从大规模蛋白质相互作用网络中挖掘频繁拓扑结构

摘要

蛋白质相互作用网络（PPI 网络）是一类重要的生物网络。通过寻找不同物种的 PPI 网络中所共有的保守子结构（这种保守子结构可以通过图的拓扑结构来定义），可以发现不同生物之间的同源相似性，这对于研究生物的起源、进化以及生物功能的比较和预测等，都具有重要的现实意义。

本文对在大规模 PPI 网络上进行频繁拓扑结构挖掘的问题进行了初步的探索和研究，基于一般标记图集上频繁模式挖掘的相关工作提出了一个分两阶段进行挖掘的算法框架并给出了各主要部分的详细实现，同时结合 PPI 网络自身的特点对算法进行了不少改进和优化。另外，通过实验对算法的性能进行了深入地测试和评估，发现了不少有价值的规律，为算法的进一步完善提供了依据。在本文的最后，演示了算法在 Pathway 系统上的实际运行效果，证明了算法的实用价值。

关键词：蛋白质相互作用网络，图的拓扑结构，频繁模式挖掘

Abstract

Protein-Protein Interaction (PPI) network is an important class of biologic networks. By finding common conserved substructures from different species' PPI networks, (this kind of conserved substructure can be defined as graph's topological minor) we can discover the homologic similarities among these spieces, which are of great significance to many biological research fields such as the origin and evolution of species, the comparison and prediction of biologic functions and etc. .

In this paper, we try to solve the problem of mining frequent topological structures from large scale PPI networks as mentioned above. Based on the related research work of frequent pattern mining on general labeled graph datasets, we propose a two-stage mining algorithm framework and specify every key part's implementation details. Meanwhile, we introduce many improvements and optimizations into our algorithm according to the special characteristics of PPI networks. Furthermore, we do many experiments to test and evaluate the performance of our algorithm, and find some valuable information and rules. We will improve both the time and space efficiency of our algorithm in the future work with respect to these experimental results. We demonstrate the algorithm's practicability at the end of this paper by showing the actual running effect of the Pathway system when it uses our algorithm.

Keywords：PPI networks, Graph topological minor, Frequent pattern mining

目录

第一章 绪论

蛋白质相互作用网络（Protein-Protein Interaction Network）是一类重要的生物网络。生物学研究表明，不同物种的蛋白质相互作用网络之间所共有的保守子结构（即拓扑结构，后文将详细介绍和定义）可能反映了不同生物之间的某种同源相似性，因而寻找这些公共子结构对于研究生物的起源与进化，以及生物功能的比较与预测等，都有着特别重要的意义。然而，实际的蛋白质相互作用网络往往规模巨大，因此单纯依靠人工方法来进行网络间的比对进而发现保守子结构是不现实的，必须借助于计算机这一工具。而在计算机科学领域，数据挖掘技术尤其是图挖掘技术的发展使得寻找有效的计算机算法来解决上述问题成为了可能。

在这一章里，我们先简要地介绍一下数据挖掘领域特别是图挖掘领域的研究背景，然后我们将对蛋白质相互作用网络的挖掘问题进行一个概念上的描述（问题的精确定义将在 3.1 节给出），最后我们将对本文的研究对象和组织结构进行一个说明。

1.1 数据挖掘领域的研究背景

数据挖掘（Data Mining）是从大量的、不完全的、有噪声的、模糊的、随机的数据中提取隐含在其中的、事先不为人所知的、但又是潜在有用的信息和知识的过程。

对数据挖掘技术的研究兴起于上世纪 80 年代后期。当时，关系数据库模型已经确立，数据库技术经过数十年的发展已经逐渐趋于成熟，越来越多的企业和组织已经习惯于采用数据库来存储和查询信息。另一方面，互联网（Internet）开始进入一个飞速发展的阶段，人们开始习惯于通过互联网来实现信息的传递和资源的共享。数据库和互联网技术飞速发展的一个直接结果就是信息量的迅猛膨胀。但是，对于某一个个体而言，他并不一定能从增加的信息量中获益，理由是随着信息量的增加，如何从中提取对自己有价值的信息（或者说是知识）将变得更为困难。数据挖掘研究就是在这样的背景下产生和发展起来的。

数据挖掘所要处理的数据对象是多种多样的：根据信息存储格式，用于挖掘的对象有关系数据库、面向对象数据库、数据仓库、文本数据源、多媒体数据库、空间数据库、时态数据库、异质数据库以及 Internet 等。即使是同样的数据对象，也存在着诸如规模，完整性，可靠性等差异。

数据挖掘的任务主要有：模式挖掘、关联分析、聚类分析、分类、预测、时

序模式和偏差分析等。

目前，数据挖掘的许多子领域已经进行了比较深入的研究，得到了许多经典的算法和结果。但是，随着信息技术的高速发展，数据量的急剧增长，出现了许多新的需求，一些原有的结论和方法在这些新的需求上会变得不太适用。特别是一些新的复杂的数据对象和挖掘任务的出现，迫切需要新的方法来进行处理。

1.2 图（结构化）数据集上的频繁模式挖掘工作

首先简单地介绍一下频繁模式挖掘的概念。频繁模式挖掘是指对给定的数据集，找出其中出现频率（或相应的频次）高于某个给定的支出度阈值的那些模式，而具体模式的形式则是多种多样的。（比如说对于一个给定的文本数据集，我们可以要求寻找出所有出现频率超过 10%的单词，这里的单词就可以视为一个模式。）这些挖掘出来的频繁模式往往包含着有意义的信息，（比如上面的例子中抽取出的单词可能就包含着关于该文本数据集内容的重要信息）更多的情况下，这些模式常常可以作为后续挖掘工作的坚实基础（比如关联分析的第一步就是频繁模式挖掘），因此，进行频繁模式挖掘的研究是具有现实意义的。

近年来，在数据挖掘领域，越来越多的研究者将自己的研究兴趣投入到结构化数据集上的频繁模式挖掘工作中来。结构化数据集的例子有：化学合成物，生物蛋白质，社会关系网络，XML 数据集等等。由于这些数据通常都可以用图来建模，所以目前主要的研究集中在频繁子图的挖掘算法上。（关于子图的概念将在 2.1 节介绍）

然而，在现实世界的不少应用中，比如生物网络（包括我们下面将要介绍的蛋白质相互作用网络），社会关系网络，远程通讯网络等，挖掘出图的频繁拓扑结构（定义将在 2.3 节给出）往往比普通的频繁子图更有价值。比如，在分析社会关系网络或远程通讯网络时，许多时候我们并不关心直接相邻的两个顶点之间的联系，而是更关注这样的模式：几个顶点通过相互独立的路径进行相互联系。但是，想直接通过现有的频繁子图挖掘算法来得到这样的模式是相当困难的，这就需要寻找新的途径。我们将在 3.3 节详细介绍在图数据集上进行频繁模式挖掘方面的相关研究工作。

1.3 蛋白质相互作用网络的挖掘

1.3.1 蛋白质相互作用网络挖掘的现实意义

生物的蛋白质之间通过相互作用实现相关的生理机能，一组相关的作用构成功能模块。有些功能代表生物的基本生理功能，为大多数或多种物种所共有。一些功能在漫长的进化时间里相对稳定地得到保存。如果我们能找出物种之间蛋白质作用过程的对应关系，那么我们可以将已有的相对成熟的研究成果，移植到相对陌生的生物上。通过对多个蛋白质作用网络图的全局比对和局部比对，可以找到物种间的保守路径或功能模块，这样就为物种间研究成果的共享借鉴及蛋白质结构功能的预测提供了依据。

从近期国际上蛋白质组学研究的发展动向可以看出，揭示蛋白质之间的相互作用关系，建立相互作用关系的网络图，已成为揭示蛋白质组复杂体系与蛋白质功能模式的先导，业已成为蛋白质组学领域的研究热点。

1.3.2 蛋白质相互作用网络的图模型

至今为止，对蛋白质相互作用网络模型已经有了很多的研究。这些研究的结果表明，蛋白质相互作用网络可以被建模为一种无尺度的顶点度数服从幂律分布(scale-free power law degree sequence)的复杂网络。（即网络中出现同 k 个其它蛋白质作用的蛋白质的概率为 k^{y}，y 是一个网络特定的参数。Barab´asi 等在[1]中提出了一种基于偏好连接的网络模型来解释这一现象，也就是在网络中新增加的顶点和度比较高的顶点连接的概率更高，这意味着两个节点的连接能力的差异可以随着网络的扩张而增大。）这一结果为我们用无向图来对蛋白质相互作用网络进行建模提供了可靠的理论依据。

除了上述的特点外，蛋白质相互作用网络图还有以下一些特点：

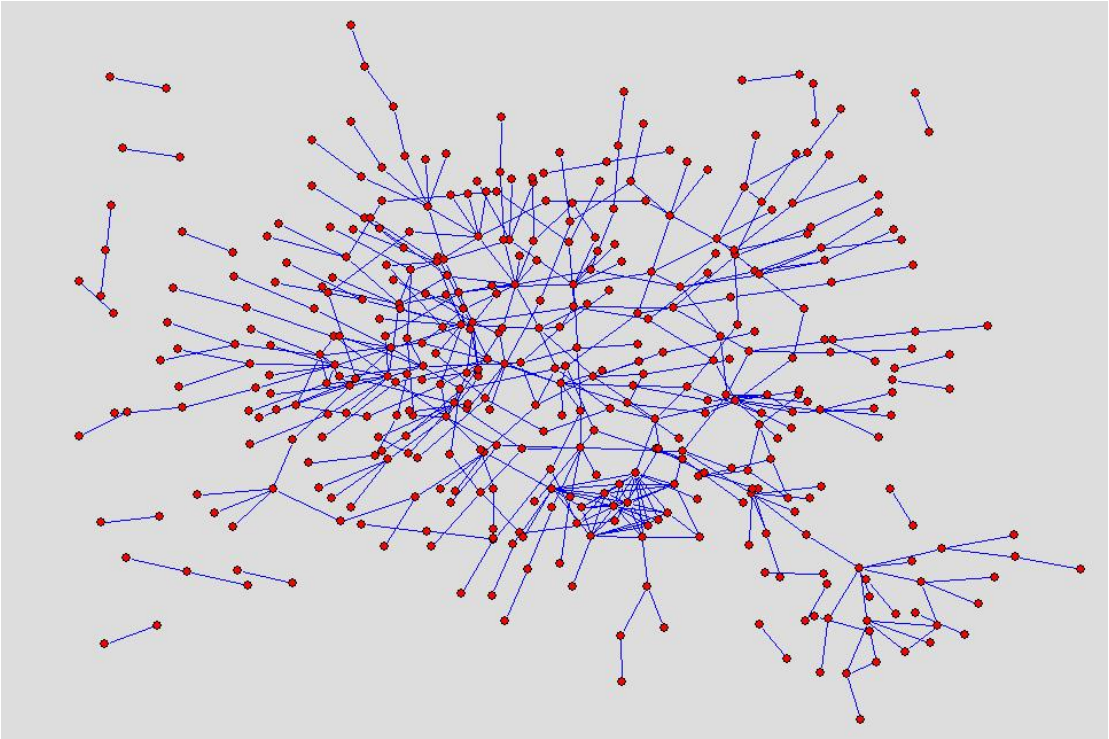
1）图的规模比较大，每张图大概包含数百至数千个顶点，更大规模的图甚至可以达到几万个顶点。这与当前所有频繁子图挖掘算法所处理的对象有显著的不同。（现有算法处理的图的规模最多只有数十个顶点。）

