频繁模式挖掘

邱文韬

May 18, 2023

1 频繁模式

1.1 频繁项集

根据韩家炜 [1] 等，首先给出关于频繁项集的形式化定义：

Definition 1.1 (项集)**.** 项的集合称为项集 (Itemset)，包含 k 个项的项集称为 k-项集。

Definition 1.2 (支持度)**.** 项集的支持度 (Support) 是一个项集在数据中的出现频率。

Definition 1.3 (频繁项集)**.** 频繁项集 (Frequent Itemset) 是根据用户自行设定最小支持度阈值 $minsup$，支持度大于 $minsup$ 的项集称为频繁项集。

频繁项集具有单调特性，即如果一个项集是频繁的，则其所有子集都是频繁的。相应地，如果一个项集不是频繁的，则其所有的超集都是不频繁的。

1.2 频繁子图

有了前述频繁项集的相关定义，相应的，可以给出频繁子图的相关概念：

Definition 1.4 (子图)**.** 图 $G' = (V', E')$ 是另一个图 $G = (V, E)$ 的子图，如果它的顶点集 V' 是 V 的子集，并且它的边集 E' 是 E 的子集。子图关系记作 $G' \subseteq sG$

Definition 1.5 (子图的支持度)**.** 给定图的集族 ζ, 子图 g 的支持度为包含它的所有图所占的百分比即

$$s(g) = \frac{|\{G_i | g \subseteq sG_i, G_i \in \zeta\}|}{|\zeta|} \tag{1}$$

Definition 1.6 (频繁子图挖掘)**.** 给定图的集合 ζ 和支持度阈值 $minsup$，频繁子图挖掘的目标是找出所有使得 $s(g) \geq minsup$ 的子图 g。

1.3 频繁子图挖掘的复杂度

因为具有指数级别的搜索空间，所以挖掘频繁子图的计算量很大。考虑包含 d 个实体的数据集，在频繁项集挖掘中，每个实体是一个项，搜索空间大小为 2^d，这是可能产生的候选项集的个数。在频繁子图挖掘中，以无向连通图为例，每个实体是一个顶点，并且最多可以有 $d-1$ 条到其他顶点的边。假定顶点的标号是唯一的，则子图的总数为

$$\sum_{i-1}^{d} C_d^i \times 2^{i(i-1)/2}$$

其中，C_d^i 是选择 i 个顶点形成子图的方法数，而 $2^{i(i-1)/2}$ 是子图的顶点之间边的最大值。韩家炜等 [1] 给出了不同实体个数下项集与子图的数量，如表1

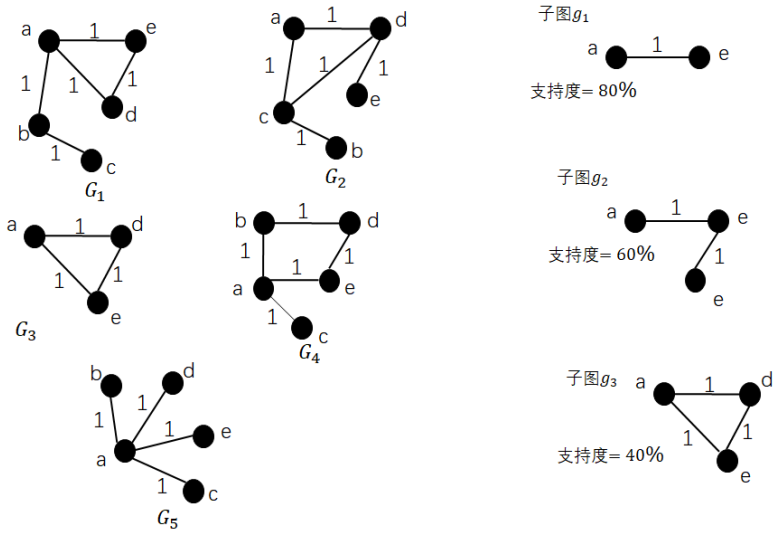


Figure 1: 由图集计算子图的支持度

Table 1: 不同维度 d，项集和子图个数的比较

实体个数 d	1	2	3	4	5	6	7	8
项集个数	2	4	8	16	32	64	128	256
子图个数	2	5	18	113	1450	40069	2350602	286192513

2 频繁项集生成的经典算法

2.1 *APriori* 算法

APriori 算法是一个经典的频繁项集挖掘算法，在 1994 年由 IBM 研究员 Agrawal 提出 [2]，它的核心思想是: 广度优先搜索，自底而上遍历，逐步生成候选集与频繁项集，它基于一个重要的原理——反单调性原理 (Antimonotonicity)，其定义如下:

Definition 2.1 (反单调性)**.** 如果一个项集是频繁的，则它的所有子集一定也是频繁的。即

$$\forall X, Y : X \subseteq Y \rightarrow support(X) \geq support(Y)$$

其中 *support* 表示支持度，依据该性质，对于某 k+1 项集，只要存在一个 k 项子集不是频繁项集，则可以直接判定该项集不是频繁项集。算法步骤主要分为连接步和剪枝步。

连接步: 从频繁 $K-1$ 项集生成候选 K 项集 ($K \geq 2$)。具体来说，对于两个频繁 $K-1$ 项集 I 和 J，如果它们的前 $K-2$ 个元素相同，那么可以将它们合并成一个候选 K 项集 $I \cup J$。

剪枝步: 从候选 K 项集中筛选出频繁 K 项集。具体来说，对于每个候选 K 项集 C，检查它的所有 $K-1$ 子集是否都出现在频繁 $K-1$ 项集中，如果有任何一个子集不是频繁的，则可以将 C 剪枝掉。

