# 功能模块

[功能模块 1](#_Toc16887)

[1. 单结局因果图 1](#_Toc1826)

[1.1相关性排序 2](#_Toc26929)

[1.2单结局因果图 3](#_Toc32258)

[·计算布局 4](#_Toc24448)

[·绘制因果图 5](#_Toc10339)

[·设置画布 7](#_Toc24088)

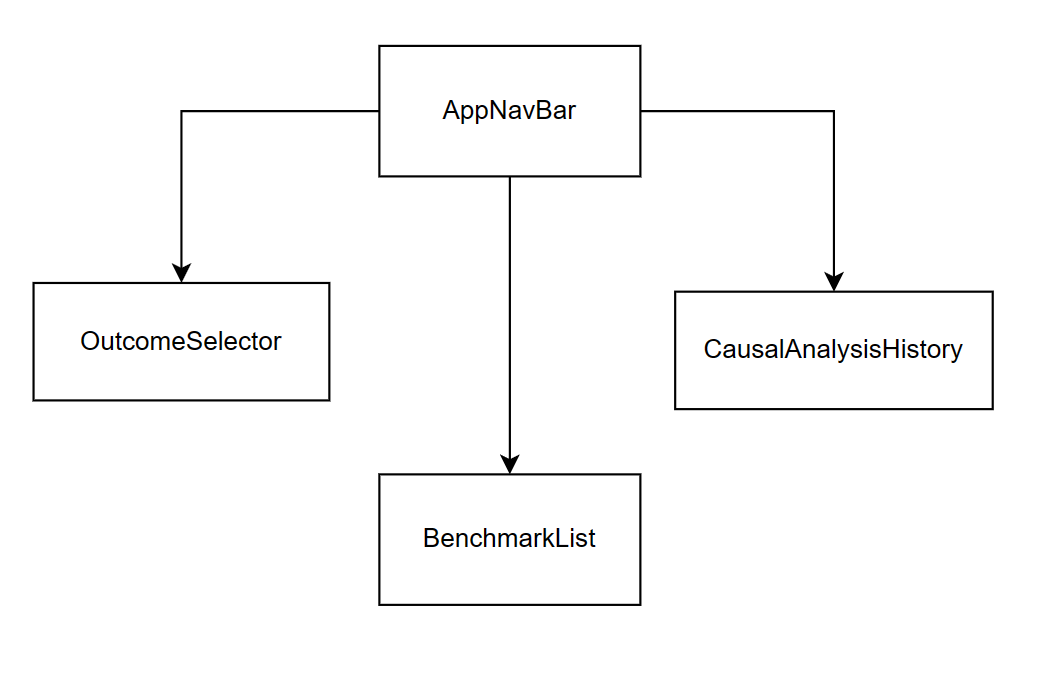
[1.3 Matrix 7](#_Toc17038)

[1.4 Benchmark 7](#_Toc8283)

[1.5历史记录表 7](#_Toc20892)

## 单结局因果图

**页面结构**



下面三个子组件都属于AppNavBar，即页面左侧操作栏

OutcomeSelector：相关性排序

CausalAnalysisHistory：历史记录表

BenmarkList：关于benchmark数据的说明信息

在AppNavBar中主要的操作是选定数据集类型：benchmark、default、clhls或ukb。AppNavBar将参数dataset传给OutcomeSelector和CausalAnalysisHistory，这两个子组件内部通过侦听器监听dataset值的变化。

### 1.1相关性排序

代码文件：Components/OutcomeSelector.vue

OutcomeSelector从浏览器localstorage中加载上一次存储的数据集类型，选定数据集类型后，OutcomeSelector通过对AppNavBar传值dataset的监听，捕捉到其值的改变，在当前页面及浏览器localstorage中记录dataset当前值（在localstorage中存储的名称为DATATYPE）。当dataset类别改变时，它负责清空当前画布内容，并根据当前数据集类型从src/plugin/variable.js中加载对应的结局列表到选择器。

清空画布内容的方法：专门有一个空白的redirectPage（url: /redirect），它通过重定向至目标url实现对应页面内容的刷新。此处，OutcomeSelector将控制路由先导航至redirectPage再重新导航至DirectedViewGraph，同时清空浏览器localstorage中供绘制因果图所存储的数据。用这种做法是因为层级关系复杂的两个页面（OutcomeSelector和DirectedViewGraph）之间使用函数相互控制（页面间的方法调用）、或者进行数据的传递（父子组件的数据传递）会较为困难，而控制路由则可无视页面关系，用于实现页面刷新的实现逻辑比较简单。

之后用户选择结局与变量个数，将调用后端接口/api/covariant获得相关性最高的n个变量。

关键方法：getOutcomeCovariant()

**请求数据**

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 举例 | 含义 |
| dataset | String | ukb | 数据集类型 |
| outcome | String | Hypertension | 结局 |
| CovariantNum | Integer | 5 | 因变量数量 |

**返回数据**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | | | 类型 | 举例 | 含义 |
| Variables\_result | nodes | id | String | Hypertension | 节点名称 |
| type | Integer | 0/1 | 是否为结局（0：outcome/  1：factor） |
| allValue | variable | Array[String] | [BMI, Sex] | 因变量列表 |
| outcome | Array[Float] | [0.41, -0.29] | 相关系数列表 |

其中nodes前n个为按相关度降序排列的factor，最后一个为用户选定的outcome。获得返回数据后，按序展示相关度最高的factors列表。整理allValue，按相关系数绝对值从大到小排序所有factors，使用Echarts绘制所有factors与outcome间相关系数的柱状统计图，正数用浅蓝色，负数用浅橙色。

根据获得的数据绘制当前因果图，需要先将数据存储在localstorage中，localstorage中存储的数据能够在浏览器会话关闭后继续保存较长时间，故而在下次打开该页面时，仍能显示出之前的绘制结果，较为方便（缺点是不能同时开两个标签页，不然数据存储会相互覆盖）。并且localstorage也避免了页面间进行数据传递的复杂逻辑。