2）蛋白质相互作用网络图是稀疏图，平均而言每个蛋白质只与个位数的其他蛋白质发生作用，但是顶点的度数分布非常不均匀。

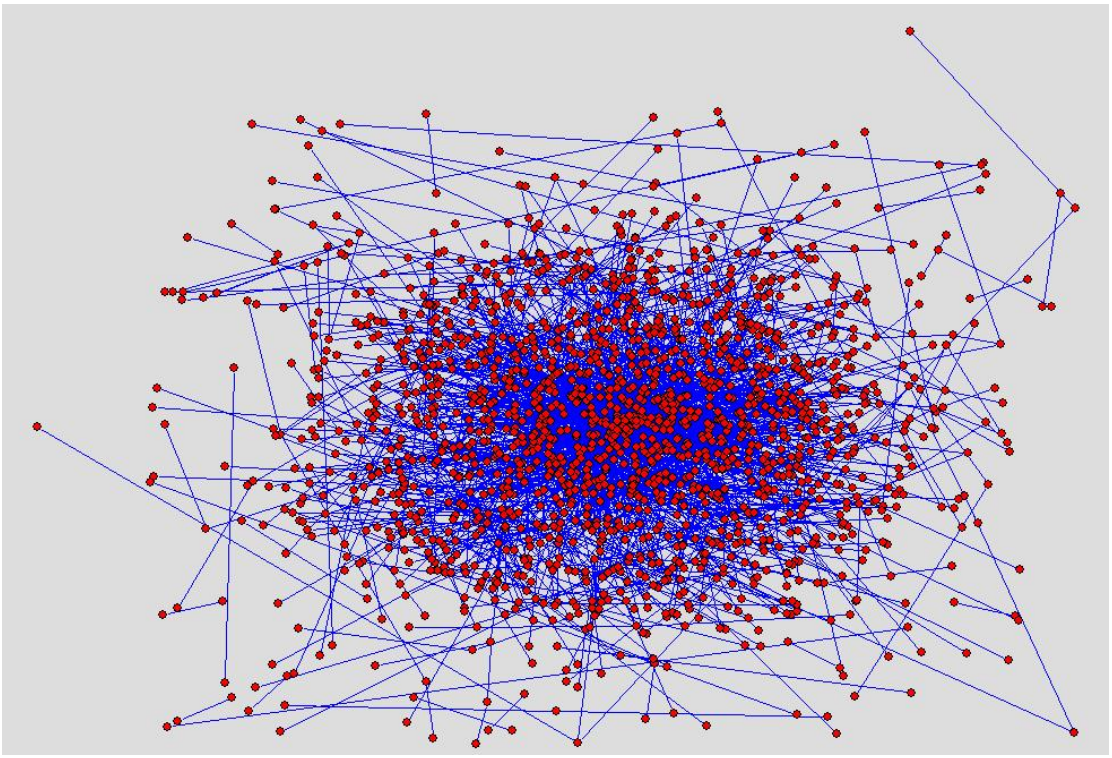
3）每张图的顶点是唯一标记的。（标记图的概念见 2.1 节）

这些特点使得在蛋白质相互作用网络上的频繁模式挖掘工作较之一般标记图

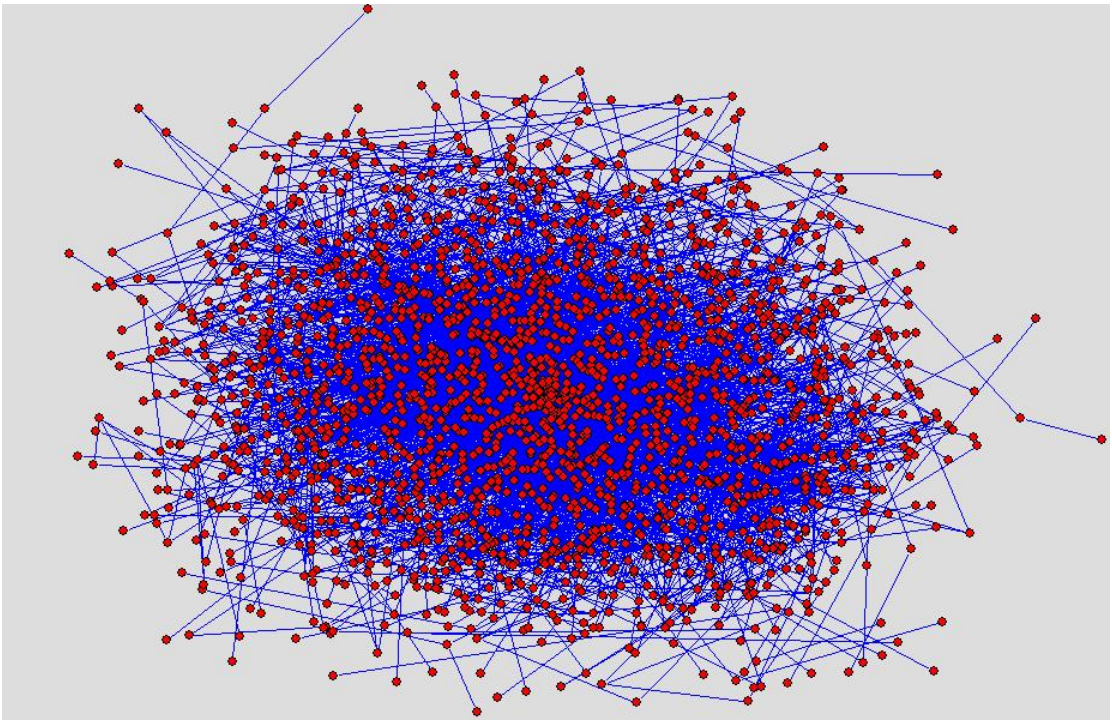
集上的频繁模式挖掘工作有了一些明显的区别，将对算法的设计、实现与性能产生深刻的影响，后面将会有比较详细的讨论。下面我们来看几张蛋白质相互作用网络图的实例，这样可以有一个比较直观的印象，从而对我们后面所要定义的问题的复杂性有一个更深刻的认识。



(a) Drosophila Melanogaster（果蝇），404 个顶点, 481 条边



（b）Caenorhabditis elegans（蛔虫），1722 个顶点, 2384 条边



（c）Saccharomyces Cerevisiae（酵母），2187 个顶点, 4837 条边

图 1 一些物种的蛋白质相互作用网络图

图 1 给出了一些真实物种的蛋白质相互作用网络图的实例。所有的数据均来自相互作用的蛋白质数据库（Database of Interacting Proteins,简称 DIP[2]）。DIP 数据库收集了由实验验证的蛋白质-蛋白质相互作用。数据库包括蛋白质的信息、相互作用的信息和检测相互作用的实验技术三个部分。用户可以根据蛋白质、生物物种、蛋白质超家族、关键词、实验技术或引用文献来查询 DIP 数据库。

为了方便起见，从第二章起我们将用 PPI(Protein-Protein Interaction)网络这一术语来指代蛋白质相互作用网络。

1.4 本文的研究对象和组织结构

本文研究的问题是如何从多张给定的蛋白质相互作用网络图中找到保守区域，也就是共有的子图。由于生物的进化过程中，会发生分支进化和基因复制等情况，精确的子图挖掘的约束太过严格，会导致大量有效结果的丢失，所以我们允许一定程度的不精确。事实上，我们所关心的保守区域正是前文所提到的图的拓扑结构。问题的正式定义见 3.1 节。

本文其余部分的组织结构如下：

第二章将介绍一些图的基本概念和其他预备知识，引入一些必要的记号。

第三章将对要研究的问题给出一个精确的定义，介绍一些这方面的相关工作。

第四章将给出解决问题的具体算法及有关实现。

第五章将给出有关实验的结果并作相应的分析。

第六章将给出算法在 Pathway 系统上的实际应用情况。

第七章将给出本文的总结以及对进一步的工作进行一下展望。

第二章 预备知识与符号定义

这一章介绍一些图论中的相关知识，并给出相应的符号定义，后文的论述将直接使用这些符号，不再进行说明。由于图论所涉及的概念非常多，所以我们假定读者对图论的基本知识已经有所了解，本章只提一些需要在后文中频繁使用的概念，其它一些概念将在后文需要时一并介绍。除非特别说明，本文所提到的图均指简单连通无向图。

2.1 图的一些基本概念

定义 1 标记图（Labeled Graph）

一张标记图可以用如下的五元组来表示：$G=(V, E, L, l_V, l_E)$，其中：

V 为图 G 的顶点（Vertex）集；

$E \subseteq V \times V$ 为图 G 的边（Edge）集；

L 是一个标记（Label，即字符串 String）的集合；

$l_V: V \rightarrow L$ 是从顶点集 V 到标记集 L 的映射，作用是给 G 的每个顶点赋上标记；

$l_E: V \rightarrow E$ 是从边集 E 到标记集 L 的映射，作用是给 G 的每条边赋上标记。

特别地，如果 l_E 为空，则称 G 为顶点标记图（Vertex Labeled Graph）；类似地，如果 l_V 为空，则称 G 为边标记图（Edge Labeled Graph）。如果 l_V 和 l_E 均为空，则事实上 G 是一张无标记图（Unlabeled Graph），但是我们也可以将无标记图视为标记图的一种特殊情况，即所有顶点（或边）都标有相同的标记。这将为我们处理某些问题提供方便。

在下文中，我们将用 $V(G)$来表示 G 的顶点集，$E(G)$来表示 G 的边集。

定义 2 路径（Path）

图 G 中的一个顶点序列$(v_1,v_2,\ldots,v_k)$称为图中的一条路径 p，如果满足：$v_i \in V(G)$，$(v_i, v_{i+1}) \in E(G)$。

顶点 v_1 和 v_k 称为 p 的端点（ends），其它的顶点称为 p 的内点（inner vertices），我们用 $IVS(p)$来表示由 p 的所有内点组成的集合。

特别地，如果 p 的所有顶点均不相同，则称 p 为一条简单路径（Simple Path），本文下面所提到的路径都是指简单路径。

此外，我们定义 p 的长度（Length）$|p|$为 p 中内点的数目，即$|p|=|IVS(p)|$。为了简便起见，我们将用 p^k 来表示一条长度为 k 的路径 p。

定义 3 路径独立性（Independent Paths）

设 p_1 和 p_2 是图 G 的两条路径，如果 $IVS(p_1)\cap IVS(p_2)=\varnothing$，我们称 p_1 与 p_2 是独立的（Independent），记为 $p_1\perp p_2$。进一步，设 p 是图 G 的一条路径，P 是 G 的一个路径集合，如果有 $IVS(p)\cap IVS(p_i)=\varnothing$，$\forall p_i\in P$，则称 p 与 P 是独立的。为了简便起见，我们称 p 是图 G 的一条独立路径（相对于集合 P），记为 $p\perp P_G$。独立路径将是我们刻画图的拓扑结构的有力工具。此外，若 G 的路径集合 P 中的路径两两之间都是独立的，则称 P 是 G 的一个独立路径集。

定义 4 子图（Subgraph）

设 G_1 和 G_2 是两张标记图， G_1 称为是 G_2 的一张子图，如果满足：$V(G_1)\subseteq V(G_2)$，并且 $E(G_1)\subseteq E(G_2)$。

特别地，如果 $\forall u, v\in V(G_1)$，$(u, v)\in E(G_2)\Rightarrow(u, v)\in E(G_1)$，则称 G_1 为 G_2 的 $V(G_1)$导出子图（Subgraph induced by $V(G_1)$）。

