图算法调研

目录

[图算法表格汇总 2](#_Toc128388143)

[图遍历类算法 4](#_Toc128388144)

[广度优先遍历算法BFS 4](#_Toc128388145)

[单源最短路径算法SSSP 6](#_Toc128388146)

[强连通分量算法： 10](#_Toc128388147)

[PageRank 13](#_Toc128388148)

[三角形计数 16](#_Toc128388149)

[图挖掘类算法 16](#_Toc128388150)

[K-clique listing算法 16](#_Toc128388151)

[Subgraph listing算法 17](#_Toc128388152)

[K-motif counting算法 18](#_Toc128388153)

[K-frequent subgraph mining算法 19](#_Toc128388154)

[图学习类算法 20](#_Toc128388155)

[图卷积神经网络算法 21](#_Toc128388156)

[GraphSAGE算法 22](#_Toc128388157)

[GIN算法 23](#_Toc128388158)

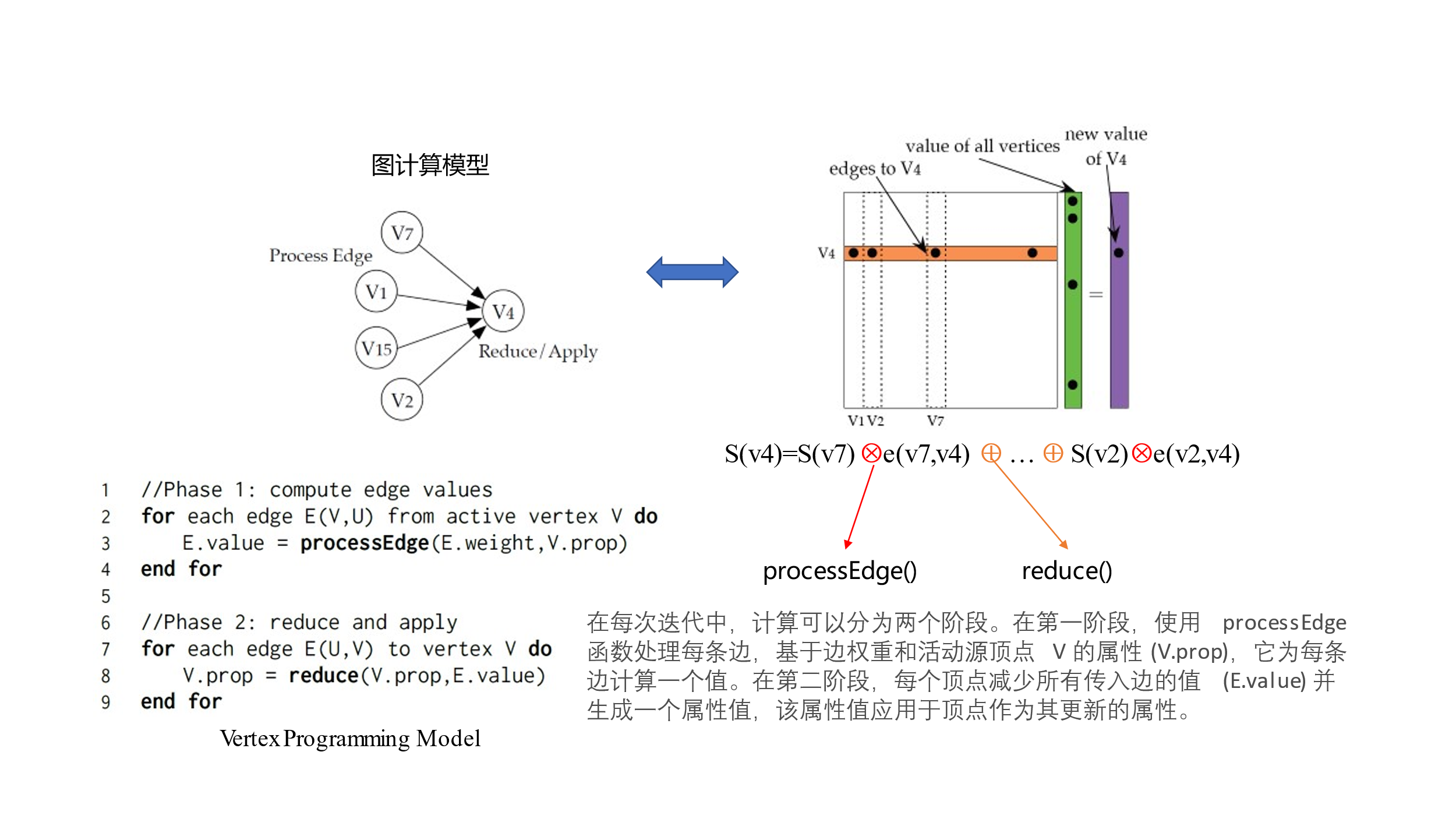
[GAT算法 24](#_Toc128388159)