随后由redirectPage导航至DirectedViewGraph，通过此过程刷新因果图绘制页。在DirectedViewGraph根据将数据情况判断导航至单结局因果图页面SimpleDirectedGraph还是多结局因果图页面DirectedSuperGraph。（此处因为outcome只有一个，导航至单结局因果图。

**单结局因果图数据**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | | | 类型 | 举例 | 含义 |
| GET\_JSON\_RESULT | nodesList[] | id | String | Hypertension | 节点名称 |
| type | Integer | 0/1 | 是否为结局（0：outcome/  1：factor） |
| linksList[] | source | String | BMI | 边起点 |
| target | String | Income Score | 边终点 |
| corr | Float | 0.54 | 相关系数 |
| value | Float | 0.21 | 边效应值 |
| history | | Array[] | 见历史管理 | 操作历史 |

\*以上为OutcomeSelector中存储GET\_JSON\_RESULT的数据格式，就“单结局因果图数据”而言，该表格并不完整，见后续表格补充说明。

### **1.2 PC算法因果图**

代码文件：views/SimpleDirectedGraph.vue

SimpleDirectedGraph从localstorage中加载GET\_JSON\_RESULT、DATATYPE至multipleSearchValue、dataset，两个值分别是绘制因果图所需的数据和当前数据集类型。根据数据集类型，从variable.js中加载对应的factors列表，构成页面上端VariablesCheckbox的变量多选框。根据multipleSearchValue中nodesList得到当前因果图的变量构成，并设定多选框初始默认值。

检查multipleSearchValue中是否有数据，有则计算布局，准备画因果图。此外，检查multipleSearchValue中是否有dag-gnn和hcm算法所对应的边，若有，则存储至当前页的gnnLinks和aaaiLinks，若无，则调用后端接口进行计算。

**单结局因果图数据（续）**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | | | 类型 | 举例 | 含义 |
| GET\_JSON\_RESULT | aaaiLinks[] | source | String | BMI | 边起点 |
| target | String | Income Score | 边终点 |
| value | Integer | 1 | 默认值 |
| dagLinks[] | source | String | BMI | 边起点 |
| target | String | Income Score | 边终点 |
| value | Float | 0.21 | 值（具体不知道） |

#### ·计算布局

关键方法：setGraph()

因果图绘制分为两个阶段，分别使用两个不同的库。首先调用superGraph.js的setSuperGraph方法使用dagreD3计算布局，然后保存其中所有节点坐标及边的控制点坐标至simplePos，再使用jointjs绘制最终因果图。

**simplePos数据结构**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | | | 类型 | 举例 | 含义 |
| simplePos | nodesList[] | id | String | Hypertension | 节点名称 |
| type | Integer | 0/1 | 是否为结局（0：outcome/  1：factor） |
| indexes | Array[Integer] | [4,6] | 所属组编号 |
| x | Integer | 85 | 计算所得x坐标 |
| y | Integer | 245 | 计算所得x坐标 |
| linksList[] | source | String | BMI | 边起点 |
| target | String | Income Score | 边终点 |
| corr | Float | 0.54 | 相关系数 |
| value | Float | 0.21 | 边效应值 |
| points | Array[point] | [{x: 4, y: 5}] | 所有控制点坐标 |

#### ·绘制因果图

关键方法：drawGraph()

根据点坐标的最大值、最小值，dom的宽、高，计算出绘制时的偏移量和缩放比例，调用superGraph.js的drawSuperGraph()绘制单结局因果图。该方法将在绘制超图部分进一步介绍，需要说明的是，在使用jointjs绘制的方法中，自定义了几种边的类型，以适用于不同类型的图的坐标，显示出尽量自然的曲线：

**基于d3.curveBasis**

* SuperCurve：根据未加处理的控制点坐标自然绘制的类型，只处理了首尾控制点的坐标，目的是与节点平滑连接
* ExtractedCurve：与SuperCurve类型，稍微改了参数大小
* DagGnnCurve：相比SuperCurve向右侧偏移（移动方向为首尾连接线的切向方向）
* AAAICurve：相比SuperCurve向左侧偏移（移动方向为首尾连接线的切向方向）
* TightenedCurve：压缩子图，计算压缩后的节点及控制点坐标后使用的控制曲线，调整了首尾控制点

**基于d3.curveNatural**

* TreeCurve：压缩子图并重新计算控制点后的绘制曲线，弧度明显，根据节点半径和重点箭头粗细调整了首尾两个控制点的坐标

使用该方法绘制了PC算法所得的所有边之后，若是数据中已有dag-gnn和hcm算法的边数据，将会使用 linksOperation.addLink分别绘制这两种边。

此外，还有三种特殊的边。由于可以对边进行操作（增、删、翻转），在操作后，因果图布局不会立刻改变，而是先记录了边的状态，如下：

**linksList数据结构（续）**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 类型 | 举例 | 含义 | 名称 |
| linksList[] | source | String | BMI | 边起点 |
| target | String | Income Score | 边终点 |
| value | Float | 0.21 | 边效应值 |
| reverse | Boolean | true/false | 是否翻转 |
| add | Boolean | true/false | 是否新增 |
| hidden | Boolean | true/false | 是否隐藏 |

从历史表格中选择数据后，若存在这样的边，在绘制时会对这些边作特殊处理，reverse和hidden的边按原样参与布局计算，只在使用drawSuperGraph()绘制时进行处理。而对于add新增边的情况，不会参与布局计算，没有控制点数据，不使用drawSuperGraph()绘制，而是使用 linksOperation.addLink方法，绘制一条从起点到终点的**直线**。

##### 工具方法linksOperation.addLink

代码文件：plugin/links.js

适用于各种边类型的加边方法

**传参**：(pos, link, paper, curveType, attrs)

其中，curveType即前文列举的边的类型。pos结构同simplePos，记录了之前dagreD3所得布局数据，从中寻找link对应的边的坐标数据。注意，无论边翻转与否，由source唯一确定points顺序，故而当link.source和坐标所记录的source不一致时，需要反转坐标数据中记录的控制点坐标。