在下文中，如果 G_1 是 G_2 的一张子图，我们将记为 $G_1\subseteq G_2$。

2.2 图同构与子图同构

定义 5 图同构（Graph Isomorphism）

设 G_1 和 G_2 是两张标记图，如果存在双射 $\phi:V(G_1)\to V(G_2)$，满足：

1）$\forall u\in V(G_1)$， $l_{V(G_1)}(u)=l_{V(G_2)}(\phi(u))$；

2）$\forall(u, v)\in E(G_1)$，$(\phi(u),\phi(v))\in E(G_2)$，并且 $l_{E(G_1)}((u, v))=l_{E(G_2)}((\phi(u),\phi(v)))$。

则称 G_1 和 G_2 是同构的（Isomorphic），记为 $G_1\cong G_2$, 称 ϕ 为 G_1 和 G_2 之间的一个同构映射（Isomorphism）。特别地，如果 $G_1=G_2=G$，则称 ϕ 为 G 的一个自同构映射（Automorphism）。

定义 6 子图同构（Subgraph Isomorphism）

设 G_1 和 G_2 是两张标记图，如果存在 $G'\subseteq G_2$，并且 $G_1\cong G'$，则称 G_1 子图同构于 G_2，记为 $G_1\leq G_2$。G_1 和 G'之间的任一同构映射称为 G_1 到 G_2 的一个子图同构映射。

2.3 图的拓扑基与拓扑结构

这一节里我们首先考虑无标记图上的相关定义，然后推广到标记图上去。

2.3.1 图的拓扑基

直观上来说，图G的一个拓扑基T是通过将G的一张子图中的独立路径集中的路径压缩为边而得到的。在给出正式的定义之前，让我们先来看一个图的拓扑基的具体例子：

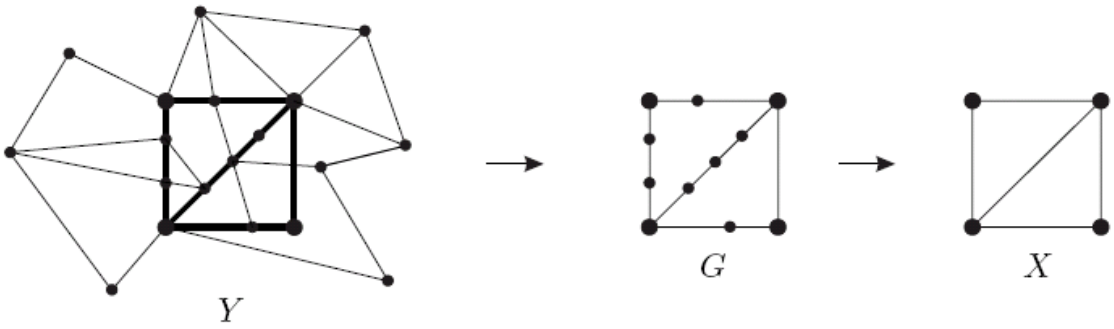


图 2 图的拓扑基实例

如图 2 所示，图 X 是图 Y 的一个拓扑基，因为 X 可以通过将图 G 中的独立路径集中的路径压缩为边来得到，而 G 是 Y 的一张子图。下面给出图的拓扑基的正式定义：

定义 7 图的子分割操作（Subdivision Operation）

设 X 是一张无标记图，对 X 的子分割操作定义为：$\forall e \in E(X)$，将 e 替换为一条路径 p，记为 $e \to p$；定义路径集合 $P=\{p_i \mid e_i \to p_i$ 且 $e_i \in E(X)\}$，则上述一系列替换操作必须保证 P 是独立路径集。

称对X进行子分割操作后得到的图为X的一张子分割图（Subdivision Graph）。

例如，在图 2 中，图 G 即为图 X 的一张子分割图。

事实上，子分割操作和我们前面所说的路径压缩操作互为逆操作。

定义 8 图的拓扑空间（Topological Space）

设 X 是一张无标记图，由 X 的所有子分割图组成的集合，称为 X 的拓扑空间，记为 $T(X)$。

定义 9 图的拓扑基（Topological Minor）

设 X 和 Y 为无标记图，如果存在 X 的一张子分割图 G（$G \in T(X)$），满足 $G \subseteq Y$，则称 X 是 Y 的一个拓扑基。X 的顶点称为分枝顶点（Branch Vertices）。$\forall x \in V(X)$，

记 Y 中对应于 x 的顶点为 $Occ_{X,G,Y}(x)$。

2.3.2 图的拓扑结构

定义 10 图的*(l,h)*-子分割操作（*(l,h)* -Subdivision Operation）

设 l, $h \in N$，（N 为自然数集），$0 \le l \le h$， X 是一张无标记图，对 X 的*(l,h)*-子分割操作定义为： $\forall e \in E(X)$，将 e 替换为一条路径 p，满足 $l \le |p| \le h$，记为 $e \xrightarrow{(l,h)} p$；定义路径集合 $P=\{p_i \mid e_i \xrightarrow{(l,h)} p_i$ 且 $e_i \in E(X)\}$，则上述一系列替换操作必须保证 P 是独立路径集。

称对 X 进行 *(l,h)*-子分割操作后得到的图为 X 的一张 *(l,h)*-子分割图（*(l,h)*-Subdivision Graph）。

例如，在图 2 中，G 即为 X 的一张(0, 3)-子分割图。

定义 11 图的*(l,h)*-拓扑空间（*(l,h)*-Topological Space）

设 X 是一张无标记图，由 X 的所有*(l,h)*-子分割图组成的集合，称为 X 的*(l,h)*-拓扑空间，记为 $T_{l,h}(X)$。

定义 12 图的拓扑结构（Topological Structure）

设 X 和 Y 为无标记图，如果存在 X 的一张*(l,h)*-子分割图 G（$G \in T_{l,h}(X)$），满足 $G \subseteq Y$，则称 X 是 Y 的一个*(l,h)*-拓扑基（*(l,h)*-Topological Minor），或者称 X 为 Y 的一个拓扑结构，记为 $X \le_{(l,h)} Y$。

例如，在图 2 中，X 是 Y 的一个(0, 3)-拓扑基。

引入图的拓扑结构概念的目的在于控制子分割操作的粒度，从而控制产生的子分割图的规模。换个角度说，在后文我们讨论 PPI 网络上的频繁拓扑结构挖掘时，数据图中对应于某个拓扑结构的嵌入子图（Embedding Subgraph，例如在图 2 中，G 即为 Y 中对应于拓扑结构 X 的一张嵌入子图，在第四章里我们将定义一个专门的术语——出现（Occurrence）来指代数据图中对应于拓扑结构的嵌入子图）将被控制在一定的规模之内，具体而言，我们有如下的简单性质：

性质 1 若 G 是 X 的一张 *(l,h)*-子分割图，则 G 的顶点数$|V(G)|$和边数$|E(G)|$满足以下约束条件：

$|V(X)|+l\times|E(X)| \le |V(G)| \le |V(X)|+ h\times|E(X)|$

$(l+1)\times|E(X)|\le|E(G)|\le(h+1)\times|E(X)|$

证明：由*(l,h)*-子分割图的定义即知。

下面再给出图的拓扑结构的两条重要性质，它们的证明都可以直接从上述相关的定义中导出。

性质 2 若 X 是 G 的一个*(l,h)*-拓扑基，则$\forall l'$，h'，$0\le l'\le l$，$h\le h'$，X 是 G 的一个 *(l',h')*-拓扑基。

性质 3 设 X 是一张无标记图，则$|T_{l,h}(X)|\le(h-l+1)^{|E(X)|}$。

2.4 图的拓扑基和拓扑结构概念在标记图上的推广

由于本文后面的部分只涉及顶点标记图，为了简化我们的讨论，以下我们只考虑顶点标记图情况下上述相关定义的推广。后文中凡是涉及标记图的地方都是指顶点标记图。

定义 13 顶点标记图的*(l,h)*-子分割操作

设 l，$h\in N$，（N 为自然数集），$0\le l\le h$，X 是一张顶点标记图，对 X 的 *(l,h)*-子分割操作定义为，

$\forall e\in E(X)$，将 e 替换为一条路径 p，同时需要满足以下条件：

1）$l\le|p|\le h$；

2）设 v_1，v_2 为 p 的两个端点，L 是一个标记集且$\{l_{V(X)}(v_1), l_{V(X)}(v_2)\}\subseteq L$，定义 $l_{V(p)}: V(p)\to L$，满足：$l_{V(p)}(v_1)=l_{V(X)}(v_1)$，$l_{V(p)}(v_2)=l_{V(X)}(v_2)$，并且$\forall v\in IVS(p)$，$\exists l\in L$，使得 $l_{V(p)}(v)=l$。

记为 $e\xrightarrow{(l,h)}(p, l_{V(p)})$。定义路径集合 $P=\{p_i\mid e_i\xrightarrow{(l,h)}(p_i, l_{V(p_i)})$且 $e_i\in E(X)\}$，则上述一系列替换操作必须保证 P 是独立路径集。

其它的概念，包括*(l,h)*-子分割图，*(l,h)* -拓扑空间，以及*(l,h)*-拓扑基（拓扑结构），在标记图的情况下和无标记图情况下的定义是一样的，这里不再赘述。

最后我们来看一个标记图情况下图的拓扑基的实例。

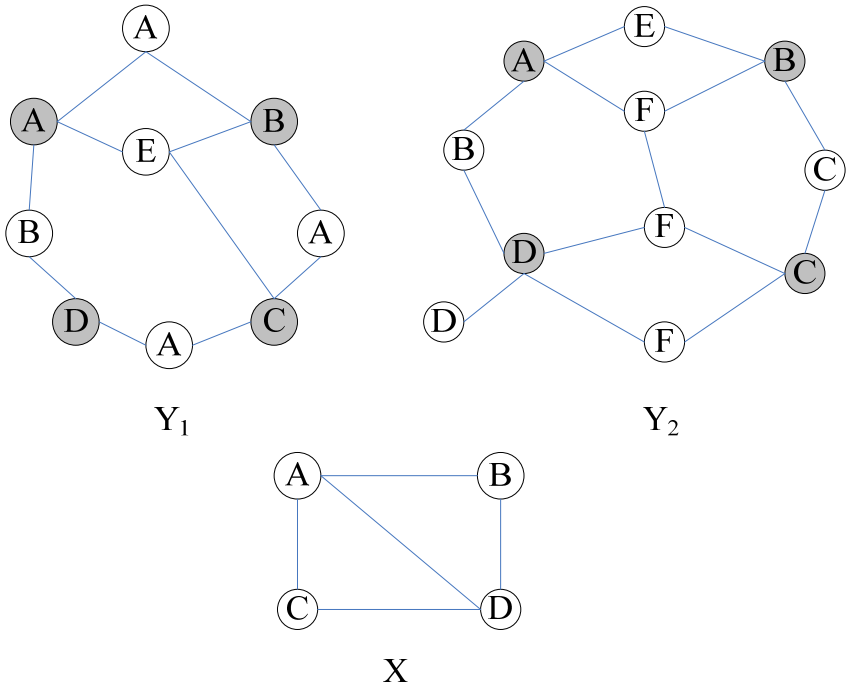


图 3 标记图的拓扑基实例

如图 3 所示，Y_1 和 Y_2 是两张标记图，则 X 是 Y_1 的一个(1, 1)-拓扑基，同时也是 Y_2 的一个(1, 2)-拓扑基。

第三章 问题描述与相关工作

这一章中，我们将给出本文所要处理的问题的一个正式的定义，在此基础上，我们将对问题进行深入的分析并简单介绍一些相关的工作。

3.1 问题描述

在 1.4 中我们已经提到，本文要解决的问题是如何从多张给定的 PPI 网络图中找到保守区域，也就是共有的子图。但是我们所要寻找的子图并不是严格的，因为精确的子图挖掘可能会丢失大量的有效信息。在第二章介绍有关概念和性质的基础上，我们现在可以对问题来进行一个准确的定义了。

