

# Neo4j 图算法

1 中心性算法 (Centrality algorithms)	3
1) 度中心性 (algo.degree)	3
2) 紧密中心性 (Closeness Centrality, algo.closeness)	3
3) 中介中心性 (Betweenness Centrality, algo.betweenness)	4
4) 网页排名 (PageRank, algo.pageRank)	5
5) 其他算法	5
2 社区发现算法 (Community detection algorithms)	5
1) The Louvain algorithm (algo.louvain)	6
2) 标签传播算法 (Label Propagation, algo.labelPropagagtion)	7
3) 连通组件算法 (Connected Components, algo.unionFind)	8
4) 强连通组件 (Strongly Connected Components, algo.scc)	8
5) 三角计数/聚类系数 (algo.triangleCount)	9
6) 平衡三角算法 (Balanced Triads, algo.balancedTriads)	10
3 路径寻找算法 (Path Finding algorithms)	10
1) 最短路径 (Shortest Path, algo.shortestPath)	10
2) 最短路径变体 A* (algo.shortestPath.astar)	12
3) 最短路径变体 Yen's K-最短路径 (algo.kShortestPaths)	13
4) 所有结对最短路径 (All Pairs Shortest Path, APSP, algo.allShortestPath)	14
5) 单源最短路径 (SSSP, algo.shortestPath.deltastepping)	15
6) 最小 (加权) 生成树算法 (Minimum Spanning Tree, MST, algo.mst)	16
7) 随机行走算法 (Random Walk, RW, algo.randomWalk)	16
4 相似度算法 (Similarity algorithms)	17
1) Jaccard 相似度 (Jaccard Similarity, algo.similarity.jaccard)	17
2) 余弦相似度 (Cosine Similarity, algo.similarity.cosine)	18
3) Pearson 相似度 (Pearson Similarity, algo.similarity.pearson)	19
4) 欧式距离 (Euclidean Distance, algo.similarity.euclidean)	20

5)重叠相似度 (Overlap Similarity, algo.similarity.overlap)	21
5 链接预测算法 (Link Prediction algorithms)	21
1)Adamic Adar (algo.linkprediction.adamicAdar)	21
2)相同邻居 (Common Neighbors, algo.linkprediction.commonNeighbors)	22
3)Preferential Attachment (algo.linkprediction.preferentialAttachment)	23
4)Resource Allocation (algo.linkprediction.resourceAllocation)	23
5)相同社区 (Same Community, algo.linkprediction.sameCommunity)	24
6)总邻居 (Total Neighbors, algo.linkprediction.totalNeighbors)	24
6 一些参考链接	25

## 1 中心性算法 (Centrality algorithms)

中心度算法主要用来判断一个图中不同节点的重要性，用于理解图中特定节点的角色及其对网络的影响。帮助我们了解群体动态，例如可信度、可访问性、事物传播的速度以及群体之间的桥梁。主要包括以下算法：度中心性 (Degree Centrality)、紧密中心性 (Closeness Centrality)、中介中心性 (Betweenness Centrality)、网页排名 (PageRank)、文档排名算法 (ArticleRank)、谐波中心性 (Harmonic Centrality)、特征向量中心性 (Eigenvector Centrality)，下面介绍几种常用算法：

### 1) 度中心性 (algo.degree)

计算节点上传入和传出关系的数量，该算法可以用于在图中查找“热” (popular) 的节点。

**使用场景：**希望通过查看传入和传出关系的数量来分析影响，或者找到单个节点的“流行度”。当关注即时连通性时，它会很好地工作。不仅仅于此，当想要评估整个图的最小程度、最大程度、平均程度和标准偏差时，度中心性也应用于全局分析。

**Neo4j 算法调用和结果：**

```
Call algo.degree.stream('User', 'FOLLOWS',
    {direction:'incoming', weightproperty: 'degree'})
yield nodeId,score
return algo.getNodeById(nodeId).id as name,score
order by score desc
```

Name	Followers
Doug	5.0
Bridget	1.0
Charles	1.0
Michael	1.0
Mark	0.0
Alice	0.0

其中，User 是节点，Name 是各个节点的属性即名字，FOLLOWS 是关系

### 2) 紧密中心性 (Closeness Centrality, algo.closeness)

度量节点与所有其他节点的距离近的程度。高紧密中心性的节点与所有其他节点的距离最短。

**算法原理：**紧密中心性算法，计算一个节点到所有其他节点的距离之和。然后将得到的和求倒数，以确定该节点的紧密性中心性得分。节点的紧密中心性用以下的公式来计算：

原始紧密中心度=1/（与此节点相连的所有节点的和）

标准化中心度=节点个数-1/（与此节点相连的所有节点的和）

**使用场景：**当需要知道哪个节点传播的东西最快时，应使用紧密中心性。在紧密中心性算法中，使用加权关系的话，会特别有助于评估交流和行为分析中的交互速度。

**Neo4j 算法调用和结果：**

```
CALL algo.closeness.stream("User", "FOLLOWS",# {improved: true})
```

```

YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id, centrality
ORDER BY centrality DESC

```

**Improved:true** 是变体算法，用于计算具有多个无相互连接的子图的紧密中心性。

**algo.closeness.harmonic** 解决不连通图的问题

结果：

algo.getNodeById(nodeId).id		centrality
"Alice"		1.0
"Doug"		1.0
"David"		1.0
...	...	
"Charles"		0.625
"Mark"		0.625

### 3) 中介中心性 (Betweenness Centrality, algo.betweenness)

指一个结点担任其它两个结点之间最短路的桥梁的次数。一个结点充当“中介”的次数越高，它的中介中心度就越大。如果要考虑标准化的问题，可以用一个结点承担最短路桥梁的次数除以所有的路径数量。