在每次绘制边的时候，用以下公式重新计算了宽度：

使得宽度被映射到一个合适的范围，不至于太细看不清楚，也不至于太粗覆盖其他边和点。

在绘制时，根据边的方向确定是否有箭头，根据效应值正负确定是否为虚线，根据curveType确定边的曲线类型、颜色（AAAICurve、DagGnnCurve）、控制点（TreeCurve）。attrs包含一些不同类型的边特定的参数，比如AAAICurve、DagGnnCurve是否高亮（highlight），TreeCurve在计算控制点时所需的坐标中值（mid）、ExtractedCurve的缩放值（gap）等。设置完成后，将边加入画布即完成。

#### ·设置画布

关键方法：setPaper()

设置了因果图的多种交互效果，连接边、点的悬浮加粗、边的悬浮高亮提示和点击操作。其中，点的悬浮效果是通过修改样式完成的，而边的悬浮框是通过d3的tooltip贴士工具实现的。交互检测都是使用jointjs原生监听方法实现的，该方法封装得很好，当用户进行交互时，首先检测是否为PC算法得到的边，若是，才有交互效果。当光标移入边范围时，第一个贴士工具将在点击位置展示悬浮框，显示边的起点、终点、效应值。当光标点击边时，第二个贴士工具将显示操作菜单，显示删除和翻转两个选项，并添加监听器listener，监听页面点击，此时将保存该边信息，以供后续操作使用。

若监听器检测到点击操作，将判断点击位置，若点击选项，执行对应边操作。同时，监听器将被移除，悬浮窗关闭。

##### 边的删除

关键方法：deleteEdge()

若监听器检测到点击的是第一个删除选项，将调用deleteEdge方法，该方法在因果图数据multipleSearchValue中寻找该边，若该边为新增的边（link.add=true），直接移除该边；若不然，则将该边的hidden值设为true。这一操作上的区别，其本质原因是：所有边状态值（hidden、add、reverse）的存在，是为了记录用户操作，同时不改变图的布局，直接移除一条有坐标数据的边，会影响dagre计算布局，而没有坐标数据的边（新增的边），本身没有参与布局计算，单独进行绘制的，不会影响因果图布局，所以可以直接移除。

接着，需要在因果图操作历史multipleSearchValue.history中添加删除边操作，具体将在操作历史管理部分进行介绍。随后保存数据，并从画布中移除边。

##### 边的翻转

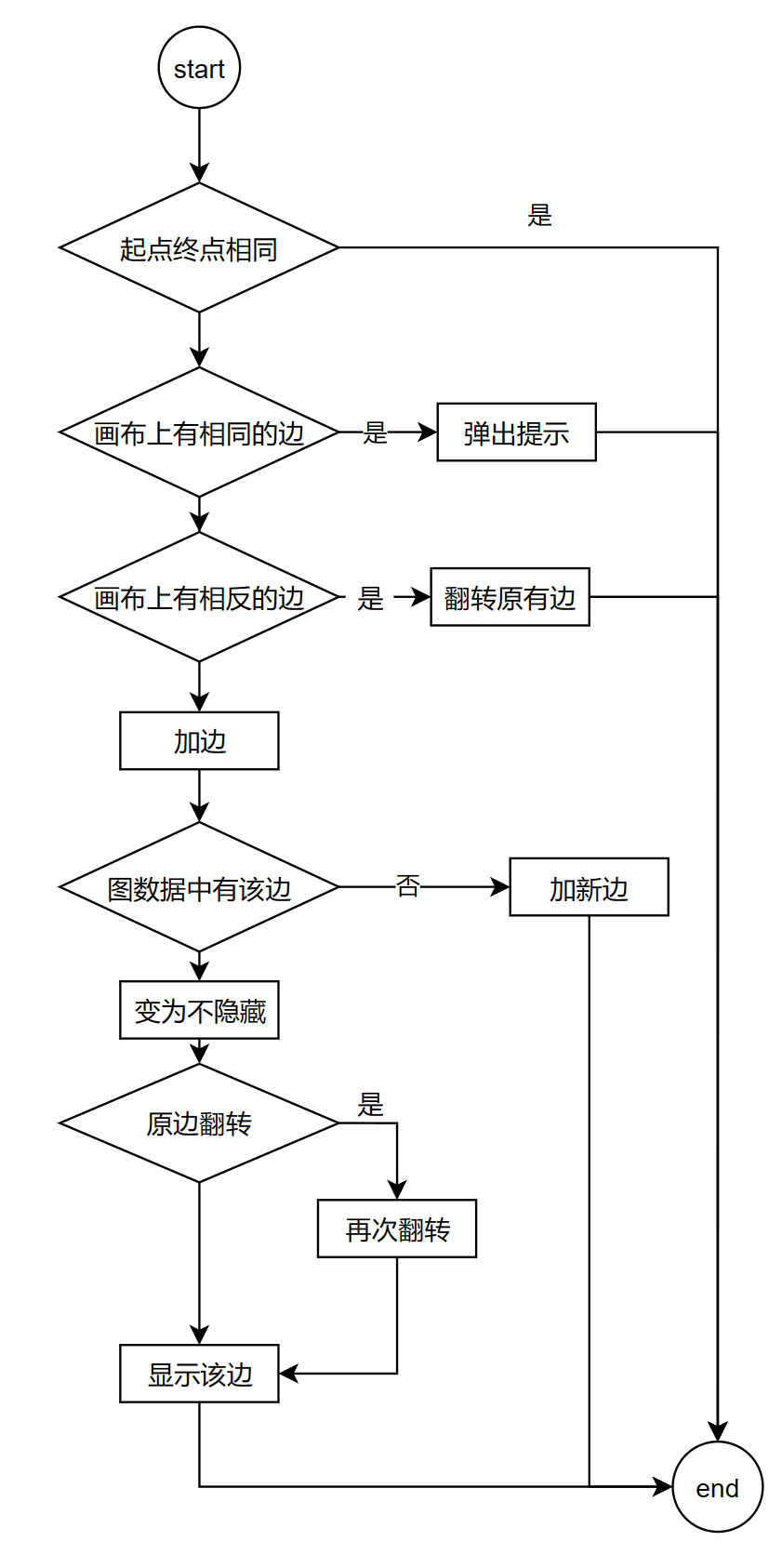
关键方法：changeEdge()