[Collaborative Filtering 24](#_Toc128388160)

[参考文献 25](#_Toc128388161)

图算法表格汇总

|  |  |
| --- | --- |
| 算法 | 算法描述 |
| PageRank | 网页排名，根据网页之间相互的超链接数来计算网页的相关性和重要性 |
| SSSP | 单源最短路径，给定一个源顶点，计算其分别到其它顶点的最短路径的问题 |
| SSWP | 单源最宽路径，给定一个源顶点，计算其分别到其它顶点的最宽路径的问题 |
| Connected Components | 给定一个图，计算其中的连通分量 |
| BFS | 广度优先遍历，给定一个源顶点，计算其它顶点相对于此源顶点的深度 |
| Adsorption | 吸附算法，从图中的一个顶点开始随机移动到与其连接的其它顶点，用它到达另一个顶点次数作为两个顶点之间相似度的度量 |
| Katz Centrality | Katz中心度（网络中心度），计算在社交网络中，每个图顶点的重要性 |
| SimRank | 协同过滤推荐算法，在二部图中，通过计算，得到二部图中每一个子集顶点之间的相似度，从而进行一些实际的推荐应用 |
| HITS | 基于超链接的主题搜索网页排序算法，将网页分成两类，Hub页面和Authority页面，计算网页的Hub值和Auth值，并进行排序 |
| k-Core | K核心度紧密关联子图，在图中找出符合指定核心度的紧密关联的子图结构，在k-Core的结果子图中，每个顶点至少有k的度数，且所有顶点都至少与该子图中的k个其它顶点相连 |
| Jacobi Method | 雅可比法，一种解对角元素几乎都是各行和各列的绝对值最大的值的线性方程组的算法 |
| Rooted PageRank | 层次关系推断，计算图中两个顶点之间的相似度，通过迭代计算PageRank向量得到某一点相对于其它顶点的相似度 |
| Betweenness Centrality | 介数中心性，基于最短路径，对网络图中心性进行衡量，图中每个顶点的介数中心性即为最短路径穿过该顶点的次数 |
| SSR | 单源可达性，给定一个源顶点，找出图中所有可以到达源顶点的其它顶点 |
| Viterbi | 维特比动态规划算法，用于寻找最有可能产生观测事件序列的维特比路径，特别是在马尔可夫信息源上下文和隐马尔可夫模型中 |
| Radii estimation | 半径估计，通过运行多个单源最短路径算法，选择所有结果中的最大值用来估计图半径 |
| SSNSP | 单源最短路径数，给定一个源顶点，不仅计算其分别到其它顶点的最短路径，还计算到其它顶点的最短路径的数量 |
| SSNP | 单源最窄路径，给定一个源顶点，计算其分别到其它顶点的最窄路径的问题 |
| Belief Propagation | 置信度传播算法，在贝叶斯网络、马尔可夫随机场等概率图模型中用于推断的一种信息传递算法，在给定已观测顶点时，可以用该算法高效地计算未观测顶点地边缘分布 |
| Label Propagation | 标签传播聚类算法，是一种基于标签传播的局部社区划分算法，在每一个迭代的过程中，每一个节点根据与其相连的节点所属的标签改变自己的标签，最终紧密连接的节点将有共同的标签 |