**使用场景：**适用于现实网络中的各种问题。我们使用它来发现瓶颈、控制点和漏洞。

**Neo4j 算法调用和结果：**

```

CALL algo.betweenness.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC

```

结果：

user	centrality
"Alice"	10.0
"Doug"	7.0
"Mark"	7.0
"David"	1.0
...	...
"James"	0.0

```

CALL algo.betweenness.sampled.stream('User', 'MANAGE',
{strategy:'random',probability:0.81, direction:"out"})

```

大型图采用近似算法，随机丢弃一些节点

#### 4) 网页排名 (PageRank, algo.pageRank)

指向这个网页的链接数越多，那么这个网页就越重要。

**算法原理：**  $PR(A) = (1-d) + d (PR(I)/C(I) + \dots + PR(n)/C(n))$

(1)  $PR(A)$  是页面 A 的 PR 值。

(2)  $PR(T_n)$  是页面  $T_n$  的 PR 值，在这里，页面  $T_n$  是指向 A 的所有页面中的某个页面。

(3)  $C(T_n)$  是页面  $T_n$  的出度，也就是  $T_n$  指向其他页面的边的个数。

(4)  $d$  为阻尼系数，其意义是，在任意时刻，用户到达某页面后并继续向后浏览的概率，该数值是根据上网者使用浏览器书签的平均频率估算而得，可以设置在 0 和 1 之间，通常  $d=0.85$ 。

**使用场景：** 只要你想在网络上获得广泛影响的计算，就使用这个算法。例如，如果你想寻找一个对生物功能影响最大的基因。

**Neo4j 算法调用和结果：**

```
CALL algo.pageRank.stream('User', 'FOLLOWS',
    {iterations:20, dampingFactor:0.85})
YIELD nodeId, score
RETURN algo.getNodeById(nodeId).id AS page, score
ORDER BY score DESC
```

page	score
"Doug"	1.671956494869664
"Mark"	1.5623059164267037
"Alice"	1.1116563910618424
...	...
"James"	0.15000000000000002

#### 5) 其他算法：

文档排名算法 (ArticleRank)：algo.pageRank

谐波中心性 (Harmonic Centrality)：algo.closeness

特征向量中心性 (Eigenvector Centrality)：algo.eigenvector

## 2 社区发现算法 (Community detection algorithms)

社区发现算法用来发现网络中的社区结构，也可以看做是一种聚类算法。这些被划分的社区之间关系紧密，而社区之间的关系相对稀疏。评估一个群体是如何聚集或划分的，以及其增强或分裂的趋势：社区是由紧密连接 (densely connected) 的点组成。通常来说，社区的成员在群体内的关系比在群体外的节点多，这是社区检测的一般原则。识别这些相关集体可以揭示节点群集、独立组和网络结构。此信息有助于推断对等的各组的相似行为和

偏好、弹性估算和查找嵌套关系，也可以为其他分析准备数据。主要包括以下算法：三角形计数和聚类系数 (Triangle Counting/Clustering Coefficient)，强连通组件和连通组件 (Strongly Connected Components/ Connected Components)，标签传播算法 (Label Propagation)，鲁汶模块化算法 (Louvain)

#### 1) The Louvain algorithm (algo. louvain)

基于模块度的社区发现算法，该算法相比较于普通的模块度算法，在效率和效果上都表现的比较好，并且能够发现层次性的社区结构，其优化的目标是最大化整个图结构的模块度，通过比较关系权重和标准对比的密度，逐渐最大化预设的群体精度。

模块度 (Modularity) 是一种评估社区网络划分好坏的度量指标。它的物理含义是社区内顶点的连接边数与随机情况下的边数只差，它的取值范围是  $[-0.5, 1)$ ，其定义如下

$$Q = \frac{1}{2m} \sum_{i,j} [A_{ij} - \frac{k_i k_j}{2m}] \delta(c_i, c_j)$$

$$\delta(u, v) = \begin{cases} 1 & \text{when } u=v \\ 0 & \text{else} \end{cases}$$

$A_{ij}$  表示表示顶点  $i$  和顶点  $j$  之间的权重，如果是无权图，所有的边权重可以看作是 1； $k_i = \sum_j A_{ij}$  表示与顶点  $i$  相连的所有边的权重之和； $c_i$  表示顶点  $i$  所属的社区；另外  $m$  的计算公式如下：

$$m = \frac{1}{2} \sum_{i,j} A_{ij}$$

模块度也可以理解是社区内部边的权重减去所有与社区顶点相连的边的权重和，对无向图更好理解，即社区内部边的度数减去社区内顶点的总度数。基于模块度的社区发现算法，都是以最大化模块度  $Q$  为目标。

#### 算法过程：

**step1:** 将图中的每个顶点看成一个独立的社区，初始社区的数目与顶点个数相同；

**step2:** 对每个顶点  $i$ ，尝试把顶点  $i$  分配到你邻居顶点所在的社区中，并计算分配前与分配后的模块度变化  $\Delta Q$ ，并记录  $\Delta Q$  最大的那个邻居顶点；如果最大  $\Delta Q > 0$ ，则把顶点  $i$  分配到  $\Delta Q$  最大的那个邻居顶点所在的社区，否则放弃此次划分；

**step3:** 重复 **step2**，直到所有顶点的社区不再变化；

**step4:** 对图进行压缩，将所有在同一个社区的顶点压缩成一个新的顶点，社区内顶点之间的边的权重转化为新顶点的环的权重，社区间的

边权重转化为新顶点间的边权重；具体压缩过程如下图所示：

**step5:** 重复 step1 直到整个图的模块度不再发生变化。

**使用场景：**

**Neo4j 命令及结果：**

```
call algo.louvain.stream('User','FRIEND')
YIELD nodeId,community
return algo.getNodeById(nodeId).id as name ,community
```