3.1.1 PPI网络的建模：

从现在开始，我们将每一张给定的 PPI 网络图抽象为一张顶点标记图，图的每个顶点表示一个蛋白质，顶点的标记定义为对应的蛋白质的标记（我们前面已经提到，每张 PPI 网络图中蛋白质都是唯一标记的）。如果两个蛋白质之间存在相互作用，则在对应的两个顶点之间用一条边相连接。这里我们忽略各种蛋白质相互作用之间的差异，因此对应的边是不带标记的（也可以认为所有边的标记都相同）。

3.1.2 从PPI网络图集中寻找频繁拓扑结构：

我们所要寻找的保守区域，正是我们在第二章中所定义的拓扑结构。由于前一节中我们已经说明 PPI 网络可以建模为顶点标记图，所以问题首先转化为如何从给定的顶点标记图集中寻找频繁拓扑结构。下面我们先给出一般的顶点标记图集上频繁拓扑结构挖掘问题的形式化定义，然后基于 PPI 网络的实际背景作一些必要的说明。

定义 14 拓扑结构在给定图集中的支持度（Support）

设 D 是一个给定的顶点标记图的集合，l，h 是两个给定的自然数，$0 \le l \le h$，G 是一张给定的顶点标记图，定义集合 $S_D(G)=\{Y \mid G \le_{(l,h)} Y, Y \in D\}$，称 $S_D(G)$的基

数 $s=|S_D(G)|$为 G 在 D 中的支持度（即 D 中包含拓扑结构 G 的图的数目）。

定义 15 频繁拓扑结构（Frequent Topological Structure）

设 D 是一个给定的顶点标记图的集合，l，h 是两个给定的自然数，$0\le l\le h$。给定一个阈值 θ，设 G 是一张标记图，若满足$|S_D(G)|\ge\theta$，则称 G 是 D 的一个频繁拓扑结构。

定义 16 从给定的顶点标记图集中挖掘频繁拓扑结构

设 D 是一个给定的顶点标记图的集合，l，h 是两个给定的自然数，$0\le l\le h$，给定一个阈值 θ，则从 D 中挖掘频繁拓扑结构可以定义为：

找出集合 $FreTS(D)=\{G|\ |S_D(G)|\ge\theta\}$。

下面我们将就 PPI 网络挖掘的实际背景进行一些说明，然后给出 PPI 网络上频繁拓扑结构挖掘问题的正式定义。

1）虽然我们将 PPI 网络抽象为顶点标记图，但是我们前面已经提到过，这些图中顶点都是唯一标记的。事实上，设 G_1 和 G_2 分别代表两张 PPI 网络对应的顶点标记图，则若定义 $L_{V(G)}=\{l\mid l\in L,\exists v\in V(G), l_{V(G)}(v)=l\}$，则有 $L_{V(G_1)}\cap L_{V(G_2)}=\varnothing$，也就是说 G_1 和 G_2 中没有相同标记的顶点。因此，我们无法直接将定义 16 移植到 PPI 网络的挖掘上来，否则我们必然无法得到任何结果。

2）我们现在来考察一下两个顶点具有相同标记的含义是什么。我们认为，标记代表了顶点的某种属性信息，如果我们可以给两个顶点赋予同一个标记，则这两个顶点在该种属性上是等价的。反之，如果两个顶点在该种属性上等价，则我们认为这两个顶点可以被赋予相同的标记。如果我们能够给两个顶点赋予相同的标记，则称这两个顶点是可匹配的（Matched），否则称它们是失配的（Unmatched）。设 G_1 和 G_2 分别代表两张 PPI 网络对应的顶点标记图，我们将原有的顶点标记信息视为对该顶点的一个标识，换言之，该顶点的原有的标记所起的作用只是给该顶点一个名称以便我们可以指出它，并不带有任何的属性信息。因此，从属性等价的意义上说，我们现在可以将 G_1 和 G_2 视为无标记图。

3）所以，从 PPI 网络图集中挖掘频繁拓扑结构事实上可以视为从无标记图集中挖掘频繁拓扑结构。但是，正如我们前面已经提到过的那样，无标记图可以视为标记图的一种特殊情况，即所有顶点都带有相同的标记，因此我们前面基于标记图的环境所作的关于频繁拓扑结构的那些定义在无标记图的环境下仍然成立。

另一方面，在无标注图的环境下，出现了新的问题。由于所有的顶点都视为具有相同的标记，因此任意两个顶点都被视为是可匹配的。而我们在 1.3.2 中已经指出，PPI 网络图的规模一般都比较大，如果我们不对顶点之间的匹配条件进行一些约束的话，实际进行挖掘时的计算代价将无法想象。事实上，在生物进化的意义下，PPI 网络中的保守子结构不仅要求相应的嵌入子图在结构上具有相似性，而且要求构成该结构的分枝顶点所对应的蛋白质之间也具有同源相似性。生物学上一般用氨基酸序列来表示蛋白质，通过比对氨基酸序列来确定对应蛋白质之间的同源相似性。目前对生物序列比对的研究已经相对成熟，产生了不少进行序列比对的工具。这里我们采用 BLAST[3]，它是目前最为常用的用来计算蛋白质同源相似性的工具。关于 BLAST 的使用和具体的数据预处理过程将在第六章介绍。我们现在需要了解的只是：

设 G_1 和 G_2 是给定的两张 PPI 网络图，通过 BLAST 我们可以得到 G_1 中的每个蛋白质和 G_2 中相应蛋白质之间的同源相似度列表（简称评分表），记为 $SL(G_1,G_2)=\{(v_1,v_2,score(v_1,v_2)) \mid v_1\in G_1,\ v_2\in G_2,\ score(v_1,v_2)\in I^+\}$，其中 I^+指正整数集。

设 $v_1\in G_1$，$v_2\in G_2$，则只有当$\exists n\in I^+$，使得$(v_1,v_2,n)\in SL(G_1,G_2)$时，我们才认为 v_1 和 v_2 是可匹配的，否则我们认为 v_1 和 v_2 是不匹配的。这样做能够有效地减少可能的匹配顶点对数。进一步，我们可以根据挖掘的粒度在 $SL(G_1,G_2)$上进行过滤来得到不同粒度下的评分表。具体地说，设δ为给定的阈值，则我们可以得到 G_1 和 G_2 的同源相似度至少为δ 的评分表，记为

$SL(G_1,G_2,\delta)=\{(v_1,v_2,score(v_1,v_2)) \mid (v_1,v_2,score(v_1,v_2))\in SL(G_1,G_2),$
$score(v_1,v_2)\geq\delta\}$，称为 G_1 和 G_2 的δ-评分表。

4）由于我们所关心的保守子图结构（拓扑结构）需要在给定的 PPI 网络图集 D 的每一张图中出现，所以一般来说，我们设定所要寻找的拓扑结构的支持度为 $|D|$。

基于上述几点认识，我们在这一节的最后给出我们所要处理的问题的正式定义：

定义 17　从给定的 PPI 网络图集中挖掘频繁拓扑结构

设 D 是一个给定的 PPI 网络图集，l，h 是两个给定的自然数，$0\leq l\leq h$，给定阈值$\theta=|D|$，δ，则从 D 中挖掘频繁拓扑结构可以定义为：

找出集合 $FreTS(D)=\{X \mid |S_D(X)|=\theta, X$ 为无标记图且$\forall v\in V(X),\ \forall Y_i,Y_j\in D,\forall G_m,G_n$

$\in T_{l,h}(X), G_m \subseteq Y_i, G_n \subseteq Y_j, \exists l_{mn} \in I^+, (Occ_{X,G_m,Y_i}(v), Occ_{X,G_n,Y_j}(v), l_{mn}) \in SL(Y_i, Y_j, \delta)\}$。

3.2 问题的复杂性

频繁拓扑结构挖掘问题是对频繁子图挖掘问题的一种推广。换言之，在我们已经定义了频繁拓扑结构挖掘问题之后，频繁子图挖掘问题就可以视为前者的一个特例，因为我们可以将图的一张子图视为图的一个(0,0)-拓扑基。

直观上说，设 $D_i \in D$，G 是一个候选的*(l,h)*-拓扑基，判断是否成立 $G \leq_{(l.h)} D_i$ 可以分为两步：先对 G 进行*(l,h)*-子分割操作，然后判断得到的*(l,h)*-子分割图是否是 D_i 的子图。直观方法显然是非常低效的，但是能使我们对问题的复杂性有一个深刻的印象。

首先，抛开第二步的开销（第二步事实上需要进行子图同构判断，我们马上会讨论这个问题）不谈，由性质 3 我们可以知道，对 G 进行*(l,h)*-子分割操作后得到的子分割图有$(h-l+1)^{|E(G)|}$张，这里还没有考虑标记图的情况，而实际对 PPI 网络图进行*(l,h)*-子分割操作是需要考虑顶点的标记的，设 $L(D_i)$表示所有在 D_i 中出现的顶点标记的集合，则实际需要考虑的子分割图的数目为 $N = \sum_{i=1}^{(h-l+1)^{|E(G)|}} C_{|L(D_i)|-|V(G)|}^{|V(X_i)|-|V(G)|}$，$X_i \in T_{l,h}(G)$。（PPI 网络图中的顶点是唯一标记的）。假设 l=1,h=2,$|E(G)|$=5,$|V(G)|$=4,$|L(D_i)|$=100，则 $N = \sum_{i=1}^{32} C_{|L(D_i)|-|V(G)|}^{|V(X_i)|-|V(G)|} \geq \sum_{i=1}^{32} C_{|L(D_i)|-|V(G)|}^{|V(G)|+l\times|E(G)|-|V(G)|} = \sum_{i=1}^{32} C_{96}^{5} = 1,955,970,048$。也就是说，要判断一个顶点数为 4，边数为 5 的图 G 是否是一张包含 100 个蛋白质的 PPI 网络的(1, 2)-拓扑基，如果采用上面的直观方法，最坏情况下我们需要考察的(1, 2)-子分割图有 20 亿张之巨。事实上，这些子分割图在实际中只可能有极少数会真正出现，因此，如何减少子分割图的枚举数量成为我们设计算法时一个需要重点考虑的问题。

下面我们再来介绍一下上述直观方法中的第二步的开销。上面已经提到，第二步事实上要进行的工作是要判断得到的*(l,h)*-子分割图是否子图同构于数据图 D_i。然而，很遗憾的是，子图同构判断已经在理论上被证明是 NPC 问题[4]。这意味着在一般情况下，我们是无法找到很有效率的子图同构算法的。因此，我们所能做的，只能是尽可能地减少需要进行子图同构判断的次数或者干脆避免它。

从上面的讨论中，我们应该已经体会到我们所要处理的问题的复杂程度了。本文的目的，就是希望找到一个具有实际应用价值的算法来解决这个问题。（具有实际应用价值是指空间、时间开销在实际应用中处于一个可以接受的范围）下一节先介绍一些这方面的相关工作，我们将在下一章中详细介绍我们的算法。

3.3 相关工作

在这一节里，我们将介绍一些图数据集上进行频繁模式挖掘的相关工作。这里的模式包括子图和拓扑结构。

3.3.1 频繁子图挖掘的相关工作

第一章中已经提到过，频繁子图挖掘是近年来图数据集上频繁模式挖掘研究的一个热点。频繁子图的定义类似于我们前面频繁拓扑结构的定义：设 D 是一个给定的（标记）图集，θ 为给定的阈值，G 是给定的一张（标记）图，定义集合 $Supp_D(G)=\{D_i \mid G \le D_i, D_i \in D\}$，若成立 $|Supp_D(G)| \ge \theta$，则 G 是 D 的频繁子图。