算法伪代码如下:

Algorithm 1 Apriori Algorithm

Input: Dataset D, minimum support $minsup$

Output: Set of frequent itemsets L

$L_1 \leftarrow$ {frequent 1-itemsets};

$k \leftarrow 2$;

while $L_{k-1} \neq \emptyset$ **do**

  $C_k \leftarrow$ candidate k-itemsets generated from L_{k-1};

  prune C_k to obtain L_k;

  $k \leftarrow k + 1$;

end

return $\bigcup_k L_k$;

2.2 *FP-Growth* 算法

FP-Growth 算法也是一个经典的频繁项集挖掘算法，于 2000 年由韩家炜等提出 [3]，它的核心思想是构造一棵能够压缩原始数据的频繁模式树 (Frequent Pattern Tree, FP-tree)，成功构造 FP 树后，就可以使用递归的分治方法来挖掘频繁项集。其伪代码如下：

Algorithm 2 FP-Tree 构造算法

输入: 事务数据集 D，最小支持度阈值 $minsup$

输出: FP-Tree

扫描事务数据集 D 一次，获得频繁项的集合 F 和其中每个频繁项的支持度。

对 F 中的所有频繁项按其支持度进行降序排序，结果为频繁项表 L;

创建一个 FP-Tree 的根节点 T，标记为 “null” ;

for 事务数据集 D 中每个事务 $Trans$ **do**

  对 $Trans$ 中的所有频繁项按照 L 中的次序排序;

  对排序后的频繁项表以 $[p|P]$ 格式表示，其中 p 是第一个元素，而 P 是频繁项表中除去 p 后剩余元素组成的项表;

  调用函数 $insert\_tree([p|P], T)$;

Algorithm 3 $insert\_tree([p|P], root)$

输入: 项表 $[p|P]$，FP-Tree 的根节点 $root$

输出: 更新后的 FP-Tree

if $root$ 有孩子节点 N 且 $N.item\_name = p.item\_name$ **then**
  $N.count++$;
else
  创建新节点 N;
  $N.item\_name = p.item\_name$;
  $N.count++$;
  $p.parent = root$;
  将 $N.node\_link$ 指向树中与它同项目名的节点;
if P 非空 **then**
  把 P 的第一项目赋值给 p，并把它从 P 中删除; 递归调用 $insert\_tree([p|P], N)$;

有如下购物记录事务数据库，取最小支持度阈值 $minsuport = 3$，由于篇幅限制且构造过程过于冗长，在此仅给出最终得到的 FP-Tree，如图2所示。

Table 2: 购物记录事务数据库

牛奶, 鸡蛋, 面包, 薯片
鸡蛋, 爆米花, 薯片, 啤酒
鸡蛋, 面包, 薯片
牛奶, 鸡蛋, 面包, 爆米花, 薯片, 啤酒
牛奶, 面包, 啤酒
鸡蛋, 面包, 啤酒
牛奶, 面包, 薯片
牛奶, 鸡蛋, 面包, 黄油, 薯片
牛奶, 鸡蛋, 黄油, 薯片

3 挖掘频繁子图的算法

3.1 概述

Akihiro Inokuchi 等人最早将 *Apriori* 算法思想应用到频繁子图挖掘中 [4]，引起了诸多学者对频繁子图挖掘的注意，各种算法也就应运而生 [5, 6]，韩家炜等人提出了将 *FP-Growth* 思想应用到子图挖掘中 [7, 8]，使得频繁子图挖掘算法得到了迅速发展。后来，许多研究人员，如 Jun Huan 等人提出了 *FFSM*[9] 等基于 *FP-Growth* 思想的算法，使得频繁子图挖掘算法得到了进一步的发展。

频繁子图挖掘的主要方法有：

1) 基于贪心的方法，基于贪心策略的频繁子图挖掘是频繁子图挖掘领域最先发展起来的技术之一，其中最著名的是 *SUBUE* 算法 [10]，*SUBUE* 算法基于最小描述长度原则 (minimum description length, *MDL*) 来发现子结构。