**结果：**

name	community
"Alice"	0
"Bridget"	0
"Charles"	1
"Doug"	1
"Mark"	1
"Michael"	0

## 2) 标签传播算法 (Label Propagation, algo.labelPropagagtion)

标签传播是机器学习中的一种常用的半监督学习 (semi-supervised) 方法，用于向未标记 (unlabeled) 样本分配标签。该算法通过将所有样本通过相似性构建一个边有权重的图，然后各个样本在其相邻的样本间进行标签传播。

**算法步骤：**

**Step1:** 对于一个未划分社区的网络结构，为每个顶点初始化一个唯一的 label，设定最大迭代次数  $m$ 。

**Step2:** 针对每一个顶点，筛选出该顶点所有邻接顶点中出现次数最多的标签（如果出现次数最多的标签有不只一个，随机挑选一个），然后将当前顶点的标签 label 修改为该标签。

**Step3:** 如果上一步中没有任何顶点的标签被修改或者迭代次数达到  $m$ （或其他收敛条件），则算法停止迭代，输出结果。否则继续 Step 2。

**应用场景：**

**Neo4j 命令和结果：**

```
CALL algo.labelPropagation.stream("User", "FOLLOW",
{direction: "OUTGOING", iterations: 10, weightProperty:'weight',
writeProperty:'partition', concurrency:4})
yield nodeId,label
return algo.getNodeById(nodeId).id as name ,label
```

**结果：**

name	label
"Alice"	5
"Bridget"	5
"Charles"	4
"Doug"	4
"Mark"	4
"Michael"	5

### 3) 连通组件算法 (Connected Components, algo.unionFind)

连接组件算法(有时称为联合查找或弱连接组件)在无向图中查找连接节点集,其中每个节点都可以从同一集中的任何其他节点访问。它不同于 SCC 算法,因为它只需要在一个方向上的节点对之间存在路径,而 SCC 需要在两个方向上都存在路径。

**使用场景:** 与 SCC 一样,连接的组件通常在分析的早期用于理解图的结构。

**Neo4j 命令和结果:**

```
CALL algo.unionFind.stream("Library", "DEPENDS_ON")
YIELD nodeId, setId
RETURN setId, collect(algo.getNodeById(nodeId)) AS libraries
ORDER BY size(libraries) DESC
```

结果:

```
+-----+-----+
| setId | libraries
+-----+-----+
| 1      | [ "six",  "pandas",  "numpy",  "python-dateutil",
|          |          "pytz",   "matplotlib", "spacy" ] |
| 9      | [ "jupyter", "jpy-console", "nbconvert",
|          |          "ipykernel", "jpy-client", "jpy-core" ] |
| 5      | [ "pyspark", "py4j" ]
+-----+-----+
```

### 4) 强连通组件 (Strongly Connected Components, algo.scc)

SCC 在有向图中查找连接的节点集,其中每个节点都可以从同一集中的任何其他节点的两个方向上访问。它的运行时操作伸缩性很好,与节点数量成比例。SCC 组中的节点不需要是直接邻居,但是在集合中的所有节点之间必须有方向路径。

**使用场景:**

使用强连接组件作为图分析的早期步骤,以了解图的结构,或确定可能需要独立调查的紧密集群。对于推荐引擎等应用程序,可以使用强连接的组件来分析组中的类似行为或关注的重点。

类似 SCC 的许多社区检测算法被用于查找集群并将其折叠为单个节点,以便进一步进行集群间分析。您还可以使用 SCC 来可视化分析的周期,例如



查找可能死锁的进程，因为每个子进程都在等待另一个成员采取操作。

#### Neo4j 命令和结果：

```
CALL algo.scc.stream("Library", "DEPENDS_ON")
YIELD nodeId, partition
RETURN partition, collect(algo.getNodeById(nodeId)) AS libraries
ORDER BY size(libraries) DESC
```

结果：

partition	libraries
0	[(:Library {id: "six"})]
1	[(:Library {id: "pandas"})]
2	[(:Library {id: "numpy"})]
3	[(:Library {id: "python-dateutil"})]
4	[(:Library {id: "pytz"})]
5	[(:Library {id: "pyspark"})]
6	[(:Library {id: "matplotlib"})]

#### 5) 三角计数/聚类系数 (algo.triangleCount)

##### 算法原理：

三角形计数确定通过图中每个节点的三角形数。三角形是由三个节点组成的集合，三个节点中的每个节点与所有其他节点都有关系。三角形计数也可以全局运行以评估我们的整体数据集。

聚类系数算法的目标是测量一个组的聚类程度与它的可能的聚类度之比值。该算法在计算中使用三角形计数，它计算现有三角形数与可能的关系的比率。最大值为 1 表示组内的每个节点都连接到其他节点。

聚类系数有两种类型：局部聚类 and 全局聚类。

局部聚类系数：一个节点的局部聚类系数是其相邻节点也之间被互相连接的可能性。这个分数的计算涉及三角形计数。

一个节点的聚类系数=将通过该节点的三角形数乘以 2，然后将其除以组中的最大关系数（始终是该节点的度数减去）。

全局聚类系数：全局聚类系数是局部聚类系数的归一化之后的和。

##### 使用场景：

当需要确定组的稳定性（相互连接的数量，代表了稳定性）或作为其他网络度量（如聚类系数）的一部分时，使用三角形计数。

聚类系数可以提供随机选择的节点被连接的概率。还可以使用它快速评估特定组或整个网络的内聚性。

#### Neo4j 命令和结果：

```
CALL algo.triangle.stream('Person','KNOWS')
YIELD nodeA,nodeB,nodeC #返回的是节点
```

```
RETURN algo.getNodeById(nodeA).id AS nodeA,
algo.getNodeById(nodeB).id AS nodeB,
algo.getNodeById(nodeC).id AS node
```