目前已有的频繁子图挖掘算法根据模式（子图）枚举的方式大致可以分为两大类：

1）宽度优先的模式增长方式：

这类算法按照子图的大小（边数）从小到大地进行模式枚举，具体的做法一般为：

首先找出数据集中所有大小为 1 的频繁子图（只有一条边），然后将这些子图两两连接（Join）得到边数为 2 的候选子图，到数据集中去统计这些子图的出现频次（这里需要涉及子图同构检测），从而得到大小为 2 的频繁子图（两条边）。然后对大小为 2 的频繁子图重复上述的处理过程。一般地，我们通过对大小为 k 的频繁子图进行两两连接得到大小为 $k+1$ 的候选子图，然后通过统计这些候选子图的出现频次来得到大小为 $k+1$ 的频繁子图。

由于统计候选子图在数据集中的出现频次需要对数据集中的每一张图进行子图同构检测，因此我们需要尽可能地缩小每次得到的候选子集的规模。这里一般在连接操作时使用 Apriori 性质进行剪枝（Pruning）。Apriori 性质最早是由 Agrawal 等在 94 年提出的[5]，当时被应用于在事务数据库(Transaction Database)上挖掘频繁项集(Frequent Itemsets)，现在它已经被引入到频繁子图挖掘中来。在图数据的环境下，Apriori 性质可以被表述为：若 G 是图数据集 D 的一张频繁子图，则 G 的所有子图也是频繁的。若 G 不是频繁的，则 G 的任何父图也不是频繁的。（若 $G \subseteq G'$，则称 G' 为 G 的父图）因此，当我们得到一张规模为 $k+1$ 的候选子图时，我们可以通过检测它的每张规模为 k 的子图（共有 $k+1$ 张）是否在规模为 k 的频繁子图集中出现来决定其取舍。（这里需要涉及图同构检测，关于图同构检测的问题我们将在下一章讨论到算法相关部分时再进行详细说明。另外，当决定两个规模为 k 的模式是否可以连接为一个规模为 $k+1$ 的模式时，我们同样需要通过图

同构检测来判断这连个模式是否具有规模为 *k-1* 的公共子图。）

宽度优先模式增长方式的主要优点是模式是从小到大地进行增长，为利用 Apriori 性质进行剪枝创造了条件。但是宽度优先增长方式也存在着不少自身无法克服问题，主要有：

（1）在每一阶段（特别是中间阶段）都需要维护一个巨大的候选子图集；

（2）每一阶段进行频繁子图计数时，都需要重新遍历数据集，当数据集驻留在磁盘上时需要频繁地访问磁盘。当然，现在的算法大多通过记录前一阶段的匹配情况来减少下一阶段的访问代价。但是这一方面增加了大量的内存开销，而且当候选集的规模非常大时，这一优化的实际对性能的改进效果不明显。

（3）很难找到规模很大的模式。在实际应用中，宽度增长方式很难进入到较大模式增长的阶段，这是因为之前巨大的空间和时间代价早已使我们无法忍受。

采用宽度优先模式增长方式的典型算法是 FSG[6]。更早的算法是 AGM[7]，但是它采用的是基于顶点的模式增长方式而不是基于边的模式增长方式（即模式增长是将两个 *k* 个顶点的模式连接为一个 *k+1* 个顶点的模式）。

2）深度优先的模式增长方式：

这类算法和宽度优先的算法在思想上明显不同，其具体做法一般为：

首先同样找出数据集中所有大小为 1 的频繁子图。（只有一条边，一般子图的规模仍然用边数来度量）然后，从每个规模为 1 的频繁子图（频繁边）出发，递归地进行模式增长。设开始进行增长的频繁边为 *e*，当前的递归深度为 *k*，即我们已经从一条频繁边 *e* 出发扩展到了规模为 *k* 的频繁子图。我们然后通过往每个规模为 *k* 的频繁模式中加入一条可以扩展的频繁边的方式来得到一个候选的 *k+1* 大小的模式，然后我们统计该模式在数据集中的出现频次即可。注意，这里的频次统计不再需要重新遍历数据集，而只需要根据对应的规模为 *k* 的模式在数据集中的出现情况进行进一步的检测即可。这是深度优先扩展方式相对于宽度优先扩展方式的一大优点。

深度优先扩展方式的另一个优势在于可以使我们尽早地发现那些与已经被挖掘出来的某个模式同构的模式，从而进行剪枝来避免大量的重复工作。如果我们把整个搜索过程设想为对一棵状态搜索树的访问的话，那么当我们访问搜索树的一个节点发现该节点对应的模式已经被找到时，我们就可以不再访问以该节点为根的子树，而是直接回溯（Backtracking）到当前节点的父节点，访问其下一个子节点。

深度优先扩展方式还能有助于我们更早地发现规模较大的模式，有助于我们将相关算法只进行较小的修改就可应用到其他带有约束条件的挖掘问题上去。

相关的实验结果表明，采取深度优先模式增长方式的算法在性能上要明显优

于采取宽度优先模式增长方式的算法。但是深度优先算法也不是完美的，主要的问题是同一个模式可能会被多次重复地枚举到，虽然我们前面说可以通过剪枝和回溯的策略来尽可能地减少枚举次数，但是当模式本身的规模较大时，以上策略的效果是有限的。换句话说，对于规模较大的模式而言，有许多种枚举顺序最终可能都指向同一模式，但是我们只能在搜索树的较深层次才能发现这种同构性，这会带来大量的重复工作。特别是当图集中大规模的模式较多时，算法的性能就会受到很大的影响。而在这种情况下，宽度优先的增长方式反而不会受到太大影响，因为宽度优先的增长方式可以保证每个模式只可能在本阶段被重复枚举到，而不会在下一阶段被重复枚举，所以同一模式被重复枚举到的可能性要比深度优先方式下小。但是前面已经提到，宽度优先方式在实际中又很难进入到较大模式增加的阶段，这是一个非常矛盾的地方。

采用深度优先的模式增长方式的典型算法有 gSpan[8],FFSM[9],Gaston[10]等。

必须指出的是，当数据集本身包含规模很大的频繁子图时，无论采用宽度优先还是深度优先的模式枚举方式，所需的时空代价都是非常巨大的。这是因为任何频繁子图的子图都是频繁的，假设数据集中包含一张 100 条边的频繁子图，则它的 2^{100}-1 张子图都是频繁子图！

3.3.2 频繁拓扑结构挖掘的相关工作

到目前为止，对于频繁拓扑结构挖掘的研究只能说尚处于起步阶段。TSMiner[11]是我们找到的唯一一篇专门论述频繁拓扑结构挖掘工作的文章。这篇文章提出的算法将整个挖掘过程划分为两个阶段，第一个阶段产生所有的频繁树模式，（树的有关概念我们将在下一章阐述算法时一并说明）第二个阶段以第一阶段找到的树模式作为生成树来寻找相应的图模式。

这个两阶段挖掘的算法框架并非 TSMiner 的首创，它的思想来源于 Gaston 算法[10]。Gaston 算法是一个进行频繁子图挖掘的算法，TSMiner 的工作相当于把这一框架推广到了频繁拓扑结构的挖掘上。Gaston 的算法思想则是基于以下的实验观察结果：在小分子数据库上的实验表明，该类数据库中绝大部分的频繁子图模式事实上是一种自由树（free tree，我们将在下一章中详细介绍）的形式。自由树相对于有环(回路)图而言是一种相对简单的结构，存在着许多针对于自由树结构的高效算法。因此，如果将频繁子图的挖掘分阶段进行，可以针对每个阶段子图结构上的特点采用不同的算法，从而提高整个挖掘过程的效率。另一方面，

采用分阶段挖掘可以更便于我们控制挖掘过程的进度和粒度，从而得到最符合我们实际需要的模式。需要指出的是，分阶段挖掘算法的本质仍然是采用深度优先的模式增长方式的，只是对模式的增长在不同阶段采取了不同的约束而已。

在下一章中，我们将给出在 PPI 网络上进行频繁拓扑结构挖掘算法的具体设计和实现。我们仍然采用上述分阶段挖掘的思想作为算法总的框架，另外，我们借鉴了 TSMiner 中提出的不少想法。但是，PPI 网络上频繁拓扑结构的挖掘与一般的标记图上频繁拓扑结构的挖掘还是有许多不同点的，这些不同之处我们在 3.1 节定义问题时已经作了详细的说明。后面我们将看到，我们针对 PPI 网络挖掘的特点对 TSMiner 的算法框架进行了不少修改和改进，形成了自己的算法。

必须要指出的是，到目前为止，我们还不能确定分阶段挖掘的算法框架是否能真正提高 PPI 网络挖掘的效率，因为我们现在并不能确定 PPI 网络图集是否具有和小分子数据库相同的特点。(Gaston 和 TSMiner 的实验数据集都可以视为小分子数据库，并且图的规模都比较小，这些与 PPI 网络图具有显著的差异。)但是，从 PPI 网络的稀疏性以及图 1 对真实 PPI 网络图的刻画我们可以预测 PPI 网络数据集也具有类似的性质，即频繁模式大多为自由树的形式。当然，这有待于实验的进一步验证，我们将在第五章介绍实验结果时再来讨论这个问题。

第四章 算法设计与实现

在这一章里，我们分两个部分对算法进行讨论。

在第一部分(4.1 节和 4.2 节)，我们将给出一般的标记图集上频繁拓扑结构挖掘的详细算法。我们首先介绍算法的一些关键步骤及其实现，然后我们会给出整个算法的详细框架。在论述的过程中，我们会适时地引入一些必要的概念和结论。

在第二部分(4.3 节和 4.4 节),我们针对 PPI 网络频繁拓扑结构挖掘问题的特殊性对第一部分给出的算法框架进行改进，并对最终得到的算法的时间代价作一个分析。

4.1 算法的一些关键步骤及其实现

在上一章的最后，我们已经提到我们的算法将沿用 Gaston 和 TSMiner 的分阶段挖掘的思想。我们首先挖掘出那些频繁的树（自由树）模式，然后以这些树模式作为生成树来进一步挖掘图（有环图）模式。为了后面论述的方便，我们先引入有关概念并进行符号定义。

定义 18 回路（Cycle），树（Tree）与自由树（Free tree）

设 G 一张连通的（标记）图，p 是 G 中的一条简单路径，设 p 的顶点序列为 $(v_1,v_2,\ldots,v_m)$，若满足 $v_1=v_m$，则称 p 为 G 的一条回路，一般我们用 c 来表示回路。若 G 中没有回路，则称 G 为树，为了和下面将要定义的有根树进行区别，又称这里定义的树为自由树。换句话说，（自由）树就是连通的无回路的图。一般我们用 T 来表示树。

定义 19 顶点的度数（Degree），出度（Outgoing Degree）与入度（Incoming Degree）

设 G 为无向图，$v\in V(G)$，定义集合 $N_G(v)=\{u \mid (u,v)\in E(G)\}$，称 $N_G(v)$为 v 的邻域。称$|N_G(v)|$为 v 的度数，记为 $d(v)$。$\forall u\in N_G(v)$，称 u 为 v 的邻居。进一步，若 G 为有向图，定义集合 $N_G^+(v)=\{u \mid <v,u>\in E(G)\}$， $N_G^-(v)=\{u \mid <u,v>\in E(G)\}$。（注记：本文中，我们用$(u,v)$表示无向边，用$<u,v>$表示有向边。）称$|N_G^+(v)|$为 v 的出度，记为$d^+(v)$；$|N_G^-(v)|$为 v 的入度，记为$d^-(v)$。