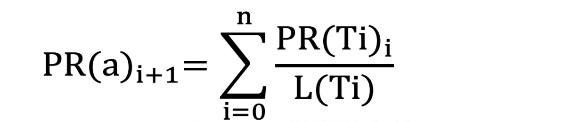
矩阵运算表格汇总

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **公式** | **Operation of ⊗** | **Operation of ⊕** | **矩阵优化** | **其它** |
| PageRank |  | × | + |  |  |
| NumPaths |  | × | + |  |  |
| Adsorption |  | ×λ | + |  |  |
| BFS |  | + | min |  |  |
| Single Source Shortest Path |  | + | min |  |  |
| Connected Components |  | × | min |  |  |
| Single Source Widest Path |  | min | max |  |  |
| Single Source Narrowest Path |  | min | min |  |  |
| Minimum Reliability Path |  | × | min |  |  |
| Maximum Reliability Path |  | × | max |  |  |
| Maximum Cost Path |  | + | max |  |  |
| Single Source Reachability |  | and | or |  |  |

详细内容

PageRank

背景

PageRank算法最初作为计算互联网网页重要度的算法被Google提出的，基于网页链接分析对关键词匹配搜索结果进行处理。它是定义在网页集合上的一个函数，它对每个网页给出一个正实数值，表示网页的重要程度，PageRank值越高，表示该网页越重要。如果一个网页被很多网页链接则其PageRank值会相对较高，另一方面一个PageRank值很高的网页链接到其它的网页，那么被链接的网页的PageRank值也会因此而提高。由于网络中网页链接的相互指向，该分值的计算为一个迭代过程，最终网页根据所得分值进行检索排序。实际应用中网页的数据主要以图的形式存在，PageRank是找出图中顶点（网页链接）的重要性。

PR（a）表示当前节点a的PR值

PR（Ti）表示其他各个节点（能够指向a）的PR值

L（Ti）表示其他各个节点（能够指向a）的出链数

i代表当前时刻或迭代次数

Google给每一个网页都赋予一个初始,然后根据PageRank算法计算其。事实上，根据以上公式进行的PR计算并不是仅仅一次就结束的，由于网页之间的相互链接，任一网页的变化，都会引起其他与之有链接关系的网页的的变化，因此，确定某网页的，需要进行多次重复的计算。而在经过一定次数的重复计算之后，各网页的基本上达到稳定。PageRank算法使所有网页的总和非常巧妙地达到了平衡，所有网页的总和在每一次计算后也是保持不变的。假设Google的索引页有10亿个，每页的初始都是1，这些网页的总和则是10亿。在每次计算之后，无论个体网页的怎么变化，但是这些网页的总和始终保持10亿。

矩阵优化

随机游走的特点是一个结点到有向边连出的所有结点的转移概率相等，设一共有n个网页，则可以组织成一个n维矩阵，矩阵的行id表示有向边的目的节点id，列id表示有向边的起始节点id。其中第i行j列的值表示用户从页面j转到页面i的概率，这样的一个矩阵叫做转移矩阵（Transition Matrix）。转移矩阵表示为：

的取值根据页面的链接情况。如果结点有个有向边连出，并且结点是其连出的一个结点，则 ，否则。

随机游走在某个时刻访问各个结点的概率分布就是马尔可夫链在时刻的状态分布，可以用一个维**列向量**表示，那么在时刻访问各个结点的概率分布满足

笔记：要算某一刻到达节点i的概率，先算之前的时刻到达节点i的入边邻居的概率，然后累积上各入边邻居传递到节点i的概率

由此，在时刻访问各个结点的概率分布为

通常取值为向量。

若极限

存在，则极限向量满足马尔科夫链的平稳分布，即

平稳分布称为有向图的PageRank。的各个分量为各个结点的PageRank值，即

其中，，表示页面的PageRank值。

PageRank算法的迭代公式如下：

笔记1：d是阻尼系数，公式后半部分是，以便于加入到Rk上

笔记2：为了处理那些“没有向外链接的页面”（这些页面就像“黑洞”会吞噬掉用户继续向下浏览的概率）带来的问题，d=0.85（这里的d被称为阻尼系数（damping factor），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。1-d=0.15就是用户停止点击，随机跳到新URL的概率）的算法被用到了所有页面上，估算页面可能被上网者放入书签的概率。