结果:

nodeA	nodeB	node
"Michael"	"Karin"	"Chris"
"Michael"	"Chris"	"Will"
"Michael"	"Will"	"Mark"
"Karin"	"Michael"	"Chris"
"Chris"	"Michael"	"Karin"
"Chris"	"Michael"	"Will"
"Will"	"Michael"	"Chris"
"Will"	"Michael"	"Mark"
"Mark"	"Michael"	"Will"

```
CALL algo.triangleCount.stream('Person', 'KNOWS', {concurrency:4})
YIELD nodeId, triangles, coefficient #返回节点的聚集系数
RETURN algo.getNodeById(nodeId).id AS name, triangles, coefficient
ORDER BY coefficient DESC
```

结果:

name	triangles	coefficient
"Karin"	1	1.0
"Mark"	1	1.0
"Chris"	2	0.6666666666666666
"Will"	2	0.6666666666666666
"Michael"	3	0.3

## 6) 平衡三角算法 (Balanced Triads, algo.balancedTriads)

## 3 路径寻找算法 (Path Finding algorithms)

图搜索 (Graph Search) 算法是用于在图上进行一般性发现或显式地搜索的算法。这些算法在图上找到路径,但没有期望这些路径是在计算意义上是最优的。

路径查找算法 (Pathfinding) 是建立在图搜索算法的基础上,它探索节点之间的路径,从一个节点开始,遍历关系,直到到达目的节点。这些算法用于识别图中的最优路径,算法可以用于诸如物流规划、最低成本呼叫或 IP 路由以及游戏模拟等用途。主要包括以下几种算法: 最短路径 (Shortest Path) 以及它的两种变体 (A\*和 Yen's K): 所有结对最短路径 (All Pairs Shortest Path, APSP) 和单源最短路径 (Single Source Shortest Path, SSSP)、最小生成树 (Minimum Spanning Tree, MST)、随机行走 (Random Walk)。

### 1) 最短路径 (Shortest Path, algo.shortestPath)

计算一对节点之间的最短 (加权) 路径。

### 算法原理：

最短路径算法首先找到从起始节点到直接连接节点的最小权重关系。它跟踪这些权重并移动到“最近”节点。然后，它执行相同的计算，只不过权重的累积是从初始节点开始算的。算法继续持续迭代，就可以评估从初始节点的一个累积权重的“波浪”，并始终选择要前进的最小加权累积路径，直到到达目标节点。

### 使用场景：

在一对节点之间找到最佳路径，衡量的指标是跃点数或者任何权重值。它可以提供关于分离程度、点之间短距离或最小扩展路径的实际答案。可以使用这个算法简单地探索特别节点之间的连接。

确定位置之间的方向。在手机地图软件（比如谷歌地图）用最短路径算法或者某些变体算法来提供驾驶导航。

发现社交网络中人与人之间的离散距离。比如，当你看到某人在社交网站上的简介时，它会显示出图中你离他之间有多少人，也会显示你们之间的共同联系人。

找到一个演员和 Kevin Bacon 所出现的电影之间的逻辑距离。

### Neo4j 命令和结果：

```
MATCH (source:Place {id: "Amsterdam"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, null)
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

结果：

place	cost
"Amsterdam"	0.0
"Immingham"	1.0
"Doncaster"	2.0
"London"	3.0

加权重最短路径：

```
MATCH (source:Place {id: "Amsterdam"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, "distance")
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

结果：

+-----+

place	cost
+-----+	
"Amsterdam"	0.0
"Den Haag"	59.0
"Hoek van Holland"	86.0
"Felixstowe"	293.0
"Ipswich"	315.0
"Colchester"	347.0
"London"	453.0
+-----+	

## 2) 最短路径变体 A\* (algo.shortestPath.astar)

A\*最短路径算法改进 Dijkstra 的算法，它更快一些，因为它在确定下一个探索路径时可用的额外信息都包含进来，将这些额外信息作为启发式函数的一部分。

### 算法原理：

$f(n) = g(n) + h(n)$ ，在这个函数中：

$g(n)$  是从起点到节点  $n$  的路径成本。

$h(n)$  是从节点  $n$  到目标节点的路径的估计成本，是在启发计算的结果。

在 Neo4j 的实现中，地理空间距离（中间点到目标点的物理空间距离）被用作启发。在我们的示例交通数据集中，我们使用每个位置的纬度和经度作为启发式函数的输入。

### 使用场景：

使用最短路径算法（无变体）将得到相同的结果，但在更复杂的数据集上，A\*算法将更快，因为它评估的路径更少。

### Neo4j 命令和结果：

```
MATCH (source:Place {id: "Den Haag"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.astar.stream(source,
destination, "distance", "latitude", "longitude")
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

参数	意义
<i>source</i>	最短路径搜索开始的节点
<i>destination</i>	最短路径搜索结束的节点
<i>distance</i>	关系属性的名称，指示一对节点之间的遍历成本。成本是两个地点之间的公里数
<i>latitude</i>	节点属性的名称，用于表示每个节点的纬度，作为地理空间启发式计算的一部分
<i>longitude</i>	节点属性的名称，用于表示每个节点的经度，作为地理空间启发式计算的一部分

结果：

place	cost
"Den Haag"	0.0
"Hoek van Holland"	27.0
"Felixstowe"	234.0
"Ipswich"	256.0
"Colchester"	288.0
"London"	394.0

### 3) 最短路径变体 Yen' s K-最短路径 (algo.kShortestPaths)

与最短路径算法相似，但它不只是在两对节点之间找到最短路径，而是计算最短路径的第二最短路径、第三最短路径等，最多可得到 k-1 种不同的路径。

**使用场景：**

该算法在寻找绝对的最短路径并非我们唯一目标时，会有助于找到替代的路径。当我们需要多个备份计划时，它会特别有用

**Neo4j 命令和结果：**