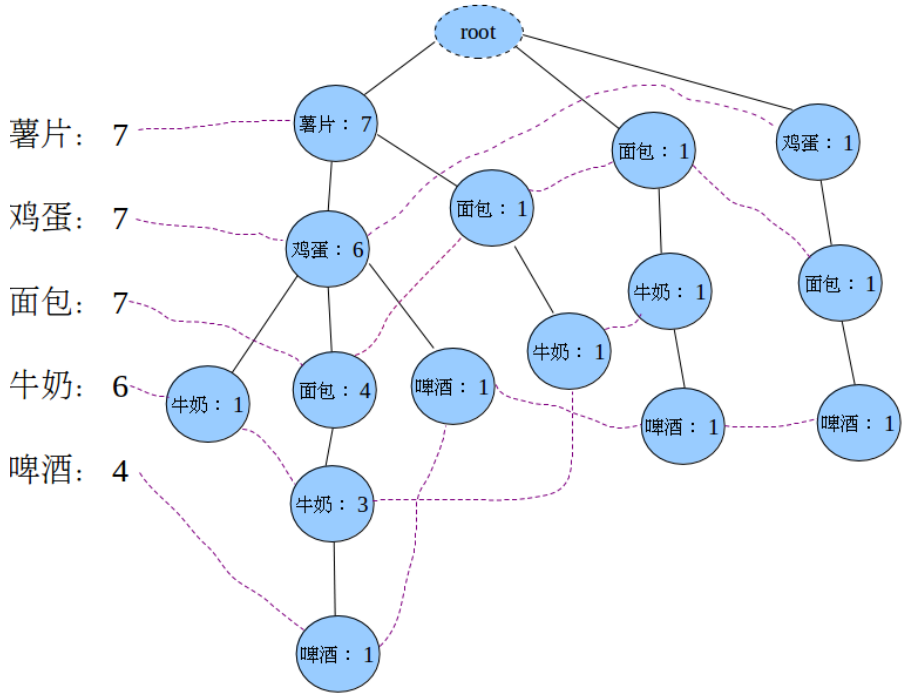


Figure 2: FP-Tree 示例

2) 基于归纳逻辑编程 (Inductive Logic Programming，*ILP*) 的方法，其优点在于大多算法能找出出现频率高的子图，且能作为交好的类识别器。但其缺点在于不能保证发现所有的频繁子图。1998 年 Dehaspe 基于 *ILP* 提出可对频繁子图进行完全挖掘的 *WARMR* 算法，其算法核心思想与 *Apriori* 算法类似。[11]

3) 基于 *Apriori* 的方法——*AGM* 算法 [4]，*AGM* 算法在效率上与采用了分层搜索的 *ILP* 算法架构相比有了比较大的提升，而且在化学分子结构这样的真实世界图数据中也有不错的表现。但是其在判断模式图 X 和 Y 是否具有相同子图时，仍要花费较多时间。且每添加一个顶点产生新候选频繁子结构时，会产生许多冗余的 $k+1$ 顶点的子结构。

4) 对 *Apriori* 方法的改进——*FSG* 算法 [12]，*FSG* 是 *AGM* 算法的一种改进。同基于 *Apriori* 的方法一样，其采用了分级扩展的方法。*FSG* 算法对 *AGM* 算法的优化主要体现在其采用了基于边的候选频繁子图生成方法，效率有所提升。不过仍产生了冗余的候选模式子图，所以仍需进行规范化判断。

5) 基于模式增长的方法——*gSpan*(graph-based Substructure pattern mining) 算法 [7], Xifeng Yan 提出的 *gSpan* 算法通过对图进行深度优先搜索遍历来发现频繁子图，解决了 *AGM* 和 *FSG* 算法基于 *Apriori* 逐层推进的方法所遇到的两个瓶颈：i. 从 k 阶频繁子图构造 $k+1$ 阶频繁子图相当复杂且代价昂贵。ii. 因为子图同构测试是一个 NPC 问题，所以处理误报的代价也是极其昂贵。

6) 混合型方法——*FFSM*(Fast Frequent Subgraph Mining) 算法 [13, 14]，快速频繁子图挖掘 *FFSM* 采用了垂直搜索模式，通过解决潜在的子图同构问题并减少冗余候选子图的生成，大大提升了效率。

后文将首先介绍类 *Apriori* 方法的步骤，再具体介绍上述算法中采用了 *Apriori* 以及 *FP-Growth* 等思想的算法。

3.2 类 *Apriori* 方法

3.2.1 步骤一：候选产生

在候选产生阶段，合并 $(k-1)$ 子图为 k 子图，首要问题是如何定义子图的大小 k。通过添加一个顶点，迭代地扩展子图的方法称作顶点增长 (vertex growing)。k 也可以是图中边的个数，添加一条边到已有的子图中来扩展子图的方法称作边增长 (edge growing)。

为了避免产生重复的候选，我们可以对合并施加条件：两个 $(k-1)$ 子图必须共享一个共同的 $(k-2)$ 子图，共同的子图称为核 (core)。由此，产生了两种子图候选产生的方法：

1. 顶点增长 (vertex growing)

   通过添加一个新的结点到已经存在的一个频繁子图上来产生候选。可以使用邻接矩阵来表示图，每一项 $M(i,j)$ 为链接 v_i 和 v_j 的边或者 0。

$$MG1 = \begin{bmatrix} 0 & p & p & q \\ p & 0 & r & 0 \\ p & r & 0 & 0 \\ q & 0 & 0 & 0 \end{bmatrix} \quad MG2 = \begin{bmatrix} 0 & p & p & 0 \\ p & 0 & r & 0 \\ p & r & 0 & r \\ 0 & 0 & r & 0 \end{bmatrix} \quad MG3 = \begin{bmatrix} 0 & p & p & q & 0 \\ p & 0 & r & 0 & 0 \\ p & r & 0 & 0 & r \\ q & 0 & 0 & 0 & ? \\ 0 & 0 & r & ? & 0 \end{bmatrix}$$

   如图3所示，其中 G_1, G_2 具有相同的核：

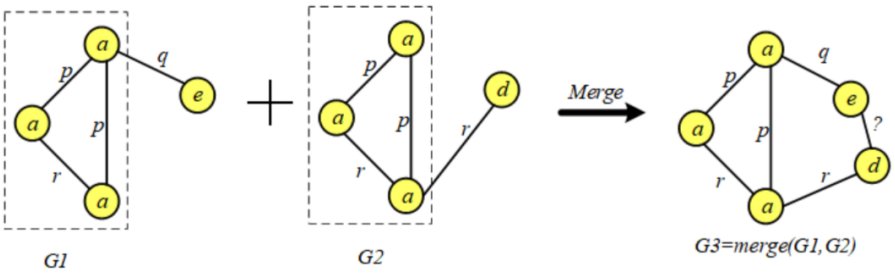


Figure 3: 顶点增长示意图

2. 边增长 (edge growing)

   边增长将一个新的边插入到一个已经存在的频繁子图中。与顶点增长不同，结果子图的顶点个数不一定增长。边增长合并过程：一个频繁子图 G_1 和另一个频繁子图 G_2 合并，仅当从 G_1 删除一条边后得到的子图与从 G_2 中删除一条边后得到的子图的拓扑等价，合并后的结果是 G_1 添加 G_2 那条之外的边。边增长示意如图4。

