定一个阈值θ，定义为“宽度/距离”，当 theta = 1 时，如果从样本点到单元格中心的距离大于或等于单元格的宽度，则将使用四叉树单元格的质心，并忽略其内部点。

如果theta=0，说明要计算每个点到“探针”位置的force，如上图所示。

theta-0.5时，可以看到距离“探针”较远的四叉树把更大的四叉树单元看到一个整体，只需计算每个“大单元”到“探针”位置的force即可，而忽略了内部的点。如下图所示。

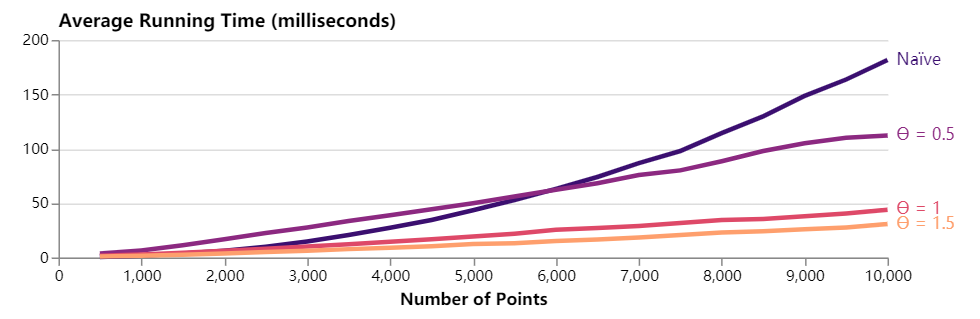


theta=1时，每个单元格变得更大了，计算量进一步减小，当然计算误差会更大没如下图所示。

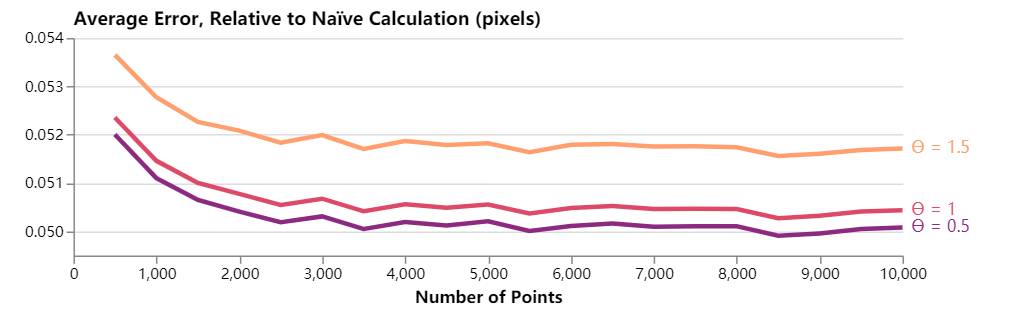


**性能分析：**

文章最后对theta不同取值的性能进行了对比分析。



从第一张图中可以看出，随着theta的增大，在计算到“探针”位置的force的时候每个被看做一个整体的四叉树大小更大，计算量更小，平均时间消耗更小。尤其是theta=1和theta=1.5时计算的平均时间都比较小，性能较好。



在第二张图中，可以看到，随着theta的增大，计算的误差也随之增大，在图一中可以看到，在时间消耗方面，theta=1和theta=1.5的表现相差不大，但theta=1.5却比theta=1的平均计算误差大了很多，所以在实践中，通常会采用theta=1来兼顾计算速度和准确率。

**第二步：**

构建用例跑一下d3的fore-directed layout

我在网上下载了一个树型结构的数据，大概长这样。



关键代码如下：

绘制画布，定义力导图

const svg = d3.select('body')

    .append('svg')

    .attr('width', width)

    .attr('height', height)

    .attr('class', 'chart')

const simulation = d3.forceSimulation(nodes)

    .force('charge', d3.forceManyBody())

    .force('link', d3.forceLink(links))

    .force('x', d3.forceX(width / 2))

    .force('y', d3.forceY(height / 2))

设置力导图属性

simulation.alphaDecay(0.05)

simulation.force('charge')

    .strength(-20)

simulation.force('link')

    .id(d => d.id) // set id getter

    .distance(0) // 连接距离，就是边的长度，但是这是一个近似值，设置为10不一定有10px

    .strength(1) // 连接强度

    .iterations(1) // 迭代次数，每次tick中模拟连接力的次数

d3.links()函数返回一个连接对象参数组，用来表示每个给定的节点对象从父节点倒子节点间的连接，建立树结构。

const root = d3.hierarchy(data)

const nodes = root.descendants()

const links = root.links()

绑定连接数据和节点数据，并设置相应属性

const simulationLinks = svg.append('g')

    .selectAll('line')

    .data(links)

    .enter()

    .append('line')

    .attr('stroke', d => '#c2c2c2')

const simulationNodes = svg.append('g')

    .attr('fill', '#fff')

    .attr('stroke', '#000')

    .attr('stroke-width', 1.5)

    .selectAll('circle')

    .data(nodes)

    .enter()

    .append('circle')

    .attr('r', 3.5)