若监听器检测到点击的是第二个翻转选项，将使用changeEdge方法实现边的翻转。它将先获取到新方向的边的效应值，移除原来的边，修改multipleSearchValue中存储的该边的reverse值与value值，将翻转操作添加到因果图操作历史multipleSearchValue.history，将翻转后的边通过linksOperation.addLink添加到画布，再保存数据到localstorage。

##### 边的新增

关键方法：addTempLink()

对节点间进行自由连接的交互效果使用jointjs实现，在监听到用户加边连接上两个节点后，将有以下逻辑：



其中翻转和加边操作同样会既改变图数据multipleSearchValue又改变操作历史multipleSearchValue.history，仍然使用工具方法linksOperation.addLink加边，最后保存数据。

#### ·重新布局

关键方法：trulyDelete()

在用户对边进行了修改之后，将可以使用relayout将修改永久添加到数据中，该过程使用LinksManagement工具类的getFinalLinks方法实现。原来删除的边只是隐藏了，仍然保存了其路径数据，在multipleSearchValue仅修改了边的状态为hidden，而此时，将从multipleSearchValue中删除该条数据。翻转和新增也是同理，原先只是在数据中记录了一个状态，现在将删除该状态表示，根据真实的边数据重新计算布局。同时，若更新边之后，有的点不再与任意边连接，这样的点将被删除。

{source: A, target: B, reverse: true} => {source: B, target: A}

{source: A, target: B, add: true} => {source: A, target: B}

### **1.3 两种比较算法**

#### ·Dag-gnn

关键方法：startLoop()

**请求数据**

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 举例 | 含义 |
| dataset | String | ukb | 数据集类型 |
| nodesList | Array[String] | [Hypertension, BMI] | 所有节点名 |

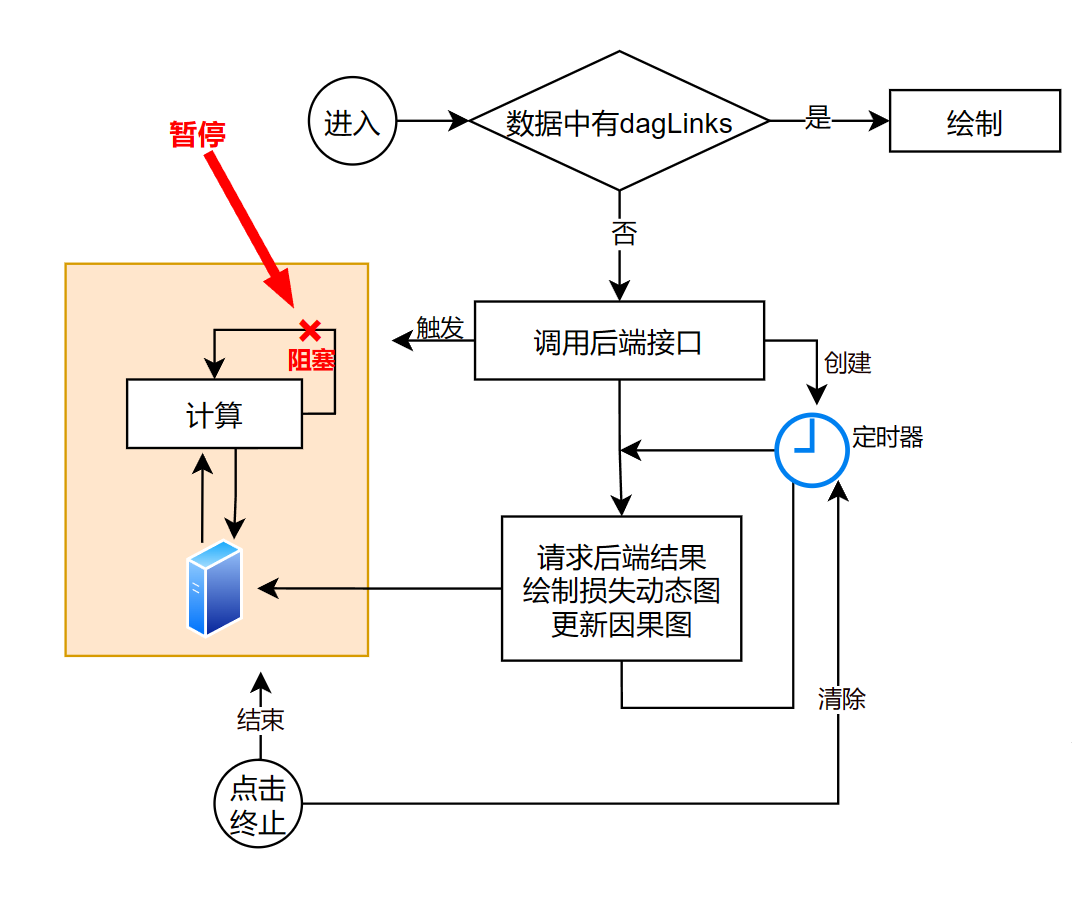
getTempResult()

**返回数据**

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 举例 | 含义 |
| graph | Array | [[0, 7.28, 4.39],  [0, 0, 0],  [0, 0, -1.25]] | 图的邻接矩阵 |
| epoch | Integer | 4 | 当前epoch序号 |
| ELBO\_loss | Float | 0.25 | 当前epoch的ELBO\_loss |
| NLL\_loss | Float | 0.12 | 当前epoch的NLL\_loss |
| MSE\_loss | Float | 0.11 | 当前epoch的MSE\_loss |