3.2.2 步骤二：候选剪枝

候选剪枝需要剪去 $(k-1)$ 子图非频繁的候选。可以通过以下方式实现：相继从 k 子图中删除一条边，并检查得到的 $(k-1)$ 子图是否连通且频繁。如果不是，则该候选 k 子图可以丢弃。意思是一次删除一条，总共有 k 个子图要检测。

为了检查 $(k-1)$ 子图是否频繁，需要将其与其他的 $(k-1)$ 子图匹配，此时需判定两个图是否拓扑等价，即处理图同构的问题，图同构的定义如下。

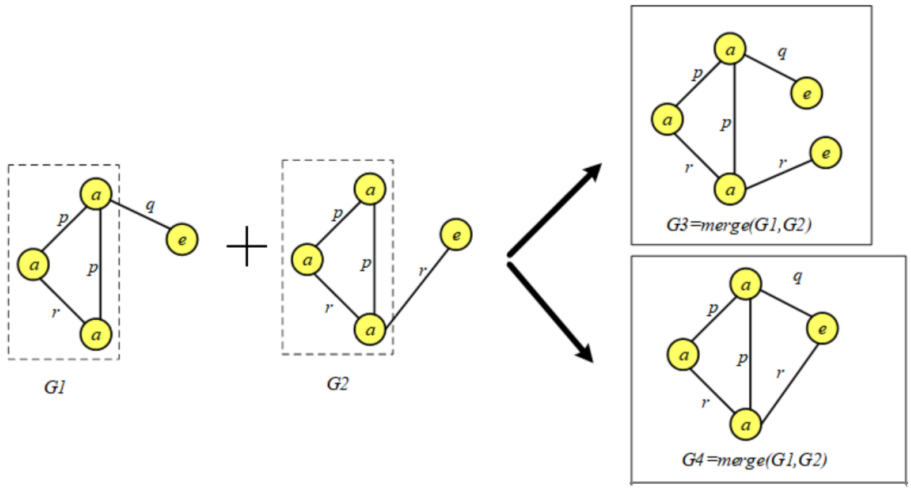


Figure 4: 边增长示意图

Definition 3.1 (图的同构)**.** 对于给定的两个图 $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$，如果存在一个定义在 $V_1 \to V_2$ 双射函数 f 满足对于 $(u, v) \in E_1 \to (f(u), f(v)) \in E_2$，那么就称这两个图是同构的 (Isomorphism)。

图5是一个图同构的例子，顶点之间并没有颜色区分，为了更好地看出顶点间的映射关系，加上了颜色。

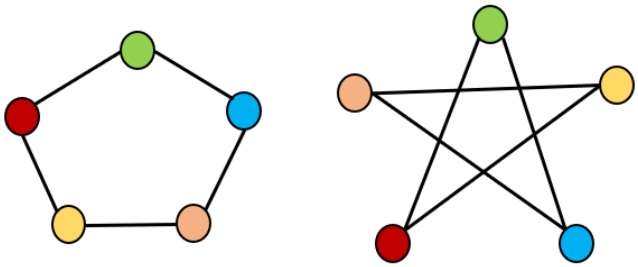


Figure 5: 图同构示意图

3.2.3 步骤三：支持度计算

当剪枝完成之后，将剪枝后剩余的候选图进行支持度计算，将其中大于支持度阈值的候选图记为满足条件的频繁子图，支持度的计算见公式1。

3.2.4 步骤四：候选删除

丢弃支持度小于 $minsup$ 的所有候选子图，对于所有频繁的候选子图，还需要进行图的同构测试，以减除重复的频繁候选子图。

3.3 *AGM* 算法

AGM 算法将图用邻接矩阵 X_k 表示为 $G(X_k)$，满足按照顶点的标签来排序顶点在邻接矩阵中顺序：

$$\min(\mathrm{lb}(v_i)) \leq \min(\mathrm{lb}(v_{i+1})) \quad \text{for } i = 1, 2, \ldots, k-1$$

同时为每个邻接矩阵表示的图都建立图的标记形式 $\mathrm{code}(X_k)$。对一个已顶点排序的邻接矩阵 X_k，

$$X_k = \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \cdots & x_{1,k} \\ x_{2,1} & x_{2,2} & x_{2,3} & \cdots & x_{2,k} \\ x_{3,1} & x_{3,2} & x_{3,3} & \cdots & x_{3,k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{k,1} & x_{k,2} & x_{k,3} & \cdots & x_{k,k} \end{pmatrix}$$

若为无向图，则其编码标记形式记为：

$$\mathrm{code}(X_k) = x_{1,1}x_{1,2}x_{2,2}x_{1,3}x_{2,3}x_{3,3}x_{1,4}\ldots x_{k-1,k}x_{k,k}$$

若为有向图，则在每个 $x_{i,j}$ 后加上其对应的 $x_{j,i}$，如下：

$$\mathrm{code}(X_k) = x_{1,1}x_{1,2}x_{2,1}x_{2,2}x_{1,3}x_{3,1}x_{2,3}x_{3,2}x_{3,3}\ldots x_{k-1,k}x_{k,k-1}x_{k,k}$$

该算法通过计算 $\sup(G_s)$，即图数据集 G_D 中包含 G_s 子结构的图 G 的个数与图数据集 G_D 中图 G 的总个数之比，如果 $\sup(G_s)$ 超过阈值 minsup，则认为 G_s 是一个频繁子结构。：

$$\sup(G_s) = \frac{\text{number of } G \in G_D \text{ where } G_s \subseteq G}{\text{total number of } G \text{ in } G_D}$$