定义 20 有向树（Directed Teee)与有根树（Rooted Tree）

设G 是一张有向图，若忽略 G 中弧的方向时 G 是一棵树，则称 G 为有向树。

若一棵有向树恰有一个顶点的入度为 0，其余所有顶点的入度均为 1，则称该有向树为有根树。入度为 0 的点称为有根树的根。出度为 0 的顶点成为有根树的叶节点，其它出度非 0 的顶点称为分枝节点。有根树还有一些专门的术语，一并介绍如下：

设 T 为有根树，r 为 T 的根，u，w，x，$z \in V(T)$。设 u 是 T 的分枝节点，若从 u 到 w 有一条弧$<u,w>$，则称 w 为 u 的孩子（Child），称 u 为 w 的父亲（Parent）。若 x 和 w 有共同的父亲 u，则称 x 为 w 的兄弟（Sibling）若从 u 到 z 有一条有向路，则称 z 为 u 的子孙（Descendant），称 u 为 z 的祖先（Ancestor）。设 p 为从 r 到 u 的路，定义 u 的层数（Level）为 $l(u)=|p|+1$，特别地，$l(r)=0$。称 T 中节点的最大层数为 T 的高度（Height），记为 $h(T)=max\{\, l(u) \mid u \in V(T)\}$。

定义 21　图的生成树（Spanning Tree）

设 G 一张连通的（标记）图，若树 T 满足 $T \subseteq G$，且 $V(T)=V(G)$，则称 T 为 G 的一棵生成树。

在这一节的余下部分，我们将讨论整个算法中一些关键的步骤并给出我们的实现。

4.1.1 拓扑结构的表示方式

首先，我们需要对拓扑结构有一个合适的表示方式，这时我们定义后续一切操作的基础。我们在设计拓扑结构的表示方式时，主要要考虑三个问题：该表示方式是否便于进行模式增长，是否便于统计模式在数据集中的出现频次，其空间效率如何。在这一小节里我们先给出我们的表示方式，再结合这一表示方式对这三个问题作一个简单的说明。

定义 22　拓扑结构在数据图中的出现（Occurrence）

给定参数 l，h，图 G，设 Y 是数据集 D 中的一张图，　若 $G \leq_{(l,h)} Y$，则我们知道，存在图 X，满足 $X \subseteq Y$ 且 $X \in T_{l,h}(G)$，称 X 为 G 在 Y 中的一个出现。我们记由 G 在 Y 中的所有出现所组成的集合为 $Occ_G(Y)$。

我们先将拓扑结构 G 表示成由它的边构成的集合，即：$G \triangleq \cup_{i=1}^{k}\{e_i\}$，$k=|E(G)|$，$e_i \in E(G)$，$\forall\, i \in \{1,2,\ldots,k\}$。根据前面对子操作图的定义，$G$ 的每一条边 e_i 对应于 X 中的一条路径 p_i，$l \leq |p_i \leq |h$ 且 $P = \cup_{i=1}^{k}\{p_i\}$ 是独立路径集。为了明确这

种对应关系，我们用$\vec{e_i}$来代替p_i。因此，X可以被表示为由G中的边所对应的独立路径$\vec{e_i}$所构成的集合，即：$X \triangleq \cup_{i=1}^{k}\{\vec{e_i}\}$。

进一步，我们可以对上述表示方式进行改进以节约存储空间。设$G' \triangleq G \cup \{e\}$同样满足$G' \leq_{(l,h)} Y$，$Occ_G(Y)=\{X_1,X_2,\ldots,X_m\}$，则我们有如下的简单性质：

性质 4 $Occ_{G'}(Y) = \{X_1' \cup \{\vec{e}\}, X_2' \cup \{\vec{e}\}, ..., X_n' \cup \{\vec{e}\}\}$，$X_i' \in Occ_G(Y)$，$1 \leq i \leq n$。

这里需要注意两点，一是$m \neq n$，二是不同的$X_i' \cup \{\vec{e}\}$的$\{\vec{e}\}$可能对应于不同的独立路径，只是为了简便起见我们才统一使用$\{\vec{e}\}$这一表示方式。性质 4 的证明是显然的。我们称X_i'为$X_i' \cup \{\vec{e}\}$的父出现，（Parent Occurrence），称G为G'的父模式（Parent Pattern），称G'为G的子模式（Child Pattern）。

现在我们定义拓扑结构G在数据集D中的出现记录表（Occurrence List）为：$Occ_{G,D} = \cup_{i=1}^{|D|} Occ_G(Y_i)$，$Y_i \in D$。由性质 4 我们可以知道，$Occ_{G'}(Y)$可以基于$Occ_G(Y)$来进行表示。这样，当我们为拓扑结构$G'$建立出现记录表$Occ_{G',D}$时，我们就可以利用$Occ_{G,D}$中的信息来增量地进行，从而大大减少了时间和空间的开销。

需要指出的是，一个拓扑结构可能会有多个父模式（因为同一个子模式可以由不同的父模式通过增加一条不同的边来得到），但是我们只需要它的一个父模式就足以建立起该子模式的出现记录表了（因为从根本上说模式的出现只与模式本身相关联）。至于选择该子模式的哪个父模式，这其实是如何进行模式增长的问题，将在 4 .1.3 小节进行说明。

经过上面的论述，现在我们可以正式地给出一个拓扑结构的表示方式了：
拓扑结构G可以表示为如下的基本结构 *G: (parent, edge, occurrence_list)*，其中 *parent* 是指向其某个父模式的指针（由模式增长的方式决定，只需要一个父模式）；*edge* 是G所对应的边（即$G \triangleq \{*parent\} \cup \{edge\}$，见本小节前面的论述）；*occurrence_list* 是G在数据集D中的出现记录表，即$Occ_{G,D}$，可以表示为由G的 *occurrence* 所组成的集合。

而每个 *occurrence* 又可以表示为 *occurrence: (graph_id, parent_occ, path)*
graph_id 该 *occurrence* 在数据集D中所出现的图的编号；
parent_occ 指向其父出现的指针（每个 *occurrence* 的父出现是唯一的）；
path 是该 *occurrence* 对应于G中 *edge* 的独立路径。

在这一小节的最后，我们就本小节开头提出的三个问题对我们给出的拓扑结构表示方式进行一个评估。先考虑空间效率，由于采用了增量式地存储结构（即每个模式的信息基于其父模式的信息来进行表示的方式），所以该表示方式是比较节约存储空间的。再看统计模式在出现频次，由于每个模式的出现记录表中记

录了该模式在数据集中的所有出现，所以我们只要统计 *occurrence_list* 中不同的 *graph_id* 的数目就可以得到模式在数据集中的出现频次了，这是一项非常简单的工作。最后，我们来看一下模式增长时的开销。当我们对当前模式通过增加一条边的方式扩展出一个新的模式时，主要的开销集中在为该新模式建立出现记录表上，这正是下一小节需要讨论的问题。相信经过下一小节的论述，我们会对这个问题有一个清楚的认识。

4.1.2 出现记录表的建立

上面已经提到，当我们由当前模式 G 扩展一条边 e 得到一个新的模式 $G'=G\cup\{e\}$时，我们需要为 G'建立它的出现记录表，这一小节中我们详细地来阐述一下出现记录表的建立过程。

建立出现记录表的最为直接的做法是：对每个 $X_i\in Occ_{G,D}$，寻找所有对应于 e 的独立路径 $p_j=\vec{e}$，即 $p_j\perp P(X_i)_{D(X_i.graph\_id)}$，$P(X_i)$为组成 X_i 的独立路径集合，$D(X_i, graph_id)$表示 D 中包含 X_i 的数据图。将每个找到的 $X_i\cup\{p_j\}$，添加到 $Occc_{G',D}$ 中。

直接做法虽然简单，但是会带来大量重复的工作。例如，假设我们现在要为两个模式 $G\cup\{e_1\}$和 $G\cup\{e_2\}$建立各自的出现记录表，而 e_1 与 e_2 又与 G 中的同一个顶点 v 相关联，那么上述做法意味着我们对每个 $X_i\in Occ_{G,D}$，都要通过从相应的 $Occ_{G,X_i,D(X_i.graph\_id)}(v)$ 出发寻找所有长度在 l 和 h 之间的路径来确定我们实际要找的与 $P(X_i)$独立的路径集合。显然，寻找所有长度在 l 和 h 之间的路径的工作在这里重复了两次，并且我们后面将会看到这一工作的代价是相当大的，因此采用上述方法的效率是非常低的。

从对上面介绍的方法的分析我们不难想到第一个可行的优化方法是我们事先为 D 中的每张数据图的每个顶点建立一张从该顶点出发的长度在 l 和 h 之间的路径表（称为该顶点的*(l,h)*路径表），这样当我们每次需要这些路径时我们就可以直接通过查表的方式来得到而不需要重新进行寻找。另外，我们只需要记录那些存在频繁边与之相对应的路径就可以了，因为根据 Apriori 性质，构成频繁拓扑结构的每一条边必定也是频繁的。

但是，即使是仅仅遍历顶点的*(l,h)*路径表，重复工作也有很多的。接着上面的例子，假设我们已经找到了模式 $G\cup\{e_1\}$和 $G\cup\{e_2\}$，现在我们在 $G\cup\{e_1\}$上通过扩展 e_2 得到一个新的模式 $G''=G\cup\{e_1\}\cup\{e_2\}$，该如何建立 G''的出现记录表呢？根据前面的做法，我们又需要重新遍历每个 $Occ_{G\cup\{e_1\},X_i,D(X_i.graph\_id)}(v)$ 的*(l,h)*路径表（$X_i\in Occ_{G\cup\{e_1\},D}$）来寻找与 e_2 所对应的独立路径。但事实上在建立 $G\cup\{e_2\}$

的出现记录表时，我们已经进行过类似的工作，并且我们可以断言，如果 $p_j = \overrightarrow{e_2} \in P(X_i)$，$X_i \in Occ_{G',D}$，则一定成立 $p_j \in P(X_k)$，$X_k \in Occ_{G\cup\{e_2\},D}$。换言之，我们只需要从 $Occ_{G\cup\{e_2\},D}$ 中与 e_2 对应的那些独立路径中来寻找与 $G\cup\{e_1\}$ 相独立的路径从而进一步构造 $Occ_{G',D}$ 就可以了，而不必从所有 D 中 e_2 所对应的独立路径重新开始寻找。这一性质启示我们，可以采用逐步过滤的方式来缩小每次需要搜索的路径空间，为了更方便地说明问题，我们先引入一些记号。

定义 23 内边（Inner Edge）与外边（Outer Edge）

设我们由当前模式（拓扑结构）G 通过扩展一条边 $e=(u,v)$ 得到一个新的模式 $G' \triangleq G\cup\{e\}$，则 e 可以分为两种情况：

1）$u\in V(G)$ 且 $v\in V(G)$，即 e 连接的是两个已经存在于 G 中的顶点，我们称 e 为 G 的一条内边；

2）$u\in V(G)$ 但 $v\notin V(G)$，即 e 连接的是一个 G 中已经存在的顶点和一个不在 G 中的新的顶点，换言之，e 引入了一个新的顶点，我们称 e 为 G 的外边。

我们用 $[G]_{inner}$ 来表示由 G 的所有内边所组成的集合，用 $[G]_{outer}$ 来表示由 G 的所有外边所组成的集合，用 $[G]_{io}$ 来表示 $[G]_{inner}$ 和 $[G]_{outer}$ 的并，即 $[G]_{io}=[G]_{inner}\cup[G]_{outer}$。定义集合 $CP(G)_{inner}=\{G'|G'=G\cup\{e\},e\in[G]_{inner}\}$，$CP(G)_{outer}=\{G'|G'=G\cup\{e\},e\in[G]_{outer}\}$，称 $CP(G)_{inner}$ 为由 $[G]_{inner}$ 导出的候选模式（内模式）集，$CP(G)_{outer}$ 为由 $[G]_{outer}$ 导出的候选模式（外模式）集，类似地，我们记 $CP(G)_{io}=CP(G)_{inner}\cup CP(G)_{outer}$。