注意，后端传回的epoch值可能不是连续的。根据dag-gnn在当前数据上完成一次训练的周期长度不同，前端每隔3s询问后端时，后端epoch数值的涨速也会有所不同。比如在某组ukb数据上，每个epoch的训练时间略大于3s，所以每个epoch的损失都会被前端获取到，而在某组clhls数据上，训练速度很快，所以每隔三个epoch左右才会有一个epoch的损失被前端获取到。

daggnn接口调用的基本流程如下：



对于后端实现，采用多线程的方式。在前端请求开始后，创建并开始线程，通过全局变量stop\_Thread（布尔值）可以控制是否打破循环，结束线程。通过线程锁suspend\_lock可以控制线程的阻塞，即当前端用户点击暂停时，后端**主线程**将获取锁，而循环线程在循环中会反复进行获取锁释放锁的动作，故而当主线程获取到锁之后，循环线程无法获取锁，将被阻塞，无法继续训练。

同时，另有全局变量epoch\_loss记载了当前epoch的各项损失，best\_MSE\_graph记录当前最好（MSE loss最小）的结果。当前端通过定时器询问后端结果时，后端将返回这两个数据。前端整理边的矩阵数据并保存到gnnLinks中，通过加边方法将计算出来的所有边加入到画布中，并保存至multipleSearchValue和localstorage。同时更新页面记录的损失数据，使用echarts可绘制出动态变化的损失折线图。

终止循环时，后端将释放锁，防止循环线程被阻塞而无法结束，并设置stop\_thread为真，使得线程可以跳出循环，提前结束。

前端可通过用户按停止按钮调用停止循环接口，此外，当通过变量多选框修改变量时、离开当前页或者刷新当前页时，都会调用停止循环的接口，防止后台线程未结束而出现问题。

#### ·HCM

关键方法：getAAAI()

**请求数据**

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 举例 | 含义 |
| dataset | String | ukb | 数据集类型 |
| nodesList | Array[String] | [Hypertension, BMI] | 所有节点名 |
| skel（已弃用） | Array | [[0, 1, 1],  [0, 0, 0],  [0, 0, 1]] | PC算法所得边的邻接矩阵，1代表有边 |

**返回数据**

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 举例 | 含义 |
| graph | Array | [[0, 1, 1],  [0, 0, 0],  [0, 0, 1]] | 图的邻接矩阵（1代表有边） |

HCM原本是基于PC算出来的骨架接着计算，但是这样所得的结果与PC算法差别不大，后来改成不传初始骨架，重新算，故而请求数据中的skel不再发挥作用。HCM的后端计算是一次完成的，但需要耗费一定时间，前端用加载动画来表示。获得后端边数据后，将整理至aaaiLinks中，并保存至multipleSearchValue和localstorage，使用加边方法绘制出每条边。

在后端计算中，根据三类数据集分别提前写好了离散型变量的列表，每一次后端获取到图节点数据时，将参照该列表找出其中的离散型变量使算法能够单独处理。为了加快算法执行速度，后端只取数据集中的3000个样本进行计算。

#### ·交互

PC、Dag-gnn、HCM的边分别使用了黑色、深蓝色、橙色加以区分，在画布右上角，用标识向用户说明了三种边的颜色，该标识有悬浮和点击的交互效果，在任意名称上悬浮，将会降低另两种边的不透明度，突出选中的边。点击任意名称，会将这种边变为禁用状态，使用户在不受干扰地情况下观察对比其他类型的边。在前端实现上，就是通过监听用户动作反复调用了边的绘制和移除方法。

### **1.3 Matrix**

代码文件：components/AppMainCharacter.vue

关键方法：drawMatrix()

在画面右侧折叠窗口中，使用矩阵图表展示了变量两两间的相关关系。在前端页面组织上，SimpleDirectedGraph页面使用了element-ui的抽屉组件，在其中包含了一个子组件AppMainCharacter。该矩阵内容仅与变量列表有关，前端会通过v-if来控制变量改变时矩阵的更新。（控制v-if的布尔值由false变为true的时候，组件会根据最新的数据重新渲染）

在矩阵中，图表类型分为四种：

1. 每个变量的分组数量统计柱状图：这类图表分布在对角线上，分组逻辑预先按照数据集和变量类型设定好了（离散型变量是确定的分组，连续型变量是按最大最小值平均分组）
2. violin图：连续型与离散型变量组合的分布图，展示了连续型变量在每类分组中的分布情况。
3. bubble图：两个离散型变量组合的分布图，通过圆的大小直观体现每种组合的数量
4. 散点图：两个连续型变量组合的分布，散点绘制时透明度为0.01，散点叠加将提高不透明度，通过颜色深浅可直观反映变量分布

获取数据的过程在前端进行，通过d3读取csv文件，并从中随机选取1000个样本。二层循环遍历变量，根据变量类型绘制对应的图表。其中除柱状图使用echarts绘制以外，其余图表均使用d3绘制。

### **1.4 Benchmark**

代码文件：views/DirectedGraph/BenchmarkGraph.vue

选取的数据为[bnlearn - Bayesian Network Repository](https://www.bnlearn.com/bnrepository/clgaussian-small.html" \l "healthcare)。查看benchmark因果图的入口和选择数据集类型的多个tab并列，在页面左上角。但是选择benchmark之后可直接查看提前计算好的因果图，不能进行选择变量、选择历史数据等操作，在左侧面板会显示该数据集的一些相关信息。

前端绘图逻辑和普通的单结局因果图没有太大区别，不过不再需要访问后端接口，而是先将边数据设定好，并且所有边都按权重为1确定粗细，此外还禁止了边的操作功能。三种算法仍按颜色进行区分，同样可以让用户高亮或隐藏某种算法所得的边。

### **1.5操作历史管理**

代码文件：/plugin/history.js

对边的操作将保留在图的历史中，这是为了优先记住用户对边的修改，在每次重新计算因果图后都能再次应用这些修改。对边的管理通过从history.js文件引入的historyManage工具模块实现。在单结局因果图部分主要使用的有以下四个方法：