类似于 *Apriori* 算法，频繁诱导子图的候选集依据子图的大小分层搜索所得。例如，若 X_k 和 Y_k 都是 k 阶顶点排序的邻接矩阵，若 X_k 和 Y_k 仅第 k 行和第 k 列的数据不同，即包含一个 $k-1$ 阶子图，则可由之构造 Z_{k+1}：

$$X_k = \begin{pmatrix} X_{k-1} & x_1 \\ x_2 & x_{k,k} \end{pmatrix}, \quad Y_k = \begin{pmatrix} X_{k-1} & y_1 \\ y_2 & y_{k,k} \end{pmatrix}$$

$$Z_{k+1} = \begin{pmatrix} X_{k-1} & x_1 & y_1 \\ x_2^T & x_{k,k} & z_{k,k+1} \\ y_2^T & z_{k+1,k} & y_{k,k} \end{pmatrix} = \begin{pmatrix} & \mathrm{X}_k & & y_1 \\ & & & z_{k,k+1} \\ y_2^T & z_{k,k+1} & & y_{k,k} \end{pmatrix}$$

并将 X_k 称为第一矩阵，将 Y_k 称为第二矩阵。为减少出现重复无用的构造，使 Z_{k+1} 的构造满足：

$$\mathrm{code}(X_k) \leq \mathrm{code}(Y_k)$$

并将之称为标准形式。

同时，[12] 定义了图邻接矩阵的规范形式 X_c，即对同一图 G 的所有标准形式的邻接矩阵集 $NF(G)$ 中具有最小 code 的那一个邻接矩阵：

$$X_c = \mathrm{argmin}_{X \in NF(G)} \mathrm{code}(X)$$

因此，对 $G(X_k)$ 删去任意一个顶点 v_m 得到 X_{k-1}^m，对其使用 T_{k-1}^m 变形矩阵进行标准化得到标准的邻接矩阵 $X_{k-1}'^m$，再使用 S_{k-1} 矩阵，将其变形为规范形式（记对应于 $X_{k-1}'^m$ 的规范化变形矩阵为 S_{k-1}^m），即：

$$(T_{k-1}^m S_{k-1}^m)^T X_{k-1}^m T_{k-1}^m S_{k-1}^m$$

其中变形矩阵的求法如下：

$$s_{i,j} = \begin{cases} s_{i,j}^{(m)} & 0 \le i \le k-1 \text{ and } 0 \le j \le k-1 \\ 1 & i = j = k \\ 0 & \text{otherwise} \end{cases}, \quad t_{i,j} = \begin{cases} t_{i,j}^{(m)} & i < m \text{ and } j \ne k \\ t_{i-1,j}^{(m)} & i > m \text{ and } j \ne k \\ 1 & i = m \text{ and } j = k \\ 0 & \text{otherwise} \end{cases}$$

$$Xck = \text{argmin}_{m=1,\ldots,k} \, \text{code}((T_{k-1}^m S_{k-1}^m)^T X_{k-1}^m T_{k-1}^m S_{k-1}^m)$$

其中，$s_{i,j}$，$s_{i,j}^m$，$t_{i,j}$，$t_{i,j}^m$ 表示矩阵 S_k^m，S_{k-1}^m，T_k^m，T_{k-1}^m 的元素。而能使得 X_{ck} 最小的 $T_{k-1}^m S_{k-1}^m$ 即为 X_k 的 S_k 变形矩阵。

显然，由于一个 $k+1$ 阶图是频繁子结构当且仅当其所有诱导子图是频繁子结构。故若 $G(X_k)$ 通过删去一个任意一个顶点 v_m（$1 \le m \le k$）所得到的 $k-1$ 阶子图 $G(X_{k-1})$ 都是频繁子结构的话，则 $G(X_k)$ 被看作是候选频繁子结构。

在得到所有的候选频繁子结构之后，扫描数据集，对图数据集中的每一个图 G 的 X_k 都要标准化来检测候选频繁子结构是否在该图 G 中。而这个过程会得到图 G 每个诱导子图的标准形式。是故，频繁度的计算是基于所有图 G 的诱导子图的标准形式的。

[12] 所提出的 AGM 算法在效率上与采用了分层搜索的 ILP 算法架构相比有了比较大的提升，而且在化学分子结构这样的真实世界图数据中也有不错的表现。但是其在判断模式图 X 和 Y 是否具有相同子图时，仍要花费较多时间。且每添加一个顶点产生新候选频繁子结构时，会产生许多冗余的 $k+1$ 顶点的子结构。

3.4 *FSG* 算法

FSG 是 *AGM* 算法的一种改进。同基于 *Apriori* 的方法一样，其采用了分级扩展的方法。但优化之处在于：

- 其采用了相对稀疏的图表示方法，来最小化存储空间和计算开销。
- 每次添加一条边来扩大频繁子图的大小，由此使得生成候选集更有效。
- 采用了对小图更有效的规范标签和图同构算法。
- 其对生成候选集进行了各种优化并统计了能适用于大规模的图数据库的优化措施。

其具体的算法思想如下：首先，枚举出所有的频繁 1 阶和 2 阶子图。然后以此为源，迭代计算。在每次迭代中，首先生成比上次迭代出的频繁子图多一条边的候选子图。然后计算每个候选子图的频繁度，除去不到阈值的，将满足阈值的加入频繁子图集。