```

MATCH (start:Place {id:"Gouda"}),
      (end:Place {id:"Felixstowe"})
CALL algo.kShortestPaths.stream(start, end, 5, "distance")
      #5 要查找的最短路径的最大数目
YIELD index, nodeIds, path, costs
RETURN index,[node in algo.getNodesById(nodeIds[1..-1]) | node.id] AS
      via, educe(acc=0.0, cost in costs | acc + cost) AS totalCost

```

结果：

index	via	totalCost
-------	-----	-----------

0	["Rotterdam", "Hoek van Holland"]	265.0
1	["Den Haag", "Hoek van Holland"]	266.0
2	["Rotterdam", "Den Haag", "Hoek van Holland"]	285.0
3	["Den Haag", "Rotterdam", "Hoek van Holland"]	298.0
4	["Utrecht", "Amsterdam", "Den Haag", "Hoek van Holland"]	374.0

#### 4) 所有结对最短路径 (All Pairs Shortest Path, APSP, `algo.allShortestPath`)

APSP 是通过跟踪迭代过程中已计算的距离并在节点上进行并行优化。在计算到未遍历节点的最短路径时，可以复用这些已知距离。有些节点之间可能无法相互的连接，这意味着这些节点之间没有最短路径。算法不会返回这些节点对的距离

##### 使用场景：

当最短路由被阻塞或变得不理想时，APSP 通常用于找到所有备用路由。例如，该算法用于逻辑路由规划，以确保多样性路由的最佳多路径。当需要考虑所有或大部分节点之间的所有可能路由时，请使用 APSP。如优化城市设施的位置和货物分配。

##### Neo4j 命令和结果：

```
CALL algo.allShortestPaths.stream(null)
#null 表示无权重，加权重可用("distance")
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN algo.getNodeById(sourceNodeId).id AS source,
        algo.getNodeById(targetNodeId).id AS target,distance
ORDER BY distance DESC
LIMIT 10
```

##### 结果：

source	target	distance
"London"	"Gouda"	5.0
"Utrecht"	"Ipswich"	5.0
"London"	"Rotterdam"	5.0
"Colchester"	"Gouda"	5.0
"Utrecht"	"Colchester"	5.0
"Amsterdam"	"Colchester"	4.0
"Immingham"	"Ipswich"	4.0
"Den Haag"	"Colchester"	4.0
"Doncaster"	"Felixstowe"	4.0

"Utrecht"	"Felixstowe"	4.0	
+-----+			

## 5) 单源最短路径 (Single Source Shortest Path, SSSP, algo.shortestPath.deltastepping)

计算了从根节点到图中所有其他节点的最短（权重）路径

**算法原理：**

step1: 它始于一个根节点，与这个根节点相关所有的路径都将被测量。在图 4-9 中，我们选择了一个节点 A 作为根。

step1: 从根节点计算出权重最小的附近节点，并将这个节点选择出来，加入到树中，与此一同被加入树上的还有和这个节点相连的其他节点。在这个例子中， $d(A, D)=1$ 。

step1: 任何未访问节点中，从根节点到它的累积权重最小的节点被选择出来，被添加到树上。我们在图 4-9 中的选择是  $d(A, B)=8$ ,  $d(A, C)=5$  或者是 A-D-C 这个路径的 4,  $d(A, E)=5$ ，因此，A-D-C 被选中，C 被添加到树上。

step1: 持续按照这个进行下去，直到没有新的节点被添加进来，我们就得到了 SSSP 的最终结果。

**使用场景：**

当你需要评估从一个固定的起始点到所有其它单个节点的最佳路径时，就适合使用 SSSP。因为路由是根据一个节点到根节点的总路径权重来选择的，所以 SSSP 算法被用来确定到每个节点的最佳路径，但在所有节点都需要被访问的遍历中（比如，汉密尔顿路径），这个算法是不适用的。例如，SSSP 有助于确定用于紧急服务的主要路线

**Neo4j 命令和结果：**

```
MATCH (n:Place {id:"London"})
CALL algo.shortestPath.deltaStepping.stream(n, "distance", 1.0)
YIELD nodeId, distance
WHERE algo.isFinite(distance)
RETURN algo.getNodeById(nodeId).id AS destination, distance
ORDER BY distance
```

**结果：**

+-----+		
destination	distance	
+-----+		
"London"	0.0	
"Colchester"	106.0	
"Ipswich"	138.0	

"Felixstowe"	160.0	
"Doncaster"	277.0	
"Immingham"	351.0	
"Hoek van Holland"	367.0	
"Den Haag"	394.0	
"Rotterdam"	400.0	
"Gouda"	425.0	
"Amsterdam"	453.0	
"Utrecht"	460.0	
+-----+		

## 6) 最小（加权）生成树算法（Minimum Spanning Tree, MST, **algo.mst**）

最小（加权）生成树算法（Minimum Spanning Tree, MST）是从一个给定的节点开始，到其所有可到达的节点并使得经过这些节点的权重尽可能地小。它从已被访问的节点出发，根据最小权重访问下一个节点，而且不产生环。

### 使用场景：

当需要访问所有节点的最佳路由时，请使用最小生成树。因为路由是根据下一步的成本来选择的，所以当你必须在一次行走中访问所有节点时，它非常有用。

### Neo4j 命令和结果：