表示第轮迭代中产生的PageRank向量，表示阻尼系数，满足,是有向图的转移矩阵，表示有向图顶点数量。

算法 1.4展示了PageRank的迭代算法。根据迭代公式计算每一轮的PageRank向量。如果迭代轮数可以不超过预先规定的上限提前结束，则可以得到PageRank向量，根据向量的分量得出各页面的重要度排名。

|  |  |
| --- | --- |
| 算法1.4：PageRank () | |
| 输入：有向图，初始向量，阻尼因子，迭代的上限 | |
| 输出：PageRank向量 | |
|  | 根据边集产生转移矩阵 |
|  | 初始化迭代次数; |
|  | while() do |
|  | 计算 |
|  | if(与充分接近) then |
|  | ; |
|  | return ; |
|  | end if |
|  | ; |
|  | end while |
|  | return ; |

算法 1.4 PageRank算法

NumPaths

numpaths算法是一种图计算中的近似最短路径算法。它的核心思想是使用一种称为“路径数”的技术来模拟路径的结果，从而估计最短路径的长度。算法的步骤如下：

1. 初始化所有节点的路径数为1。

2. 对每一条边进行遍历，计算从源节点到目的节点的路径数，即路径数=源节点的路径数\*边的权重。

3. 重复步骤2，直到所有节点的路径数都已经更新完毕。

4. 计算每个节点到源节点的路径数，即路径数=目的节点的路径数\*边的权重。

5. 计算每个节点到源节点的总路径数，即总路径数=节点到源节点的路径数之和。

6. 选择总路径数最小的节点作为最短路径的终点。

numpaths算法比传统的最短路径算法更快，可以在大型图形中快速求解出最短路径。

Adsorption

Adsorption算法是图计算中的一种算法，它通过将高维图转换为低维结构来提高图计算的性能。主要的想法是将图中的一个或多个节点捆绑在一起，然后将它们视为一个整体，将其视为一个新的节点，从而将高维图转换为低维图。Adsorption算法的优势在于它可以有效地减少图的维度，从而提高计算性能，使图计算更加高效。

它的计算步骤

1. 初始化：给定需要降维的图G和需要捆绑的节点数量k。

2. 捆绑：选择k个节点进行捆绑，形成新的节点组合。

3. 更新：更新图G中的节点和边，将捆绑的节点视为一个新的节点。

4. 迭代：重复步骤2，3，直到降维的图维度达到指定的值。

5. 输出：输出降维后的图G。

广度优先遍历算法BFS

第一个广度优先遍历算法（Breadth-First Search）是由 Moore在研究通过迷宫寻找路径的问题时发现的。广度优先遍历算法（又称宽度优先遍历）是最基础的图的遍历算法之一，这一算法是很多重要的图的算法的原型。Dijkstra单源最短路径算法和Prim最小生成树算法都采用了和广度优先遍历类似的思想。广度优先遍历属于一种盲目遍历法，所谓广度，就是一层一层的，向下遍历，直到找到结果为止，是一个从近到远的扩散过程。实质就是从顶点出发，依次遍历路径长度为1，2，...的顶点。因此可以使用迭代的BFS获取相对于某一源顶点的各顶点深度。

算法的基本思想是：首先访问起始顶点v，然后由v出发，依次访问v的各个未被访问过的邻接顶点，，...，，然后再依次访问，，...，的所有未被访问过的邻接顶点，再从这些访问过的顶点出发，再访问它们所有未被访问过的邻接顶点…。以此类推，直到途中所有的顶点都被访问过为止。为了实现这种逐层的访问，算法必须借助一个辅助队列来实现。BFS算法把图的每个顶点分为两个类别：visited与not visited，可用一个标记数组表示，这样标记的目的是为了防止出现死循环。