其中，$\mathcal{D}$ 表示图数据集，t 表示数据集中的一个图数据，k-subgraph 表示 k 条边的 (子) 图，g^k 表示 k 阶子图。$\mathcal{C}^k$ 表示具有 k 条边的候选集。$\mathcal{F}^k$ 表示 k 阶频繁子图，伪代码如下：

Algorithm 4 Algorithm of $fsg(D, \sigma)$ (Frequent Subgraph)

$F^1 \leftarrow$ detect all frequent 1-subgraphs in D;
$F^2 \leftarrow$ detect all frequent 2-subgraphs in D;
$k \leftarrow 3$;
while $F_{k-1} \neq \emptyset$ **do**
  $C^k \leftarrow fsg - gen(F_{k-1})$;
  for *each candidate graph* $g^k \in C^k$ **do**
    $g^k \cdot count \leftarrow 3$;
    for *each transaction* $t \in D$ **do**
      if *candidate* g_k *is included in transaction* t **then**
        $g_k \cdot count \leftarrow g_k \cdot count+1$;
      end
    end
  end
  $F_k \leftarrow \{g_k \in C_k | g_k \cdot count \geq \sigma|D\}$;
  $k \leftarrow k + 1$;
end
return $F^1, F^2, \ldots, F^{k-2}$;

$fsg\text{-}gen(\mathcal{F}^k)$ 是生成候选集 $\mathcal{C}^k$ 的算法，伪代码如下：

Algorithm 5 Algorithm of $fsg - gen(F^k)$ (Candidate Generation)

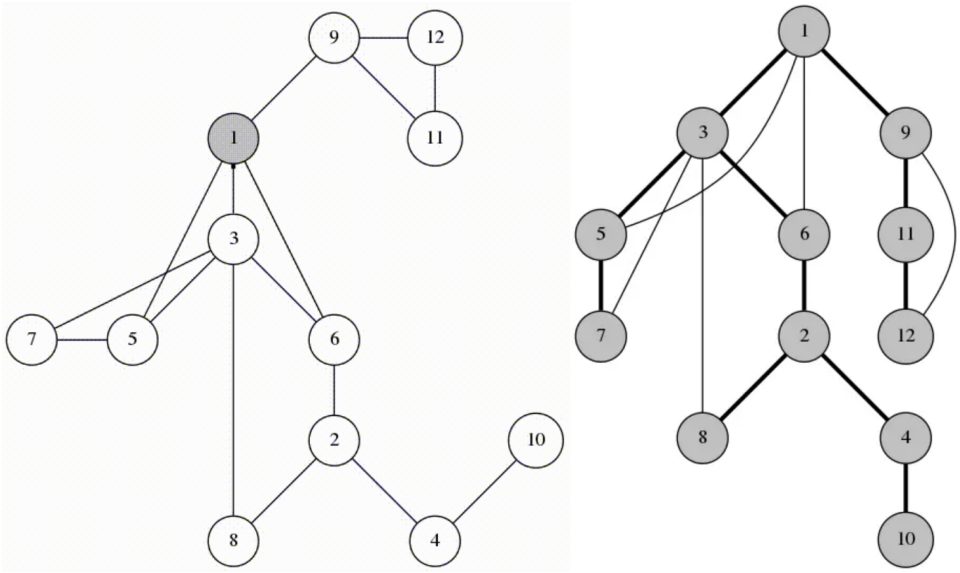
$C^k \leftarrow \emptyset$;
for *each pair of frequent subgraphs* $g_i^k, g_j^k \in F^k, i \leq j$ *such that* $cl(g_i^k) \leq cl(g_j^k)$ **do**
  for *each edge* $e \in g_i^k$ **do**
    create a $(k-1)$-subgraph of g_i^k by removing an edge e
  $g_i^{k-1} \leftarrow g_i^k - e$
  if g_i^{k-1} *is inclued in* g_j^k **then**
    g_i^k and g_j^k share the same core
  $T^{k+1} \leftarrow fsg \cdot join(g_i^k, g_j^k)$
  for *each* $g_j^{k+1} \in T^{k+1}$ **do**
    test if the downward closure property holds for g_j^{k+1}
  flag←true
  for *each edge* $f_l \in g_j^{k+1}$ **do**
    $h_l^k \leftarrow g_j^{k+1} - f_l$
    if h_l^k *is connected and* $h_l^k \notin F^k$ **then**
      flag←false
      break
  if *flag = true* **then**
    $C^{k+1} \leftarrow C^{k+1} \cup \{g^{k+1}\}$;
return C^{k+1};

在判断频繁度计算时，*FSG* 算法使用 Transaction ID lists 来对每个频繁子图维护一个事务标识符的列表。则当计算 g^{k+1} 的频繁度时，先计算其 k 阶子图的 TID 列表的交叉量，若

低于阈值，则舍弃 g^{k+1}，否则计算和搜索也仅限于 g^{k+1} 的 k 阶子图的 TID 列表的交叉事务集中。

3.5 *gSpan* 算法

AGM 和 *FSG* 算法都采用了基于 *Aporiori* 逐层推进的方法，为了解决 *Aporiori* 模式会遇到的瓶颈，Xifeng Yan 于 2003 年提出了 *gSpan*(graph-based Substructure pattern mining) 算法 [7]，通过对图进行深度优先搜索 (*DFS*) 遍历来发现频繁子图。*gSpan* 算法在挖掘频繁子图的时候，用了和 *FP-Growth* 中相似的原理，即模式增长 (Pattern-Grown) 的方式，挖掘的过程中同样没有产生候选集，也用到了最小支持度计数作为一个过滤条件。要了解 *gSpan* 算法，首先要了解几个概念：最右顶点与最右路径、前向边与后向边以及 *DFS* 编码。