* reverseEdge(record, operation)

翻转边，将操作记录保存到record历史记录中

首先会在历史记录中寻找是否有完全相同的操作，如果有，该操作将不再重复录入历史中。接着，寻找历史记录中是否有反方向的操作记录，如果有，将该条记录抵消，并不再记录本次操作。如果两种记录都没有，则将本次翻转操作记入历史。

**数据格式：**{source: A, target: B, value, reverse: true}

该条数据表明用户翻转了从A到B的边，而新得到的从B到A的边的效应值为value。

* deleteEdge(record, operation)

删除边，将操作记录保存到record历史记录中

首先会在历史记录中寻找所有与该边相关的记录，并将其历史中移除。接着在操作历史中记入本条操作。

**数据格式：**{source: A, target: B, hidden: true}

* addEdge(record, operation)

加边，将操作记录保存到record历史记录中

首先会在历史记录中找到所有该边的删除操作记录，并将其从历史中移除。接着在操作历史中记入本条操作。

**数据格式：**{source: A, target: B, add: true}

* reDoHistory(data: {linksList, history})

重做操作

在改变变量列表，重新访问后端获取新的因果图后，需要根据保存的操作记录重新修改一遍边。遍历操作记录data.history，对于删除边操作，找到新因果图中的该条边并删除；对于翻转边的操作，找到新因果图中同方向的边并翻转，并把操作记录中的效应值赋给翻转后的边。对于删除和翻转操作，若是新的因果图中没有对应的边，就不做任何操作。而对于加边操作，如果因果图中没有对应的边，就加上这条边。