算法步骤如下：

1、首先创建一个visit[ ]数组和一个队列q，并初始化数组和清空队列。用来判断顶点是否被访问过及让未访问过的点入队；

2、让起点start入队，并使该点的visit置1；

3、执行搜索操作

（1）取出队顶点v并访问；

（2）处理队头顶点v的所有邻接顶点；

（3）v的邻接顶点中未被访问的压入队列并置其visit为1；

4、继续重复执行步骤3，直到队列为空；

算法伪代码：

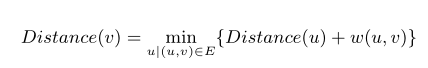
|  |  |
| --- | --- |
| 算法2.1：BFS() | |
| 输入：图，指定起点s | |
| 输出： | |
|  | Initqueue(Q),Let visited[]=0; |
|  | Q.enqueue( s ) //把顶点s压入队列中; |
|  | visited[s]=1 // 把s标记为Visited; |
|  | while ( Q is not empty) do // 如果队列Q为空则继续循环，否则退出完成; |
|  | v = Q.dequeue( ) //把队列中其邻域顶点将要被访问的顶点弹出; |
|  | Visit(v); //访问顶点； |
|  | for each all neighbours w of v in Graph G do //处理顶点v的所有邻接顶点; |
|  | If visited[w]=0 do |
|  | Q.enqueue( w ) //把v的邻接顶点中未被访问的压入队列; |
|  | visited[w]=1 //并且把该邻接点标记已经访问; |
|  | end if |
|  | end for |
|  | end while |

算法1.1 BFS算法

BFS是一种相对简单的图遍历算法，常用于求图中两点间的最短路径，以及求图中顶点的连通性等问题。同时，它也可以为图的其他遍历算法提供支持。BFS的时间复杂度为O(V + E)，BFS算法的复杂度是以顶点数量和边数量的总和来计算的，因此当图的规模不断增加时，BFS的复杂度也会相应增加。但是，由于BFS采用了宽搜的策略，因此可以保证每个结点只遍历一次，这样就可以避免出现重复遍历的情况，从而使得BFS算法的复杂度比较稳定。

单源最短路径算法SSSP

路径查找（Path finding）算法是建立在图遍历算法的基础上，它探索顶点之间的路径，从一个顶点开始，采用某种遍历顺序，直到到达目的顶点。这些算法用于识别图中的最优路由，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。最短路径（Shortest Path）算法是计算一对顶点之间的最短（加权）路径。

单源最短路径（Signal Scource Shortest Path）是一种基本的图算法，它计算了图论中的一个经典问题。在计算机网络、交通运输工程、通信工程等各种应用中，常常需要计算图中从某个源点到其余各顶点的最短路径或各顶点之间的最短路径。很多的优化问题在数学上都有研究。因此，在图论中，最短路径算法比其它算法研究得更为透彻。关于单源最短路径的算法。目前最流行的经典算法是Dijkstra算法。具体来说，Dijkstra算法首先找到从起始顶点到直接连接顶点的最小权重关系。它跟踪这些权重并移动到“最近”顶点。然后，它执行相同的计算，只不过权重的累积是从初始顶点开始算的。具体来说，它采用贪心算法的策略，可以计算非负权值图里的单源最短路径问题。初始时将源顶点以外所有顶点的距离值置为，源顶点的距离为。而后算法从源顶点开始，每次遍历都走最短的路径，并不断更新每个点到起点的最短距离。将获取到最短距离的顶点进行标记，下一次以此顶点获取其它顶点的最短路径，每次循环可以找到一个具有最短距离的顶点，由此最多重复次（为图中包含的顶点数），可以得到所有顶点的结果。

其中w(u, v)表示边(u, v)的权重。最初，每个顶点的距离设置为无穷大，但距离值设置为 0 的源除外。我们使用 Bellman-Ford 最短路径算法的细微变化，我们只更新那些与改变它们的顶点相邻的顶点的距离上一次迭代的距离。

Dijkstra算法采用了两个集合来存储图顶点，并声明一个数组dis[]来保存源点到各个顶点的最短距离。算法的主要思想是：顶点集合被划分为两部分：集合和。首先假定源点为u，初始时中仅含有源点u，源点u的路径权重被赋为0（即dis[u] = 0）。若对于顶点u存在能直接到达的边<u,m>，则把dis[m]设为k(k为对应边<u,m>的路径权重)，同时把所有其他（u不能直接到达的）顶点的路径长度设为无穷大。然后，从数组dis[]选择最小值，则该值就是源点s到该值对应的顶点v的最短路径，并且把该点v加入到S中。之后我们需要看看新加入的顶点v是否可以到达其他顶点w并且看看通过该顶点v到达其他点w的路径长度是否比源点直接到达短（即存在dis[w]>dis[v]+dis[v,w]），如果是，那么就替换这些顶点的dis[w]。然后，又从数组dis[]中找出最小值，重复上述动作，直到S中包含了图的所有顶点。dis[] 就是从源点到所有其他顶点的最短路径长度。