我们的想法是为每个当前已经得到的频繁拓扑结构 G 维护它的 $CP(G)_{inner}$ 和 $CP(G)_{outer}$。$[G]_{inner}$ 和 $[G]_{outer}$ 的意义在于它们记录了 G 的所有可能扩展方式，因此 $CP(G)_{inner}$ 和 $CP(G)_{outer}$ 记录了所有由 G 可以扩展出的模式，下面我们只需要考虑两个问题就可以了：

1）如何构造 $CP(G)_{io}$？

2）设 $G'=G\cup\{e\}$，$e\in[G]_{io}$（即 $G'\in CP(G)_{io}$），如何构造 G' 的出现记录表？

先考虑第一个问题，构造 $CP(G)_{io}$。由于我们的算法采取两阶段的模式挖掘框架，因此事实上（在 4.1.3 会提到），我们在算法的第一阶段（即树模式的挖掘阶段）只需要进行外边的扩展，而在算法的第二阶段（即（有环）图模式的挖掘阶段）只需要进行内边的扩展。也就是说，对于树模式挖掘阶段的每个模式 G，我们只需要构造 $CP(G)_{outer}$；而对图模式挖掘阶段的每个模式 G，我们只需要构造 $CP(G)_{inner}$。下面我们分别来讨论 $CP(G)_{outer}$ 和 $CP(G)_{inner}$ 的构造。

算法 1 $CP(G)_{outer}$ 的构造

```
BuildOuterPatternSet (G){
    foreach (H∈CP(G.parent)outer){ //对父模式的每个外模式进行遍历
        if ( IndependentOuterEdge (H.edge, G.edge) ){
            /*如果该外模式对应的外边仍为 G 的外边，则根据该外边构造一
            个 G 的外模式 G'，G'.parent=G，G'.edge=H.edge*/
            G'=G∪{H.edge};
            JoinOccurrenceList (H, G, G'); //建立 G'出现记录表
            if ( IsFrequent (G') )
                //如果 G'是频繁的，将其加入到 G 的外模式集合中
                CP(G)outer.add(G');
        }
    }
}

IndependentOuterEdge (H.edge, G.edge){
    if (H.edge.from==G.edge.from    && H.edge.to != G.edge.to )
        return    true;
    if (H.edge.from!= G.edge.from    && H.edge.to != G.edge.to )
        return    true;
    return    false;
}
```

算法 2　$CP(G)_{inner}$ $CP(G)_{inner}$ 的构造

```
BuildInnerPatternSet (G){
    foreach (H∈CP(G.parent)inner){ //对父模式的每个内模式进行遍历
        if ( IndependentInnerEdge (H.edge, G.edge) ){
            /*如果该内模式对应的内边仍为 G 的内边，则根据该内边构造一
            个 G 的内模式 G'，G'.parent=G，G'.edge=H.edge*/
            G'=G∪{H.edge};
            JoinOccurrenceList (H, G, G'); //建立 G'出现记录表
            if ( IsFrequent (G') )
                //如果 G'是频繁的，将其加入到 G 的内模式集合中
                CP(G)inner.add(G');
        }
```

```
    }
}

IndependentInnerEdge (H.edge, G.edge){
    if (!(H.edge.from==G.edge.from    && H.edge.to==G.edge.to) )
        return    true;
    return    false;
}
```

从算法 1 和算法 2 我们可以看到 $CP(G)_{outer}$ 和 $CP(G)_{inner}$ 是类似的，两者只有微小的差别，这是由分阶段挖掘的特点所决定的。事实上，在树模式增长阶段，一些外边随着模式大小的增加会转变为后续模式的内边，但是为了过程上的简单和统一，我们在树模式增长阶段将忽略所有产生的内边。在 4.1.3 中我们将会看到，在从树模式增长阶段转入图模式增长阶段之前，我们需要初始化各树模式的内模式集。另外这里需要指出的是，这里构造的 $CP(G)_{outer}$ 只包含了 G 的一部分外模式（继承自 G 的父模式），另一部分的外模式的获得将在 4.1.3 进行介绍。

另外上面两个算法中我们还没有给出 ***JoinOccurrenceList*** 函数的伪码，而这正是我们前面已经指出的需要考虑的第二个问题。我们马上来看一下它的具体过程。

算法 3　构造 G' 的出现记录表

```
JoinOccurrenceList (H, G, G') { //H, G, G'的含义见算法1和2中的上下文环境
    foreach ( h∈H.occurrence_list ){
        foreach ( g∈G.occurrence_list){
            if ( h.parent_occ==g.parent_occ){
                /*如果果某个 h 和 g 具有同一个父出现，则可以对它们尝试
                进行连接（Join），连接的条件是 h.path 独立于构成 g 所对应
                的子分割图的独立路径集。函数 GetOccGraph (g)用于得到 g
                所对应的子分割图，具体定义见后。*/
                if ( h.path⊥P( GetOccGraph (g) ){
                    new occurrence g';
                    g'.graph_id=g.graph_id;//当然，g.graph_id==h.graph_id
                    g'.parent_occ=g;
                    g'.path=h.path;
                    G'.occurrence_list.add( g');
                }
```

```
            }
        }
    }
}

GetOccGraph ( g ){ //g 的含义见 JoinOccurrenceList 的上下文环境
    X := ∅;
    /*若 g.parent_occ==NULL，则 g 所对应的模式是初始模式，即只具有一
    条频繁边的模式。*/
    while ( g.parent_occ!=NULL){
        X:=X∪{g.path};
        g=g.parent_occ;
    }
    return   X;
}
```

可以看到 ***JoinOccurrenceList*** 函数的关键部分就在于保证当我们在原有模式的基础上扩展一条边时，构成新的模式的出现记录表中的每个出现的路径集合仍然是独立的。另外，***JoinOccurrenceList*** 函数的时间代价取决于输入模式 H 和 G 的出现记录表的大小。

至此，我们已经详细地讨论了出现记录表的建立过程并给出了相应算法的伪码表示。应该说，出现记录表的建立和维护是整个算法过程的两个核心步骤之一。在下一小节中，我们将讨论整个挖掘过程的另一个核心步骤，即如何进行模式增长。

4.1.3 模式增长的具体过程

由于整个挖掘过程分为树模式和图模式两个阶段，因此我们也将模式增长的具体过程分为两个阶段来进行介绍。

4.1.3.1 树模式挖掘阶段的模式增长

设 T 是树模式挖掘阶段的一个频繁模式（拓扑结构），我们要决定如何在 T 上扩展一条边 e 来得到一个新的模式 $T'=T\cup\{e\}$。这里，我们取 e 为 T 的一条外边，即 $e\in[T]_{outer}$。在整个树模式挖掘阶段，我们都只通过增加外边的方式来进

行模式增长，因此每一步的增长都会有一个新的顶点被引入。

另一方面，*T* 的外边有两种来源，一种是直接继承自父模式的外模式集合，还有一种则是新扩展的外边。第一种情况我们在 4.1.2 已经讨论过，见上面的算法 1。下面我们来考虑第二种情况：设 *T.edge=(u,v)*，新扩展的外边来自于从 *v* 出发的*(l,h)*路径，显然我们无法通过继承父模式的外模式集合来得到这样的外边（外模式），因为 *T.edge* 在 *T* 的父模式中不存在，因此第二种情况下的外边与 *T* 的父模式是没有公共顶点的。

那么我们现在唯一需要解决的问题就是如何从当前模式*T*出发来寻找第二种情况下的外边 *e*。

4.1.2 中已经提到，为了减少不必要的重复工作，我们事先为 *D* 中的每张数据图的每个顶点建立该顶点的*(l,h)*路径表。设数据图为 D_i，$u \in V(D_i)$，则我们记 *u* 的*(l,h)*路径表为 $PL_{(l,h),D_i}(u)$。从当前模式 *T* 出发寻找第二类外边（外模式）的算法如下：

算法 4 从 *T* 出发寻找第二类外边（外模式）

```
ExtendOuterEdges ( T ){
    E=∅;
    foreach ( t∈T.occurrence_list ){ // t 为 T 的一个 occurrence
        X=D( t.graph_id ); //X 为 D 中包含 t 的数据图
        foreach ( p∈PL(l,h),X( t.path.to ) ){
            //p 为 X 中从 t.path.to 出发的一条(l,h)路径
            if ( p⊥P(t) ){//如果 p 与构成 t 的独立路径集是独立的
                e=( p.from, p.to );
                //构造一个以 e 为外边的外模式 T',T'.parent=T,T'.edge=e
                T'=T∪{e};
                T"= IndexOf(T', E);//检查 T'是否已经出现过
                if ( T"== NULL)
                    E=E∪{T'};
                else
                    T'=T";//T'已经出现过，则将 T'指向已经出现过的模式
                //构造 T'的一个 occurrence
                new occurrence t';
                t'.graph_id=t.graph_id;
                t'.parent_occ=t;
                t'.path=p;
```

```
                T'.occurrence_list.add( t');
            }
        }
    }
    foreach ( T'∈E ){
        if ( IsFreqent ( T') )
            //如果 T'是频繁的，则将 T'加入到 T 的外模式集合中
            CP(T)outer.add(T');
    }
}

IndexOf(T', E){
    T''=NULL;
    foreach ( Te∈E ){
        if ( Te.edeg==T'.edge){
            T''= Te;
            break;
        }
    }
    return T'';
}
```

4.1.3.2 图模式挖掘阶段的模式增长

在图模式挖掘阶段，我们将以树模式挖掘阶段得到的模式作为生成树来进行（有环）图模式的扩展，因此，每一步的增长是通过在已有模式的顶点间加一条边来进行的，不会引入新的顶点。换言之，设 G 是图模式挖掘阶段的一个模式，我们将在 G 上增加一条内边 e（即 $e\in[G]_{inner}$），来得到一个新的模式 $G'=G\cup\{e\}$。于是，模式增长的问题转变为如何寻找 G 的内边的问题。而 G 的内边来源只有一个，即继承自父模式的内模式集合。相关的工作已经在 4.1.2 种介绍过了（见算法 2）。

前面已经提到过，由于在树模式挖掘阶段我们只考虑外边的增长，因此在进入图模式挖掘阶段之前，我们需要为树模式挖掘阶段得到的每个模式初始化它的

内模式集合，见下面的算法：

算法 5 初始化树模式的内模式集合

```
InitialInnerEdgeSet ( T ){
    foreach ( e=(u,v)∈V(T)×V(T) && e∉E(T) ){
        //考虑还未加入到 T 中的每一条可能内边 e
        //以 e 为内边构造 T 的一个内模式 T',T'.parent=T,T'.edge=e
        T'=T∪{e};
        foreach ( t∈T.occurrence_list ){//构造 T'的 occurrence_list
            X=D(t.graph_id);
            foreach ( p∈PL(l,h),X( Occ T,t,X(u) ){
                //遍历 t 在 X 中对应于 u 的顶点的(l,h)路径表
                if ( p.to==Occ T,t,X(v) && p⊥P( t ) ){
                    /*如果 p 确实对应于(u,v)且与构成 t 的独立路径集相独
                    立，则基于 p 构造 T'的一个出现 t' */
                    new occurrence t';
                    t'.graph_id=t.graph_id;
                    t'.parent=t;
                    t'.path=p;
                    T'.occurrence_list.add( t');
                }
            }
        }
        if ( IsFrequent( T' )
            //如果 T'是频繁的，则将 T'加入到 T 的内模式集合中
            CP(T)inner.add( T')
    }
}
```