重做所有历史操作后，得到最终的因果图，再计算布局并绘制。

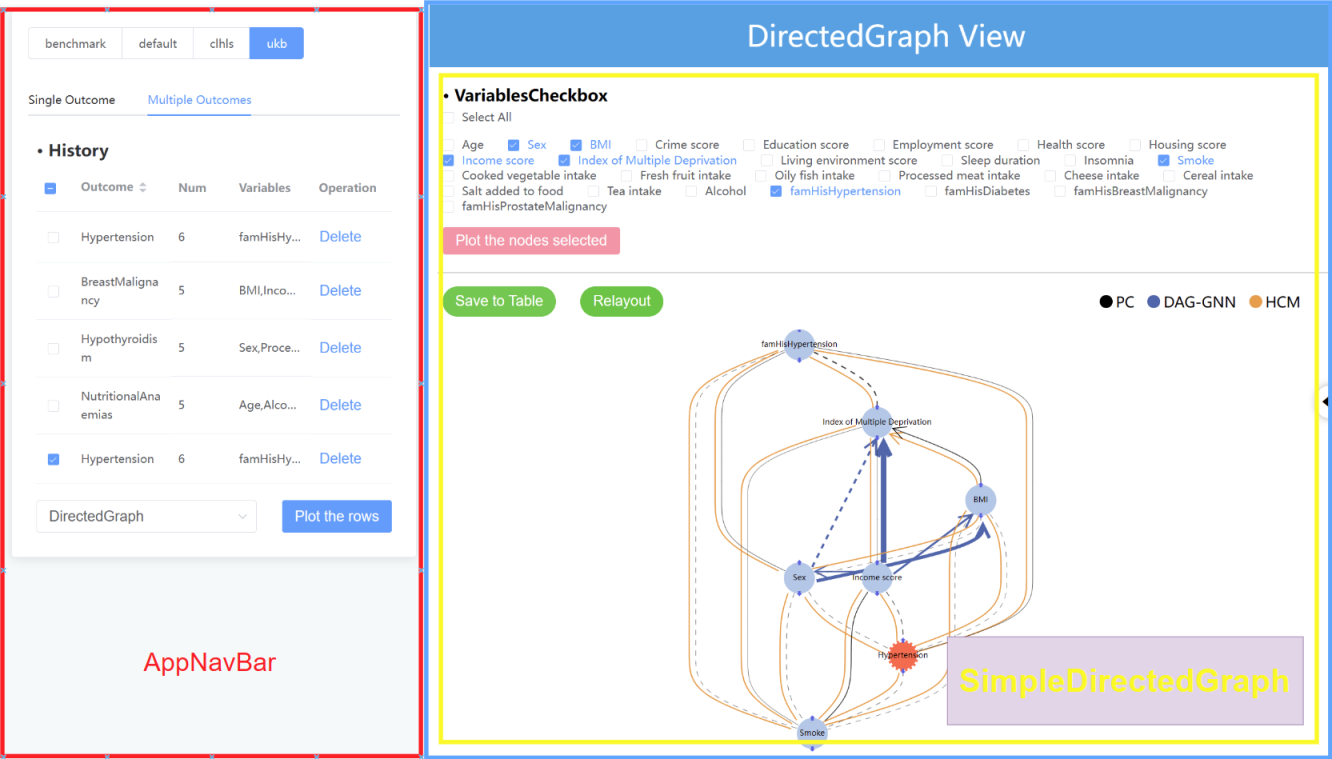
### **1.6历史记录表**

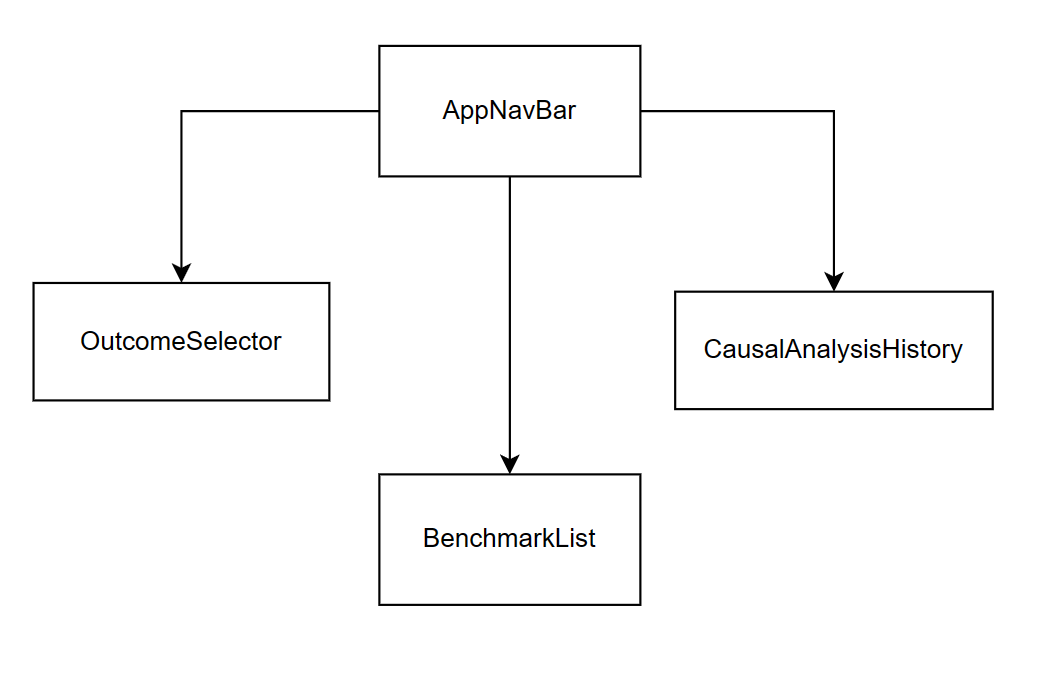
代码文件：Components/CausalAnalysisHistory.vue

关键方法：saveToTable()

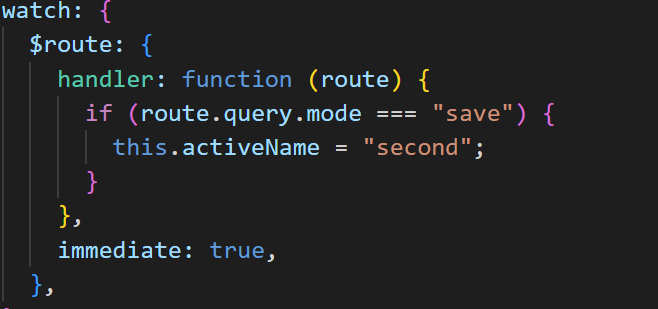
所有因果图都可以保存到历史记录表中，该表在页面左侧Multiple Outcomes为标题的tab下，与变量的相关性排序并列。

在单结局因果图的画布上方，点击save to table按钮即可将单结局因果图数据保存至历史记录表。具体而言，首先会将因果图数据multipleSearchValue保存至localStorage（存储时的数据名称为GET\_SAVE\_DATA）。然后路由导航至当前页，通过query传参mode为"save"。





这一情形与绘制因果图的页面传参相反。在绘制因果图时，由AppNavbar的子组件OutcomeSelector相关性排序或CausalAnalysisHistory历史记录表传参至SimpleDirectedGraph，采用localStorage传递数据，使用与DirectedGraphView同级路由的redirect重定向页面控制数据的刷新。



而要从SimpleDirectedGraph保存因果图数据至CausalAnalysisHistory，同样使用localStorage传参，使用路由控制组件状态，实时接收数据。但与之前相比，路由不改变url，而是用query进行传参。AppNavBar监听路由。当路由改变，并且传参mode值为save时，控制tab显示第二个页面，即历史记录表。同时，CausalAnalysisHistory自身也监听路由，当mode值为save时，将localStorage传递的数据保存至表中，并清除mode传参。在保存数据时，将只保留与结局相连通的点（不考虑方向），并只保留这些点之间连接的边。

这两种传递方式之所以会有区别，是因为本身AppNavbar所在的层级比SimpleDirectedGraph和SuperDirectedGraph要高，所以从AppNavbar向后二者传数据，只需改变当前url，就可以做到左侧栏不刷新，右侧根据数据更新。而反过来说，不能做到url改变后，右侧图不刷新，左侧操作栏数据刷新，所以面对反向的数据传递，通过相同的方式不够明智。而通过query传参，不会改变当前页面位置，又可以通过路由监听到。

**历史记录数据格式**

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 举例 | 含义 |
| Variables | Array | [BMI, Income score] | 变量列表 |
| outcome | String | Hypertension | 结局 |
| CovariantNum | Integer | 5 | 变量数量 |
| history | Array | / | 操作历史 |
| links | Array | / | PC边数据 |
| aaaiLinks | Array | / | HCM边数据 |
| dagLinks | Array | / | dag gnn边数据 |

从历史记录表中选中单条数据进行绘制时，将保存的图数据由上表格式转为本章开篇所描述的单结局因果图数据格式，保存到localstorage中，准备进行绘制。

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 类型 | 举例 | 含义 |
| nodesList | Array | [BMI, Income score] | 变量列表 |
| history | Array | / | 操作历史 |
| linksList | Array | / | PC边数据 |
| aaaiLinks | Array | / | HCM边数据 |
| dagLinks | Array | / | dag gnn边数据 |

## 多结局因果图

### **2.1历史记录表**

从历史记录中选中多条数据进行绘制时，将首先根据结局合并因果图数据。

关键方法：removeDuplicate(selections)

获得selections中所有结局的列表，遍历该列表，每次将该结局对应的所有因果图合并。首先，将这些图的history操作历史记录合并到一个数组中，不进行任何互相抵消或者修改。然后遍历所有图的边，若之前没有出现过该边则记录下来（不论方向）。同样的，遍历所有图的节点，若之前没有出现过则记录下来。得到合并的边列表后，使用LinksManagement工具类的getFinalLinks方法使原来子图中所有对边的修改发挥作用，得到真实的边数据（见1.2“计算布局”部分），然后使用historyManage模块的combineHistory方法（见本小节最后）合并子图的操作历史，并使用redo方法（见1.5操作历史管理）重做操作历史，得到最后的边数据。这样，就完成了所有同结局因果图的合并。

