Prime Path 实现详解

范照云 516106001834

说明：红色字体为代码中的变量，斜体为算法流程，下划线为算法思想

本文从图的生成出发，在生成的随机图上面寻找所有的 prime path。图的生成采用随机生成边的方式实现。Prime path 生成过程包括两个版本，算法 1 采用纯暴力解法，时间复杂度较高，效率慢；算法 2 基于 prime path 的特点生成路径，时间复杂度低，运行快。

1. 图的生成

给定节点数 nodeNum，随机生成有向图。整体思路：循环生成随即边，每次随机生成 nodeNum 范围内的节点对构成一条边，两个节点应该各异，即不生成从自己到自己的边。根据生成的边为所有的节点计算出度和入度。当除起始节点的入度和结束节点的出度可为 0，其余节点的出度和入度都不为 0 时退出循环。然后处理边不连通的情况。（概率极低，编程实现时没有考虑，源码中该部分已被我注释。）

算法步骤：

(1) 根据节点数生成所有节点的邻接表 graph ;出度统计 outdegree，入度统计 indegree;

(2) 随机生成节点 start 和 end，属于（0, nodeNum）之间；

(3) 如果 start 等于 end，则继续随机生成 start 和 end；否则以邻接表的形式将 end 插入到 graph[start]中；

(4) 更新节点的出度和入度

(5) 判断是否除起始节点的入度和结束节点的出度可 0，其余节点的初度和入度都不为 0。如果是，则结束循环，否则跳到(2)；

(6) 如果某一个节点的邻接节点序号不比它大，则声称一个比它大的节点序号，并连接两个节点构成一条边，为每个节点都做这样的检查。

图的表示：本文所述方法使用 python list 存储图的邻接表。如：

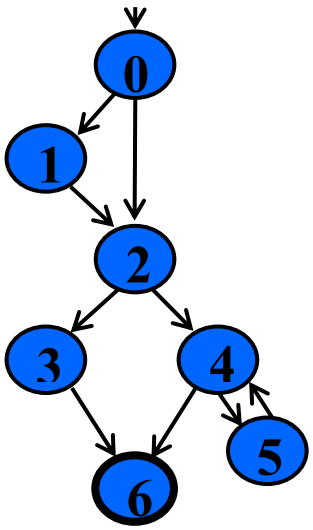

图 1 测试图

graph = [[1,2],[2],[3,4],[6],[5,6],[4],[]]

2. 查找 prime path

本部分用了两种算法实现 prime path 的查找，主要的区别在最后从 simple path 到 prime path 部分，后面将详细讲解，此处从查找 simple path 开始。

在一个图结构中，查找图的路径比较常用的是 DFS（深度优先搜索）和 BFS（广度优先搜索），从 prime path 的定义可知此处易采用 BFS，BFS 常用利用栈结构实现。本文所描述的是 python 的实现版本，鉴于语言本身的特性和实际需求，此处使用 list 存储 simple path，不再利用栈的特性。

查找 simple path 的过程：

(1) 初始化 simplePath 为只包含单个节点的列表；

(2) 判断 simplePath 中的每一个 list 中最后一个节点是否是终结点，即无后继节点的节点。(SimplePath 最开始时每个 list 中只包含一个节点，就是（1）中添加的节点。）如果没有后继节点则将此条 path 加入 tempPath 进行 primepath 的验证，本文所述两种方法的不同就在于此，所以这里不进行讲解，prime path 的验证留待后文讲解。

(3) 如果有后继结点：构造此 path 的深拷贝 copysimple，先从后继节点中取出第一个节点①，后继节点与此条 path 的头结点相同，将后继结点加入 path，并且此条 path 加入 primepath，并从 simplePath 中删除；②，后继结点与此条 path 中除头节点之外的其他结点相同，此条 path 加入 tempPath，并从 simplePath 中删除。③，后继结点不满足以上两种情况，为当前 path 添加后继节点并放回 simplePath 中。

(4) 其他后继节点，①，后继节点与此条 copysimple 的头结点相同，将后继结点加入 copysimple，并且此条 copysimple 加入 primepath；②，后继结点与此条 copysimple 中除头节点之外的其他结点相同，此条 copysimple 加入 tempPath，③，后继结点不满足以上两种情况，为当前 copysimple 添加后继节点并将 copysimple 添加到 simplePath 中。

(5) 如果 simplePath 为空则执行其他（指这里的两种方案），否则回到（2）继续。

2.1 暴力解法

在 simplePath 的生成过程中，如果遇到环则直接加入 primePath，否则加入 tempPath 进行 prime path 的验证。暴力解法中我们在每一次 simple path 迭代中只是将路径加入 tempPath，所有的验证留到迭代结束进行。具体步骤如下：

(1) 验证 tempPath 中的每条路径是否已经存在于 primePath 中了，如果存在则继续验证 tempPath 中的下一个路径，否则（2）

(2) 验证 tempPath 中的每条路径是否已经存在于其他 primePath 中，如果是则继续

验证 tempPath 中的下一个路径，否则将该条路径加入 primePath。

这种解法存在大量的重复验证，针对它的一个改进可以是先对 tempPath 再进行按长度排序，这样就只需要进行（1）的验证即可。

从 prime path 的特点来看，我们不需要等到全部迭代结束再进行验证，可以在每次迭代结束即可进行。因为总的 tempPath 是固定的，simplePath 的路径随着迭代次数的增加先增加后减少，这样看来，每次需要进行验证比较的次数将极大的减少。

2.2 根据 prime path 的特性验证 tempPath

我们从 prime path 的定义来看，它有两个特点：一是 prime path 不能存在于其他路径中，且非环路径中的节点不能出现两次及以上；二是如果存在环路，则只允许头节点和尾结点相同。后者我们在 simple path 生成时已经进行了处理，直接加入到了 primePath 中，现在要处理的问题就是前者。前者的非头节点出现两次这种情况已经在 simple path 生成时丢弃，再者，如果存在这样的路径本身也不满足我们 simple path 的定义。综上所述，我们仅需处理的就是 prime path 不能存在于其他路径中，也就是说 prime path 应该是最长路径。我们回到 simple path 的生成过程，每次 simple path 的生成是迭代生成，也就是说下一次 simplePath 中的路径一定比上一次 simplePath 中的路径长，并且每一次的迭代我们都是按照结点全迭代的方式，也就是说如果存在某条路径包含于其他路径中，那么它一定可以在下一次迭代结束之后就可以验证。如下：

我们以图 1 为例，假设现在有一条路径为[2, 3, 6]，节点 6 无后继节点，所以该条路径需要进行 prime path 的验证，我们来看本次迭代中的其他路径，一定有一条是[1, 2, 3]，由图可知，当进行下一次迭代之后，我们一定能在长度为 4 的路径中找一条路径[1, 2, 3, 6]。也就是说原来的[2, 3, 6]包含在了这里的[1, 2, 3, 6]中，所以尽管[2, 3, 6]包含了尾节点，但它不是最长的路径，所以不是 prime path。

从代码来看，我们对 tempPath 中的路径验证就不需要等到最后集中验证，只需要在下一次迭代后就可验证，验证比较的集合就是本次生成的其他所有路径（包括本次已经加入 primePath 中的环路径。），步骤如下：

（1） *验证 tempPath 中的每条路径是否已经存在于 simplePath 中，如果存在则继续验证 tempPath 中的下一个路径，否则（2）*

（2） *验证 tempPath 中的每条路径是否已经存在于 tpcirclePath(本次迭代加入 primePath 中的环路径)中，如果存在则继续验证 tempPath 中的下一个路径，否则改路径加入 primePath。*