算法步骤如下：

1、先建立两个集合：用来存已经找到的最短路径的结点即S；用来存未找到的最短路径的结点即U。S 和U互补，S U U=V（所有结点），初始时中仅含有源点，U为集合。

2、若源点u与顶点i直接相连，则dist[i]等于权重，若源点u与顶点i不直接相连，则dist[i]=∞，建立初始数组dist[]。

3、选取最小dist[i]所对应的顶点i加入到S中，并将i从U中移除。基于找到的顶点i，更新源点u到集合U中顶点的距离。

4、重复3步骤，直到U为空。

算法伪代码：

|  |  |
| --- | --- |
| **算法1.2**：SSSP\_Dijkstra () | |
| **输入：**带权图，边权值向量，指定源顶点 | |
| **输出：**源点到图中其它所有顶点的最短路径 | |
|  | 初始化访问标记数组visited[]和路径数组; |
|  | **for** **to** **do** |
|  | 找到当前未被访问且使最小的顶点; |
|  | visited[v]=1; //将顶点置为已被访问 |
|  | **while**() **do** |
|  | **if**(未被访问 && +<) **then** |
|  | ; //优化顶点的最短距离 |
|  | **end if** |
|  | **end while** |
|  | **end for** |
|  | **return** ; |

算法1.2 单源最短路径算法

Dijkstra算法支持负权边，但不支持负权环，它的时间复杂度为O(n^2)或O(E+VlogV)。

强/弱连通分量算法：

背景

图计算领域的强联通分量算法和弱连通分量算法，是用来划分有向图中的子图，使得每个子图内部的顶点之间都可以互相到达。强联通分量要求子图内部的顶点之间不仅可以沿着边的方向到达，还可以逆着边的方向到达。弱连通分量则只要求子图内部的顶点之间可以沿着边的方向到达。

参考1：[图论：连通分量和强连通分量](https://zhuanlan.zhihu.com/p/37792015)

参考2：[基本图论-连通分量(强/弱联通 割点/边 边/点双)](https://www.cnblogs.com/y2823774827y/p/10179342.html)

*参考3：[【数据结构】图的连通分量](https://blog.csdn.net/HuaLingPiaoXue/article/details/88897246)*

一种常用的强联通分量算法是Kosaraju算法。它基于两次深度优先搜索（DFS）来找出所有的强联通分量。具体步骤如下：

对原始图进行DFS，记录每个顶点结束搜索时的时间戳

对原始图进行转置，即反转所有边的方向

对转置后的图按照时间戳从大到小进行DFS，每次搜索得到一个强联通分量

一种常用的弱连通分量算法是Tarjan算法，它基于一次深度优先搜索（DFS）来找出所有的弱连通分量。具体步骤如下：

对原始图进行DFS，记录每个顶点开始搜索时的时间戳

在DFS过程中维护一个栈，每次遇到一个新顶点就将其压入栈中

在DFS过程中维护一个数组low[]，记录每个顶点能够回溯到最早时间戳的顶点

如果某个顶点v是当前栈底元素，并且low[v]=v，则将栈中从v开始往上所有元素出栈，并形成一个弱连通分量

矩阵优化

参考1：[算法学习笔记(69): 强连通分量](https://zhuanlan.zhihu.com/p/349530425)

参考2：[强连通分量（超详细！！！）](https://www.cnblogs.com/ljy-endl/p/11562352.html)

用矩阵操作来优化强（弱）连通分量的计算过程，它利用了图的邻接矩阵和强（弱）连通分量的关系。具体来说，如果A是图的邻接矩阵，那么A^n 的对角线元素就表示每个顶点能够到达自己的路径数。如果某个顶点v能够到达自己，那么它就属于一个强连通分量。如果把A转置为A'，那么A'^n 的对角线元素就表示每个顶点能够从自己出发到达其他顶点的路径数。如果某个顶点v能够从自己出发到达其他顶点，那么它就属于一个弱连通分量。这个方法可以利用高效的矩阵乘法算法和并行计算技术来加速。它还可以通过设置一个阈值来提前终止迭代过程，从而减少不必要的计算。