```
MATCH (n:Place {id:"Amsterdam"})
CALL algo.spanningTree.minimum("Place", "EROAD", "distance",
id(n), {write:true, writeProperty:"mst"})
# id(n)生成树应该从中开始的节点的内部节点 id
#EROAD 计算生成树时要考虑的关系类型
YIELD loadMillis, computeMillis, writeMillis,
effectiveNodeCount
RETURN loadMillis, computeMillis, writeMillis,
effectiveNodeCount
```

## 7) 随机行走算法（Random Walk, RW, **algo.randomWalk**）

该算法从一个节点开始，在某种程度上随机地跟踪一个关系向前或向后到邻居节点。然后，它从该节点执行相同的操作，依此类推，直到达到设置的路径长度。（我们说有点随机，因为一个节点和它的邻居之间的关系数量会影响一个节点被遍历的概率。

### 使用场景：

当需要生成一组大部分随机连接的节点时，可以将随机行走算法用作其他算法或数据管道的一部分

### Neo4j 命令和结果：



```

MATCH (source:Place {id: "London"})
CALL algo.randomWalk.stream(id(source), 5, 1)
    #5 随机行走的跃起数，1 要计算的随机行走数、
    # id(source)随机行走起点的内部节点 ID
YIELD nodeIds UNWIND algo.getNodesById(nodeIds) AS place
RETURN place.id AS place

```

结果：

```

+-----+
| place |
+-----+
| "London" |
| "Colchester" |
| "Ipswich" |
| "Felixstowe" |
| "Ipswich" |
| "Felixstowe" |
+-----+

```

## 4 相似度算法（Similarity algorithms）

计算节点间的相似度

### 1) Jaccard 相似度（Jaccard Similarity, algo.similarity.jaccard）

用于度量集合之间的相似度。

**算法原理：**

杰卡德相似性算法，主要用来计算样本集合之间的相似度。给定两个集合 A，B，jaccard 系数定义为 A 与 B 交集的大小与并集大小的比值。

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

**应用场景：**

比较有限样本集之间的相似性和差异性，只适用于二元数据的集合

**Neo4j 命令及结果：**

**①求单个节点的相似度：**

```

MATCH (p1:Person {name: 'Karin'})-[:LIKES]->(cuisine1)
WITH p1, collect(id(cuisine1)) AS p1Cuisine
MATCH (p2:Person {name: "Arya"})-[:LIKES]->(cuisine2)
WITH p1, p1Cuisine, p2, collect(id(cuisine2)) AS p2Cuisine
RETURN p1.name AS from,
p2.name AS to,
algo.similarity.jaccard(p1Cuisine, p2Cuisine) AS similarity

```

结果：

from	to	similarity
"Karin"	"Arya"	0.6666666666666666

## ②求所有节点的相似度

```

MATCH (p:Person)-[:LIKES]->(cuisine)
WITH {item:id(p), categories: collect(id(cuisine))} as userData
WITH collect(userData) as data
CALL algo.similarity.jaccard.stream(data)
YIELD item1, item2, count1, count2, intersection, similarity
    # intersection 是两个集合之间的重叠数
RETURN algo.getNodeById(item1).name AS from,
        algo.getNodeById(item2).name AS to, intersection,similarity
ORDER BY similarity DESC

```

结果:

from	to	intersection	similarity
"Arya"	"Karin"	2	0.6666666666666666
"Zhen"	"Michael"	2	0.6666666666666666
"Zhen"	"Praveena"	1	0.3333333333333333
"Michael"	"Karin"	1	0.25
"Praveena"	"Michael"	1	0.25
"Praveena"	"Arya"	1	0.25
"Michael"	"Arya"	1	0.2
"Zhen"	"Arya"	0	0.0
"Zhen"	"Karin"	0	0.0
"Praveena"	"Karin"	0	0.0

## 2) 余弦相似度 (Cosine Similarity, algo.similarity.cosine)

余弦相似度是 n 维空间中两个 n 维向量之间角度的余弦。它是两个向量的点积除以两个向量的长度（或幅度）的乘积。

算法原理:

$$similarity(A, B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^n A_i \times B_i}{\sqrt{\sum_{i=1}^n A_i^2} \times \sqrt{\sum_{i=1}^n B_i^2}}$$

应用场景:

我们可以使用余弦相似度算法来计算两件事之间的相似度。然后，我们可能会将计算出的相似度用作推荐查询的一部分。例如，根据获得与您看过的其他电影相似的评分的用户的偏好来获得电影推荐。

Neo4j 命令及结果:

### ①求单个节点的相似度:

```

MATCH (p1:Person {name: 'Michael'})-[likes1:LIKES]->(cuisine)
MATCH (p2:Person {name: "Arya"})-[likes2:LIKES]->(cuisine)
RETURN p1.name AS from,
        p2.name AS to,
        algo.similarity.cosine(collect(likes1.score),
                                collect(likes2.score)) AS similarity

```

结果:

from	to	similarity
"Michael"	"Arya"	0.9788908326303921

## ②求所有节点的相似度