合并之后，遍历所有因果图选择项，合并节点到nodesList中，合并边到linksList中。如果合并后只有一个因果图，则数据格式同1.1，保存nodesList、linksList、history到localStorage中。如果有多个因果图，则保留nodesList、linksList和合并后的因果图选项selections到localStorage中，其中nodesList和linksList用于绘制超图（超图无操作历史数据），selections用于绘制子图。

随后由redirectPage导航至DirectedViewGraph，通过此过程刷新因果图绘制页。在DirectedViewGraph根据将数据情况判断导航至单结局因果图页面SimpleDirectedGraph还是多结局因果图页面DirectedSuperGraph。

**多结局因果图数据**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | | | 类型 | 举例 | 含义 |
| GET\_JSON\_RESULT | nodesList | | Array | \ | 合并后的节点  （超图节点） |
| linksList | | Array | \ | 合并后的边  （超图的边） |
| selections[] | outcome | String | Hypertension | 子因果图结局 |
| variable | Array | [BMI, Smoke] | 子因果图变量 |
| linksList | Array |  |  |

**操作历史管理（续）**

代码文件：/plugin/history.js

* combineHistory(records)

合并所有records操作历史

首先将records拼接起来，遍历拼接后的总操作历史。首先合并增删操作：寻找所有删除边的操作，针对每条删除操作，删除历史中对应边的所有删除与添加操作。比较二者数量，当删除操作较多时，删除历史中对应边的所有操作，再加上一条该边的删除记录；当添加操作较多时，删除历史中对应边的所有增删操作，最后再加上一条添加操作的记录；当二者一样多时，互相抵消，只需删除历史中对应边的所有增删操作。

接着，合并翻转操作：针对每条翻转操作，找出对应边的所有翻转操作，将方向不同的翻转相互抵消，根据数量差距，最后仅保留一条翻转操作记录。

### **2.2超图**

#### ·绘制因果图

与单结局因果相同，多结局的超图同样首先调用superGraph.js的setSuperGraph方法使用dagreD3计算布局，然后保存其中所有节点坐标及边的控制点坐标至simplePos，再使用jointjs绘制最终因果图。

遍历simplePos中所有节点，超图为所有节点定义了属性indexes，记载该节点所在的所有子因果图的编号，这样就可以在绘制时为不同子图的节点赋予不同的颜色编码。

根据点坐标的最大值、最小值，dom的宽、高，计算出绘制时的偏移量和缩放比例，调用superGraph.js的drawSuperGraph()绘制超图。

代码文件：/plugin/superGraph.js

* drawSuperGraph(dom, nodesList, linksList, scale)

绘制单结局因果图或超图

**返回值：**paper

去除linksList中所有隐藏边（被删除的边），将所有被翻转的边调整为正确的方向，并反转该边的控制点points的顺序。注意，以上调整为深拷贝，不改变原数据。

绘制部分使用jointjs提供的方法。首先，绘制节点。对于nodesList中每个节点，先绘制一个透明的圆，用来显示节点名称标签。然后根据节点indexes属性的数组长度决定每个扇形的角度，根据indexes中的索引值从预先定义的颜色列表cmap中取色，从而绘制出多个不同颜色的扇形。因为jointjs本身没有提供扇形的绘制（svg里面没有原生的扇形形状），所以采用的方法是绘制圆形，填充设为透明，利用stroke来绘制。

参考资料：[如何快速绘制任意角度的扇形？ | wangmeijian](https://wangmeijian.github.io/blog/comprehensive/create_sectorial_by_svg.html" \l "%E6%96%B9%E6%A1%88c)

在绘制单结局因果图的时候，indexes不再具有意义，将被直接指定，取cmap中最后两个颜色。

对于结局节点来说，为了体现区别，为透明形状设定白色虚线描边，覆盖在扇形上，使得节点看起来具有锯齿。对于每个绘制好的节点，使用addTool为透明形状增加控制工具，在节点上端和下端各有一个箭头形状的连接工具，可以从此牵引出一条边连接到其他节点上，该工具的实际运作过程由jointjs决定，但当监听到连接以后，仍通过自定义的linksOperation.addLink方法来实际添加边。

然后绘制边，对于linksList中每条边，计算边的宽度，根据边的方向确定是否有箭头，根据效应值正负确定是否为虚线，使用SuperCurve曲线绘制。最后将画布整体按scale缩放及平移，使得超图以合适的大小和位置展现在画面中。

最后，利用jointjs原生的监听方法，手动设置了画布的交互效果：监听鼠标滚动来进行画布缩放，监听鼠标按下抬起来控制画布的推拽。

* cmap: [

"#3182bd", //blue

"#ff7f0e", //orange

"#2ca02c", //green

"#ff9896", //pale red

"#9467bd", //purple

"#8c564b", //brown

"#e377c2", //pink

"#c7c7c7", //gray

"#bcbd22", //yellow green

"#17becf", //light blue

"#ff6c4b", // single outcome

"#aec7e8" //single factor

];

##### 边的删除

关键方法：deleteEdge()

若监听器检测到点击的是第一个删除选项，将调用deleteEdge方法，该方法将在超图数据multipleSearchValue.linksList中寻找该边，若该边为新增的边（link.add=true），还没有控制点，直接删除该边数据；若不然，则将该边的hidden值设为true。接着，遍历所有子图数据multipleSearchValue.selections，给所有子图的操作历史录入该边的删除操作，再查找子图中是否有该边。同样，若子图中有该边，且该边为新加的边，直接删除改变数据，若不然，则设置该边为hidden。

并在因果图操作历史multipleSearchValue.history中添加删除操作，具体将在操作历史管理部分进行介绍。随后保存数据，并从画布中移除边。因为jointjs使用的是记录好的布局坐标来绘制，而非严格遵从布局规范实时更新，所以移除一条边并不会对其他边或点产生影响。

**save to table\relayout**

**删除、翻转、加边**

**颜色编码**

### **2.3子图**

**original\relative\extracted**

**apply\save**

**stress(X)**