用伪代码的形式写出用矩阵操作，优化强（弱）连通分量的计算过程，可以参考以下代码：

# 输入：图的邻接矩阵A，阈值t

# 输出：图的强（弱）连通分量的个数和划分

# 初始化

n = A.shape[0] # 图的顶点数

B = A.copy() # 保存原始邻接矩阵

C = A.transpose() # 转置邻接矩阵

count = 0 # 强（弱）连通分量的个数

partition = [0] \* n # 强（弱）连通分量的划分

# 迭代计算A^n 和C^n

while True:

# 计算A^n 和C^n 的对角线元素

diag\_A = [A[i][i] for i in range(n)]

diag\_C = [C[i][i] for i in range(n)]

# 检查每个顶点是否属于一个强（弱）连通分量

for i in range(n):

if partition[i] == 0: # 如果顶点i还没有被划分

if diag\_A[i] > 0 and diag\_C[i] > 0: # 如果顶点i能够到达自己并且从自己出发到达其他顶点

count += 1 # 增加一个强（弱）连通分量的个数

partition[i] = count # 把顶点i划分到当前的强（弱）连通分量中

# 找出和顶点i在同一个强（弱）连通分量中的其他顶点，并划分到当前的强（弱）连通分量中

for j in range(n):

if B[i][j] > 0 and B[j][i] > 0: # 如果存在路径从i到j并且从j到i

partition[j] = count # 把顶点j划分到当前的强（弱）连通分量中

# 判断是否达到终止条件，即所有顶点都被划分或者对角线元素都大于阈值t

if all(partition) or all(x > t for x in diag\_A) or all(x > t for x in diag\_C):

break

# 否则，更新A和C为下一次迭代所需的矩阵乘积

A = matrix\_multiply(A, B)

C = matrix\_multiply(C, B.transpose())

# 返回结果

return count, partition

Single Source Widest Path

Single Source Widest Path（单源最宽路径）算法是建立在有向图上的一种路径搜索算法，主要用于解决在单源最短路径问题上宽度优先搜索的改进。

它的核心思想是：从一个源点出发，可以按照任意指定的宽度值（即每条边的权值）来搜索最短路径，直到找到最终的目标点。这种算法可以帮助我们更好地理解每条路径的权重，从而有效地搜索最宽路径。

伪代码：

// 一个有向图 G=(V,E)

// 源点 s

// 每条边的权重为w

// 初始化：设置每个点的宽度值为 0

for each v in V do

v.width = 0

// 开始搜索

Q = {s} // 将源点放入队列

while Q is not empty do

u = pop(Q) // 从队列中取出一个顶点

for each v adjacent to u do // 扫描u的邻接顶点

if v.width < u.width + w(u, v) then // 更新宽度值

v.width = u.width + w(u, v)

if v is not in Q then // 将顶点 v 加入队列

push(Q, v)

Single Source Narrowest Path

Single Source Narrowest Path算法（单源最窄路径算法）是一种用于解决最窄路径问题的算法，它用于从单个源点到其他所有结点的最窄路径。它可以用于求解有向图和无向图上任意两点之间的最窄路径，也可以用于解决复杂的网络中的最窄路径问题。

伪代码：

// 初始化算法

1. 初始化距离数组dist[]，将源点设置为dist[0]=0，其他结点设置为dist[i]=∞，

2. 将源点放入集合S中，

// 主循环

3. 重复以下操作，直至所有结点都放入集合S：

(1) 在所有未放入集合S的结点中，找出距离源点最窄的结点u；

(2) 将结点u放入集合S中；

(3) 更新距离源点的距离dist[u]；

算法计算步骤不需要矩阵操作，主要是通过遍历图中所有节点，比较每个节点与源点的距离，来更新距离源点最窄的结点u，并将结点u放入集合S中，最终求出从源点到其他所有结点的最窄路径。

Minimum Reliability Path

Minimum Reliability Path（MRP）是一种图计算中的经典算法，它的目标是在边权重可能存在不确定性的情况下，求解从源点到汇点之间的最可靠的路径。MRP算法的主要步骤如下：