```

MATCH (p1:Person {name: 'Michael'})-[likes1:LIKES]->(cuisine)
MATCH (p2:Person)-[likes2:LIKES]->(cuisine) WHERE p2 <> p1
RETURN p1.name AS from, p2.name AS to,
algo.similarity.cosine(collect(likes1.score),
collect(likes2.score)) AS similarity
ORDER BY similarity DESC

```

结果:

from	to	similarity
"Michael"	"Arya"	0.9788908326303921
"Michael"	"Zhen"	0.9542262139256075
"Michael"	"Praveena"	0.9429903335828894
"Michael"	"Karin"	0.8498063272285821

## 3) Pearson 相似度 (Pearson Similarity, algo.similarity.pearson)

该相关系数通过将两组数据与某一直线拟合的思想来求值，该值实际上就为该直线的斜率。其斜率的区间在 $[-1, 1]$ 之间，其绝对值的大小反映了两者相似度大小，斜率越大，相似度越大，当相似度为1时，该直线为一条对角线。

算法原理:

$$\rho_{x,y} = \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E((X - u_X)(Y - u_Y))}{\sigma_X \sigma_Y} = \frac{E(XY) - E X E Y}{\sigma_X \sigma_Y}$$

应用场景:

在计算少量集合之间的相似度时，最好使用此函数。这些过程使计算并行化，因此更适合在较大的数据集上计算相似度。

Neo4j 命令及结果:

### ①计算两节点间相似度

```

MATCH (p1:Person {name: 'Arya'})-[rated:RATED]->(movie)
WITH p1, algo.similarity.asVector(movie, rated.score) AS p1Vector
MATCH (p2:Person {name: 'Karin'})-[rated:RATED]->(movie)
WITH p1, p2, p1Vector,
algo.similarity.asVector(movie, rated.score) AS p2Vector
RETURN p1.name AS from, p2.name AS to,
algo.similarity.pearson(p1Vector, p2Vector,
{vectorType: "maps"}) AS similarity

```

结果:

from	to	similarity
"Arya"	"Karin"	0.8194651785206903

## ②计算一节点与其他节点的相似度

```
MATCH (p1:Person {name: 'Arya'})-[rated:RATED]->(movie)
WITH p1, algo.similarity.asVector(movie, rated.score) AS p1Vector
MATCH (p2:Person)-[rated:RATED]->(movie) WHERE p2 <> p1
WITH p1, p2, p1Vector,
      algo.similarity.asVector(movie, rated.score) AS p2Vector
RETURN p1.name AS from,p2.name AS to,
      algo.similarity.pearson(p1Vector, p2Vector,
                              {vectorType: "maps"}) AS similarity
ORDER BY similarity DESC
```

结果:

from	to	similarity
"Arya"	"Karin"	0.8194651785206903
"Arya"	"Zhen"	0.4839533792540704
"Arya"	"Praveena"	0.09262336892949784
"Arya"	"Michael"	-0.9551953674747637

## 4) 欧式距离 (Euclidean Distance, algo.similarity.euclidean)

欧几里德距离测量 n 维空间中两点之间的直线距离。

算法原理:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_i - q_i)^2 + \cdots + (p_n - q_n)^2}.$$

Neo4j 命令及结果:

### ①计算两个节点之间的相似性

```
MATCH (p1:Person {name: 'Zhen'})-[likes1:LIKES]->(cuisine)
MATCH (p2:Person {name: "Praveena"})-[likes2:LIKES]->(cuisine)
RETURN p1.name AS from,p2.name AS to,
      algo.similarity.euclideanDistance(collect(likes1.score),
      collect(likes2.score)) AS similarit
```

结果:

from	to	similarity
"Zhen"	"Praveena"	6.708203932499369

### ②计算一个节点与其他几个节点之间的相似性

```
MATCH (p1:Person {name: 'Zhen'})-[likes1:LIKES]->(cuisine)
MATCH (p2:Person)-[likes2:LIKES]->(cuisine) WHERE p2 <> p1
RETURN p1.name AS from,p2.name AS to,
      algo.similarity.euclideanDistance(collect(likes1.score), collect(likes2.score))
      AS similarity
ORDER BY similarity DESC
```

结果:

from	to	similarity
"Zhen"	"Praveena"	6.708203932499369
"Zhen"	"Michael"	3.605551275463989

## 5) 重叠相似度 (Overlap Similarity, algo.similarity.overlap)

重叠相似度度量在两组之间重叠。

**算法原理：**

它定义为两个集合的交集大小除以两个集合中较小者的大小。

$$O(A, B) = (|A \cap B|) / (\min(|A|, |B|))$$

**应用场景：**

我们可以使用“重叠相似度”算法来确定哪些事物是其他事物的子集。然后，我们可能会使用这些计算出的子集来从标记数据中学习分类法

**Neo4j 命令及结果：**

### ① 计算所有节点对以及他们的交集和相似度

```

MATCH (book:Book)-[:HAS_GENRE]->(genre)
WITH {item:id(genre), categories: collect(id(book))} as userData
WITH collect(userData) as data
CALL algo.similarity.overlap.stream(data)
YIELD item1, item2, count1, count2, intersection, similarity
RETURN algo.asNode(item1).name AS from,
       algo.asNode(item2).name AS to,
       count1, count2, intersection, similarity
ORDER BY similarity DESC

```

**结果：**

from	to	count1	count2	intersection	similarity
Fantasy	Science Fiction	3	4	3	1.0
Dystopia	Science Fiction	2	4	2	1.0
Dystopia	Classics	2	4	2	1.0
Science Fiction	Classics	4	4	3	0.75
Fantasy	Classics	3	4	2	0.66
Dystopia	Fantasy	2	3	1	0.5

## 5 链接预测算法 (Link Prediction algorithms)

链接预测是图数据挖掘中的一个重要问题。链接预测旨在预测图中丢失的边，或者未来可能会出现的边。这些算法主要用于判断相邻的两个节点之间的亲密程度。通常亲密度越大的节点之间的亲密分值越高。

### 1) Adamic Adar (algo.linkprediction.adamicAdar)

Adamic Adar 是一种用于基于节点的共享邻居来计算节点的紧密度的度量，但除了共同邻居，还根据共同邻居的节点的度给每个节点赋予一个权重，即度的对数分之一，然后把每个节点的所有共同邻居的权重值相加，其和作为该节点对的相似度值。

**算法原理：**

$$A(x, y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{\log |N(u)|}$$

其中  $N(u)$  是与节点  $u$  相邻的节点集合。 $A(x, y)$  为 0 表明节点  $x$  和  $y$  不接近，该值越高表明两个节点间的亲密度越大。

Neo4j 命令及结果：

①计算两节点的 **Adamic Adar score**

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.adamicAdar(p1, p2) AS score
```