    .attr('fill', d => d.children ? null : '#000') // 叶子节点黑底白边，父节点白底黑边

    .attr('stroke', d => d.children ? null : '#fff')

    .call(d3.drag()

        .on('start', started)

        .on('drag', dragged)

        .on('end', ended)

    )

注册tick事件处理函数，基于李布局的计算结果更新所有circle元素的位置和所有link元素位置。

simulation.on('tick', ticked)

function ticked() {

    simulationLinks.attr('x1', d => d.source.x )

        .attr('y1', d => d.source.y )

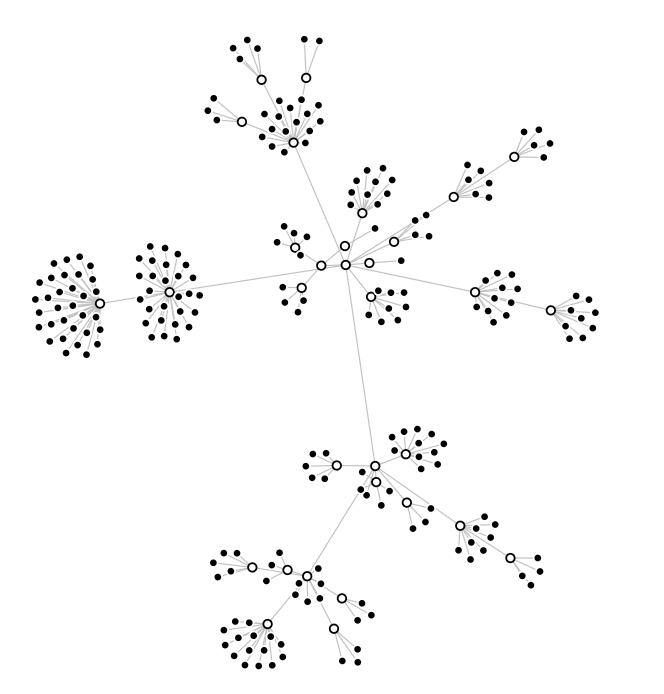
        .attr('x2', d => d.target.x )

        .attr('y2', d => d.target.y )

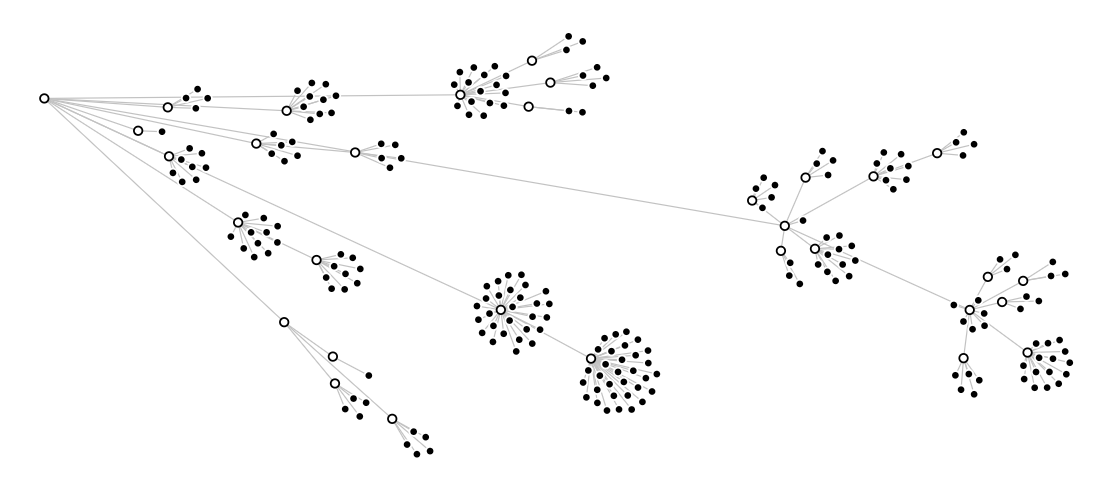
    simulationNodes.attr('cx', d => d.x )

        .attr('cy', d => d.y )

}

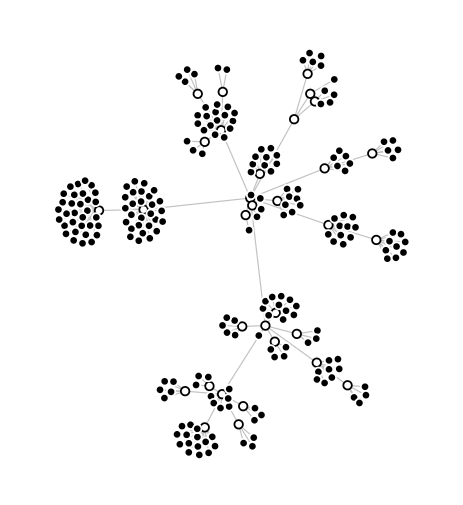
结果如下图所示：  


可以拖动点改变位置



还可以通过改变斥力的大小来改变布局

把strength强度调大后效果如下：



可以看到，在将斥力大小调小之后，每个点之间的距离变小了。