((a)) 以 1 为根结点的原始图　　((b)) 生成的 DFS 树

Figure 6: 图的深度遍历及生成的 DFS 树

3.5.1 最右顶点与最右路径

gSpan 在构建深度搜索树对图进行深度优先遍历，所有节点根据发现时间排序，其中 v_0 为根，最后发现的节点叫做最右节点 v_n，从第一个节点 v_0 到最右节点 v_n 的路径叫做最右路径。

3.5.2 前向边与后向边

基于节点先后发现的顺序，每条边可以表示为 (i, j)，如果为前向边，则 $i < j$，反之为后向边 $i > j$。

3.5.3 *DFS* 编码

根据前述，每个节点都有一个标记，则 *DFS* 编码可以将具有标记的图转化为边序列，每个边 (*DFS Code*) 用一个五元组表示：

$$(i, j, l_i, l_{(i,j)}, l_j)$$

即 (节点 i 的访问次序标记，节点 j 的访问次序标记，节点 i 的标签，边的标签，节点 j 的标签)。边的排序：假设有两条边 e_1, e_2，分别是两个五元组。如果满足 $e_1 < e_2$，那么他们符合下述情况之一：

1. e_1 和 e_2 都是前向边，并且 $j_1 < j_2$；
2. e_1 和 e_2 都是后向边，并且 ($i_1 < i_2$ 或者 $i_1 = i_2$ 且 $j_1 < j_2$)；
3. e_1 是前向边，e_2 是后向边，并且 $j_1 \leq i_2$；
4. e_1 是后向边，e_2 是前向边，并且 $i_1 < j_2$。

根据边的次序得到的边序列是图的 *DFS* 编码。图7是一个由图 (a) 得到 *DFS* 树示例，表3是其对应的 *DFS Code*，显然深度优先搜索树是不唯一的，所以 *DFS Code* 也是不唯一的。

将 *DFS* 序列定义为 $code(G,T)$。对于给定图 G 的 DFS 树 T 的集合，其有对应的 DFS 序列集 Z：

$$Z = \{code(G,T)|T是G的DFS树\}$$

其中，$code(G,T)$ 表示图 G 的 DFS 树 T 的编码。DFS 序列是指将所有 DFS 树的编码按某种顺序排列形成的序列。可以对给定图 G 的 DFS 序列集 $Z(G) = code(G,T)|T是G的DFS树$ 找到图 G 的最小 DFS 序列，也即 $min(Z(G))$，定义为图 G 的规范化标签。由此，两个图 G 与 G' 同构当且仅当 $min(Z(G)) = min(Z(G'))$。

gSpan 算法通过这样的定义构造，将寻找频繁子图的问题转化为相应的最小 *DFS* 序列的问题，而这样的序列模式挖掘问题可以用已有的序列模式挖掘算法解决。简要来说，*gSpan* 算法的流程就是，在得到 k 阶频繁子图的 *DFS* 编码树后，对其进行最右路径扩展。每次增加一条边，产生 $k+1$ 阶候选频繁子图。若 $k+1$ 阶候选频繁子图具有最小 *DFS* 序列（即规范化标签 $min(Z(G_{k+1}))$），则将其保留，否则丢弃该候选子图。而在计算 k 阶频繁子图的支持度 $support(G_k)$ 的时候，记录频繁子图的所有嵌入。由此，$k+1$ 阶候选频繁子图的支持度 $support(G_{k+1})$ 即可通过 k 阶频繁子图的嵌入进行最右扩展而计算得到。

通过这种方法，*gSpan* 算法可以有效地挖掘频繁子图。与 *Apriori* 算法和 *FP-Growth* 算法不同，*gSpan* 算法使用 *DFS* 编码树来表示子图，通过最小化 *DFS* 序列来剪枝，从而提高了算法的效率。

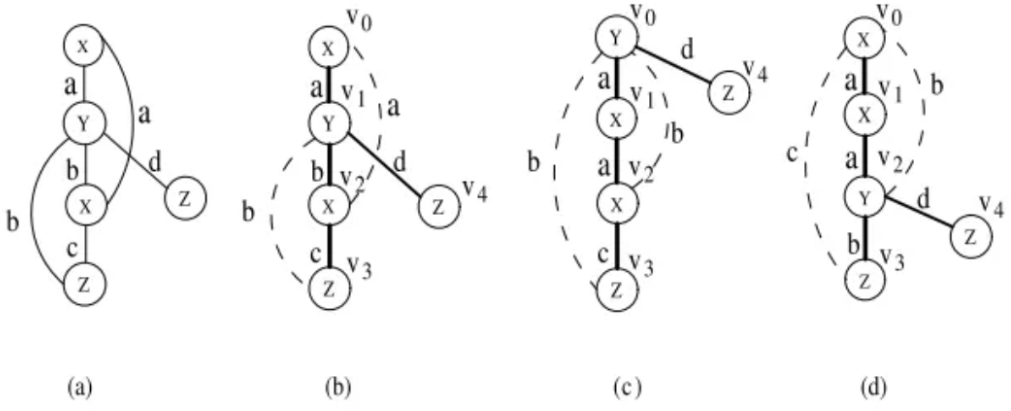


Figure 7: 深度优先搜索树

参考 GitHub 上开源的 Python 实现的*gSpan 算法*，我将项目安装至本地，并使用其中一个图数据 graph.data.simple.5 进行测试，该图为无向加权图，作为示例运行了 *gSpan* 算法，用