结果：

score  
1.4426950408889634

②基于某一关系的两节点的 **Adamic Adar score**

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.adamicAdar(p1, p2,
    {relationshipQuery: "FRIENDS"}) AS score
```

结果：

score  
0.0

## 2) 相同邻居 (Common Neighbors, algo.linkprediction.commonNeighbors)

共同邻居捕捉到这样的想法：与没有共同朋友的人相比，有共同朋友的两个陌生人更容易被介绍。

算法原理：

$$CN(x, y) = |N(x) \cap N(y)|$$

其中  $N(x)$  是与节点  $x$  相邻的节点集合，而  $N(y)$  是与节点  $y$  相邻的节点集合。值为 0 表示两个节点不靠近，而较高的值表示节点更靠近。

Neo4j 命令及结果：

①计算两节点的相同邻居数

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.commonNeighbors(p1, p2) AS score
```

结果：

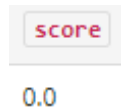
score  
1.0

②基于某一关系的两节点的相同邻居数

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.commonNeighbors(p1, p2,
```

{relationshipQuery: "FRIENDS"}) AS score

结果:



score  
0.0

### 3) Preferential Attachment(algo.linkprediction.preferentialAttachment)

优先附件是一种用于基于节点的共享邻居来计算节点的紧密度的度量。优先连接意味着节点之间的连接越多，接收新链接的可能性就越大。

算法原理:

$$PA(x, y) = |N(x)| * |N(y)|$$

其中 N(u) 是与 u 相邻的节点的集合。值为 0 表示两个节点不靠近，而较高的值表示节点更靠近。

Neo4j 命令及结果:

#### ①计算两节点的相同邻居数

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.preferentialAttachment(p1, p2) AS score
```

结果:



score  
6.0

#### ②基于某一关系的两节点的相同邻居数

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.preferentialAttachment(p1, p2,
{relationshipQuery: "FRIENDS"}) AS score
```

结果:



score  
1.0

### 4) Resource Allocation (algo.linkprediction.resourceAllocation)

资源分配是一种用于基于节点的共享邻居来计算节点的紧密度的度量。

算法原理:

$$RA(x, y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{|N(u)|}$$

其中 N(u) 是与 u 相邻的一组节点. 0 表示两个节点不靠近，而较高的值表示节点更靠近。

应用场景:

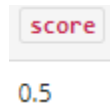
Neo4j 命令及结果:

#### ①计算两节点的相同邻居数

```
MATCH (p1:Person {name: 'Michael'})
```

```
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.resourceAllocation(p1, p2) AS score
```

结果：

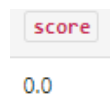


score  
0.5

## ②基于某一关系的两节点的同邻居数

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.resourceAllocation(p1, p2,
{relationshipQuery: "FRIENDS"}) AS score
```

结果：



score  
0.0

## 5) 相同社区 (Same Community, algo.linkprediction.sameCommunity)

同一社区是一种确定两个节点是否属于同一社区的方法

**算法原理：**

如果两个节点属于同一个社区，则将来（如果尚未存在）之间存在关联的可能性更大。值为 0 表示两个节点不在同一个社区中。 值 1 表示两个节点在同一社区中。

**Neo4j 命令及结果：**

### ①计算两节点是否属于相同社区

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Zhen'})
RETURN algo.linkprediction.sameCommunity(p1, p2) AS score
```

结果： score=1 表示属于一个社区， score=0 表示不属于同一社区



score  
1.0

### ②基于某一关系的两节点属于相同社区

```
MATCH (p1:Person {name: 'Arya'})
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.sameCommunity(p1, p2, 'partition') AS score
#从社区属性读取社区 partition
```

结果：



score  
1.0

## 6) 总邻居 (Total Neighbors, algo.linkprediction.totalNeighbors)

根据节点具有的唯一邻居的数量来计算节点的接近程度，它基于这样的思想，即节点之间的连接越多，接收新链接的可能性就越大。



算法原理:

$$TN(x, y) = |N(x) \cup N(y)|$$

其中  $N(x)$  是与  $x$  相邻的节点的集合,  $N(y)$  是与  $y$  相邻的节点的集合。值 0 表示两个节点不靠近, 而值越高表示节点越靠近。

Neo4j 命令及结果:

① 计算两节点的总邻居数

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.totalNeighbors(p1, p2) AS score
```

结果:

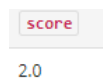


4.0

② 基于某一关系的两节点的总邻居数

```
MATCH (p1:Person {name: 'Michael'})
MATCH (p2:Person {name: 'Karin'})
RETURN algo.linkprediction.totalNeighbors(p1, p2,
    {relationshipQuery: "FRIENDS"}) AS score
```

结果:



2.0

## 6 一些参考链接

Neo4j: [https://neo4j.com.cn/user/feng\\_neo4j/topics](https://neo4j.com.cn/user/feng_neo4j/topics)

Neo4j Cypher 查询语言详解: <http://neo4j.com.cn/topic/5818519dcdf6c5bf145675c7>

Neo4j 图算法 <https://neo4j.com/docs/graph-algorithms/current/introduction/>