（1）建立路径可靠性矩阵。

（2）计算源点到每个顶点的可靠性值，其中源点的可靠性值初始化为1.0，其余顶点的可靠性值都初始化为0.0。

（3）重复以下步骤，直到所有顶点的可靠性值不再发生变化：

（3.1）对每一条边，计算出该条边的可靠性值。

（3.2）对每一个顶点，计算出该顶点的可靠性值。

（4）最终，求解源点到汇点之间的最可靠路径。

MRP算法伪代码：

（1）输入：图G=(V,E)，源点s，汇点t

（2）输出：从源点s到汇点t的最可靠路径

（3）建立路径可靠性矩阵M

（4）for each v∈V do

if v=s then

R[v]=1.0

else

R[v]=0.0

end

（5）while true do

（6）for each (u,v)∈E do

R[v]=max{R[v],R[u]\*M[u,v]}

（7）if R[v] did not change then

break

（8）end

（9）end

（10）输出R[t]

MRP算法的计算步骤需要矩阵操作，因为需要使用矩阵来储存路径可靠性信息，以及计算每一条边和每一个顶点的可靠性值。

Maximum Cost Path

Maximum Cost Path算法是图计算中一种用来求解有权图中从源点到终点的最大费用路径的算法，它通过动态规划来求解最大费用路径问题。

伪代码：

1.初始化：将一张有权图G=(V,E)转换为一张费用矩阵M，其中M[i,j]表示从顶点i到顶点j的费用；

2.动态规划：从源点s开始，初始化一个数组cost[n]，其中cost[i]表示从源点s出发，到达顶点i的最大费用；

3.循环：从源点s开始，对每一个顶点i，比较cost[i]与M[s,i]+cost[s]的大小，将较大的值赋给cost[i]，即：cost[i] = max(cost[i], M[s,i]+cost[s]);

4.结束：当计算到终点t时，最大费用路径的值就是cost[t]，即最大费用路径从源点s到终点t的最大值。

算法计算步骤需要矩阵操作，即需要转换成费用矩阵M，在计算过程中比较M[s,i]+cost[s]的大小。

Single Source Reachability

Single Source Reachability算法是一种用于解决图中两点之间是否可达的算法，即给定一个源点，通过某种算法求出图中所有点都能到达哪些点。Single Source Reachability算法的伪代码如下：

// INPUT: Graph G = (V, E) and source node s

// OUTPUT: Set of nodes that can be reached from s

Set Reachable(s) = {s} // Init reachable set to contain only the source node

Queue Q = {s} // Init queue to contain only the source node

while Q is not empty

v = pop from Q

for each neighbor w of v

if w not in Reachable

add w to Reachable

add w to Q

return Reachable

算法计算步骤是否需要矩阵操作：不需要。Single Source Reachability算法在进行计算的过程中，只需要维护一个队列和一个集合，而不需要矩阵操作。

三角形计数

背景

图数据中的三角形指的是一个由三个不同的结点构成的集合，这些结点两两之间都有边关联。三角形计数问题是指在一个图中，找出所有由三条边构成的三角形的个数。这个问题在图算法中非常重要，因为它可以用来计算图的一些性质，比如聚集系数和k-truss23，也可以用来解决一些实际应用，比如欺诈检测、推荐、社交网络分析等。三角形计数的代码有很多种：

* [Node-iterator algorithm**1**](https://cs.stanford.edu/~rishig/courses/ref/l1.pdf):

count = 0

for each node v in graph:

for each pair u,w of distinct neighbors of v:

if u and w are also neighbors:

count = count + 1

return count / 3

笔记1：之所以最后要除以3，是因为算法针对的是无向图。一个三角形会从是哪个顶点出发，计算三次，最终结果除以3才是正确的。

* [Edge-iterator algorithm**2**](https://iss.oden.utexas.edu/?p=projects/galois/analytics/triangle_counting):

count = 0

for each edge e = (n,m) in graph:

for each node a in intersection(neighbors(n), neighbors(m)):

if n < a < m: # some criteria to avoid double counting

count = count + 1

return count

* [Wedge-sampling algorithm**1**](https://cs.stanford.edu/~rishig/courses/ref/l1.pdf):

count = 0

sample\_size = some number based on graph size and desired accuracy

for sample\_size times:

pick a random edge e = (u,v) from graph with probability proportional to degree(u) \* degree(v)

pick a random neighbor w of u with probability proportional to degree(w)

if v and w are also neighbors:

count = count + 1

return count \* (number of edges / sample\_size)

矩阵优化

参考1：[如何求解无向图中三角形的个数？](https://www.zhihu.com/question/36652212)

参考2：[Finding and counting given length cycles](https://link.springer.com/article/10.1007/BF02523189)