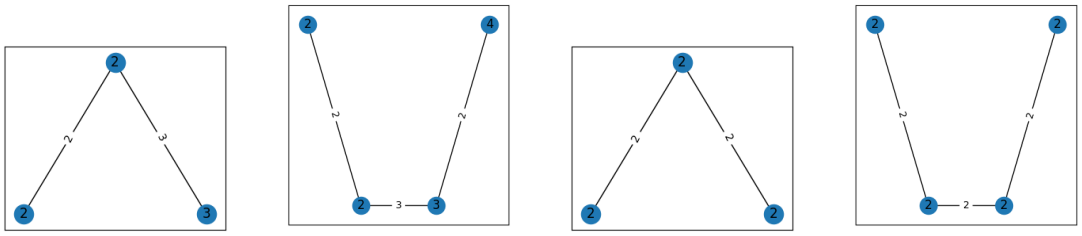
Table 3: DFS 树对应的 DFS Code

edge	Fig 7b	Fig 7c	Fig 7d
0	(0,1,X,a,Y)	(0,1,Y,a,X)	(0,1,X,a,X)
1	(1,2,Y,b,X)	(1,2,X,a,X)	(1,2,X,a,Y)
2	(2,0,X,a,X)	(2,0,X,b,Y)	(2,0,Y,b,X)
3	(2,3,X,c,Z)	(2,3,X,c,Z)	(2,3,Y,b,Z)
4	(3,1,Z,b,Y)	(3,0,Z,b,Y)	(3,0,Z,c,X)
5	(1,4,Y,d,Z)	(0,4,Y,d,Z)	(2,4,Y,d,Z)

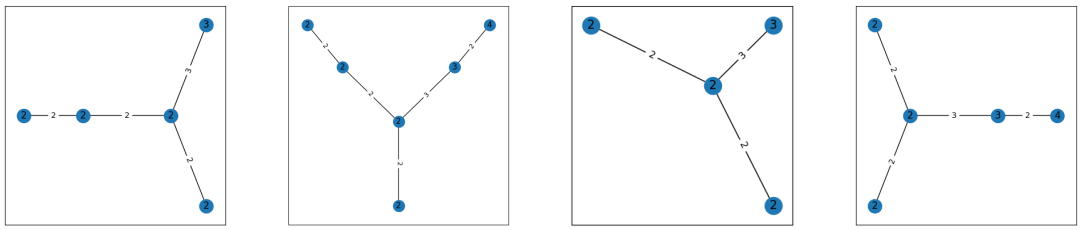
于挖掘 graph.data.simple.5 的频繁子图，由于图结构较简单，当支持度阈值设置为 3 时，仅有一个频繁子图，所以我设置最小支持度阈值为 2，并对结果进行了可视化，该测试图如8所示。

Figure 8: 测试无向加权图

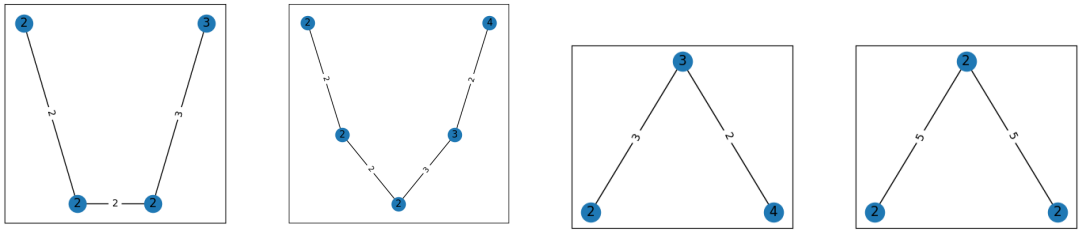
挖掘的结果如图9，支持度大于等于 2 的子图共有 12 个，在该数据集上运行的时间为 0.16s。



((a)) ((b)) ((c)) ((d))



((e)) ((f)) ((g)) ((h))



((i)) ((j)) ((k)) ((l))

Figure 9: 挖掘出的支持度大于等于 2 的频繁子图

References

[1] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499. [Online]. Available: http://www.vldb.org/conf/1994/P487.PDF

[3] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. ACM, 2000, pp. 1–12.

[4] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *Principles of Data Mining and Knowledge Discovery*. Springer Berlin Heidelberg, 2000, pp. 13–23.

[5] ——, "Applying the apriori-based graph mining method to mutagenesis data analysis," *Journal of Computer Aided Chemistry*, vol. 2, pp. 87–92, 2001.

[6] A. Inokuchi, T. K. Nishimura, and H. Motoda, "A fast algorithm for mining frequent connected subgraph," IBM Research, Tokyo Research Laboratory, Tech. Rep., 2002.

[7] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*. IEEE, 2002, pp. 721–724.

[8] A. Inokuchi, T. Washio, and H. Motoda, "Complete mining of frequent patterns from graphs: Mining graph data," in *Machine Learning*, 2003, pp. 321–354.

[9] J. Huan, W. Wang, J. Prins, and J. Yang, "Spin: Mining maximal frequent subgraphs from graph databases," UNC, Technical Report TR04-018, 2004.

[10] L. B. Holder, D. J. Cook, and S. Djoko, "Substructure discovery in the subdue system," in *KDD workshop*, 1994, pp. 169–180.

[11] L. Dehaspe, H. Toivonen, and R. D. King, "Finding frequent substructures in chemical compounds," in *KDD*, 1998, p. 1998.

[12] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*. IEEE, 2001, pp. 313–320.

[13] J. Huan, W. Wang, and J. Prins, "Efficient mining of frequent subgraphs in the presence of isomorphism," in *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*. IEEE, 2003, pp. 549–552.

[14] S. Nijssen and J. N. Kok, "A quickstart in frequent structure mining can make a difference," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2004, pp. 647–652.