至此，我们已经讨论完了模式增长的具体过程。但是，我们很容易想象，上述模式增长的过程中必定会产生许多重复的模式，因为同一个模式可以来自许多不同的增长路线。例如，在图 4 中，G_3 既可以由 G_1 扩展边(A,C)得到，又可以由 G_2 扩展边(A,B)得到。

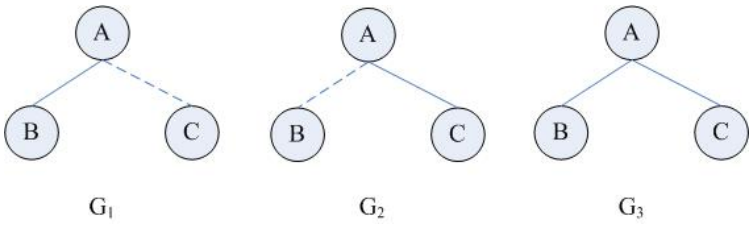


图 4 模式增长过程中重复模式的产生

显然对一个与已经发现的某个模式重复的模式继续进行扩展是没有任何意义的，因为由该模式扩展得到的模式必然也可以通过对之前发现的那个模式进行扩展得到。所以我们需要尽可能早地发现重复的模式，从而有效地缩小模式搜索的空间，提高挖掘的时间效率。至于如何来发现重复模式，这正是下一小节要讨论的问题。

4.1.4 重复模式的发现

重复模式的发现从本质上说是一个图同构判断的问题。图同构的定义我们已经在第二章进行了介绍，下面我们来看看具体到算法上应该如何判断两个图的同构。

设 G_1 和 G_2 是两张（标记）图（这里设$|V(G_1)|=|V(G_2)|=n$，否则 G_1 和 G_2 肯定不同构），直观上说，最简单的判断 $G_1 \cong G_2$ 是否成立的方法是，枚举 $V(G_1)$和 $V(G_2)$ 之间所有可能的映射，然后判断每一个映射是否符合同构的定义，如果能够找到这样的一个映射，则 $G_1 \cong G_2$ 成立，否则 $G_1 \cong G_2$ 不成立。上述方法本质上相当于固定 $V(G_1)$中的顶点于某个顺序，枚举 $V(G_2)$中顶点所有可能的排列，而这样的排列理论上有 $n!$个。但是由于我们只要找到一个符合同构定义的排列就够了，因此在实际操作时存在着许多的剪枝策略。所以并不能断言图同构问题在最坏情况下一定是指数级的时间复杂度。

事实上，对图同构问题的研究已经有了几十年的历史。但是，直到目前，人们仍然无法从计算复杂度的角度给图同构问题进行一个界定。虽然很容易知道图同构问题是 NP 问题，但是尚不清楚它是 P 问题还是 NPC 问题。不少科学家怀疑图同构问题的复杂度介于 P 和 NPC 之间，如果这一怀疑成立，那么就可以对计算理论中最著名的难题（即 P=NP 是否成立?）做出解答。因此图同构问题一直吸引着许多理论计算机科学家和数学家进行研究。关于图同构问题理论研究的更多情况请参看[12]。

当然，使得图同构问题如此引人注目的原因不仅仅在于它在计算理论研究中的特殊地位，从实践角度来看，图同构判断是一个相当基本的问题，许多实际的

应用领域都需要涉及到它，因此，寻找高效率的图同构判断算法具有巨大的现实意义。

在过去的几十年里，出现了不少在实践中被证明是行之有效的算法，如 J.R.Ullmann 的回溯算法[13]，B.D.Mckay 的 Nauty 算法[14]等，都是其中的优秀代表。近年来，许多人开始将注意力转向针对某种特殊类型的图（如自由树，平面图（Planar Graph），外平面图（Outer-planar Graph）等）的同构算法的研究上来。事实证明，虽然对于一般的图而言目前还无法找到多项式时间复杂度的算法，但是对于不少上述特殊类型的图而言，多项式时间复杂度的算法是存在的，这也是为什么我们要采取分阶段模式挖掘的一个主要动机。

在介绍具体的同构判断算法之前，我们需要先定义一个重要的概念，图的规范标记（Canonical Label）。

定义 24 图的规范标记（Canonical Label）

设 $\boldsymbol{G}$ 是由所有图组成的集合，$\boldsymbol{C}$ 是一个标记（即字符串）集合，若存在映射 $f: \boldsymbol{G}\rightarrow\boldsymbol{C}$，满足：$\forall G，G'\in\boldsymbol{G}，f(G)=f(G')\Leftrightarrow G\cong G'$， 则称 f 为图的一个规范标记函数（Canonical Labeling Function），称 G 在 f 下的象 $c=f(G)$为 G 的一个规范标记，记为 $c(G)$。

规范标记的优点在于将复杂的图同构问题转化为了简单的字符串比对问题。从上面的定义我们不难看出，规范标记的概念是相当宽泛的。事实上，现在的许多图同构判断算法都采用了规范标记的思想，所不同的只是规范标记的具体形式，比如上面提到的 Nauty 算法就是一个基于规范标记的算法。

就我们的算法框架来说，在模式增长阶段，我们考虑的是一般图的同构问题，因此我们可以直接利用 Nauty 算法来进行同构判断。（Nauty 算法的作者已经将其包装为一个可以免费使用的软件包放在互联网上供下载[15]）Nauty 算法的理论基础涉及到许多有限置换群（Finite Permutation Group）的知识，并且其算法过程比较复杂，我们在这里略去对它的介绍，有兴趣的读者可以参看[16]。

而在树模式增长阶段，我们考虑的是自由树的同构问题。自由树的同构问题要比一般图的同构问题简单许多，具体地说，自由树的规范标记可以在多项式的时间复杂度里计算得到，比如[17]。因此，我们只需要采用某种自由树的规范标记就可以有效地解决同构判断的问题。

有兴趣的读者可以参看相关文献来了解各种不同的规范标记形式，我们这里不再介绍。

下面给出进行同构模式判断函数的伪码，*ComputeTreeCanonicalLabel* 和

ComputeGraphCanonicalLabel 是计算模式规范标记的函数，由具体的规范标记的形式决定，因此这一小节里只给出接口形式：

算法 6 同构模式的发现

```
global hash_table canon_trees;
ValidTreeExtension(T){
    ComputeTreeCanonicalLabel(T);
    if ( c(T)∈canon_trees )
        return false;
    else{
        canon_trees.add( c(T) );
        return true;
    }
}

global hash_table canon_graphs;
ValidGraphExtension(G){
    ComputeGraphCanonicalLabel(G);
    if ( c(G)∈canon_graphs )
        return false;
    else{
        canon_graphss.add( c(G) );
        return true;
    }
}
```

上面讨论的是在一般的标记图集上进行图的拓扑结构挖掘时我们可以采用的同构判断策略。在 4.3 节中，我们将针对 PPI 网络的特点给出我们自己的同构判断算法。具体而言，我们将定义自己的规范标记，从而实现算法 6 中给出的两个接口函数 *ComputeTreeCanonicalLabel* 和 *ComputeGraphCanonicalLabel*。

4.1.5 其他涉及的重要函数

这一小节中我们给出前面已经提到过但还没有给出具体实现的三个较为重要的函数：一是频繁模式的判断，二是构造给定图中所有顶点的*(l,h)*路径表，三

是寻找所有规模为 1 的频繁模式（即频繁边）。

算法 7 频繁模式的判断

```
IsFrequent (G){
    hash_table diff_gids;
    for ( g∈G.occurrence_list ){ //统计 G 在数据图集中的出现频次
        if ( g.graph_id∉diff_gids)
            diff_gids.add(g.graph_id);
    }
    If ( diff_gids.size()≥θ )
        //如果 G 的出现频次不小于给定的支持度阈值θ，则 G 是频繁的
        return true;
    return false;
}
```

算法 8 构造给定图中所有顶点的*(l,h)*路径表

```
FindAllPathsInTheGraph( l, h, Di ){
    foreach ( u∈V(Di) ){
        new path p;
        hash_table visited_nodes; //visited_nodes 中记录了已经访问过的顶点
        p.addNode(u);
        //从 u 出发寻找寻找它的(l,h)路径
        FindAllPathsStartWithNode( G, u, l, h, visited_nodes, p, u);
        p.removeNode(u);
    }
}

FindAllPathsStartWithNode ( G, u, l, h, visited, p, start_node){
    if ( |p|≥h)   //p 的长度已经达到上限，停止递归
        return;
    visited.add(u);   // u 已经被访问过
    foreach ( v∈NG (u) ){ //遍历 u 的所有邻居节点
        if ( v∉visited ){//若某个邻居节点 v 尚未被访问，访问之
            p.addNode(v);
```

```
            /*若加入 v 后 p 的长度达到下限，则将 p 加入
            到 start_node 的(l,h)路径表中*/
            if ( |p|≥l )
                PL(l,h),G( start_node ).addPath( copyPath(p) );
            //从 v 出发递归地寻找路径
            FindAllPathsStartWithNode( G, v, l, h, visited, p, start_node);
            p.removeNode(v);
        }
    }
    visited.remove(u); //从已访问定点的集合中去除 u
}

copyPath(p){ //拷贝一条与给定路径 p 具有相同顶点序列的路径
    new path p';
    foreach ( v∈V(p) )
        p'.addNode(v);
    return p';
}
```

算法 9　寻找所有规模为 1 的频繁模式

```
FindFrequentEdgeTS( D, l, h, θ ){
    E=∅;
    foreach(Di∈D){
        foreach (u∈V(Di) ){
            foreach(p∈PL(l,h),Di(u) ){ //对图集 D 中的所有(l,h)路径进行遍历
                e=(p.from, p.to);
                T={e};
                T"=IndexOf(T,E); //IndexOf 的实现见算法 4
                if (T"==NULL )
                    E=E∪{T};
                else
                    T=T";
                //构造 T 的一个 occurrence
                new occurrence t;
```

```
                t.graph_id=Di.graph_id;
                t.parent_occ=NULL;
                t.path=p;
                T.occurrence_list.add(t);
            }
        }
    }
    foreach ( T∈E){
        if ( ! IsFrequent(T) ) //若 T 不是频繁的，则从 E 中去除 T
            E=E-{T};
    }
    return E;
}
```

算法 10　初始化规模为 1 的频繁模式的外模式集合

```
InitialOuterEdgeSet(T){
    foreach(e∈E){
        if ( e.edge.from==T.edge.from && e.edge.to!=T.edge.to){
            T'=T∪{e};
            JoinOccurrenceList (e, T, T');
            if (IsFrequent(T'))
                CP(T)outer.add(T');
        }
    }
}
```

说明：因为在模式扩展的过程中，我们一旦发现当前模式与已知的某个模式同构则停止对当前模式的扩展，因此必须在每一步都保证从当前模式出发的所有可能扩展都已经被枚举到，才不会造成模式的缺失。这正是我们需要对规模为 1 的频繁模式的外模式集合进行初始化的原因所在。

4.2 算法的详细框架

经过 4.1 节的讨论，我们已经清楚了算法关键步骤的具体过程。下面我们将给出整个算法的详细框架。

算法框架 在标记图集上挖掘频繁拓扑结构

```
global list fre_patterns;
MiningFrequentTS( D,l,h,θ ){
    //找出 D 中所有的(l,h)路径
    foreach( Di∈D )
        FindAllPathsInTheGraph( l, h, Di);
    //找出所有规模为 1 的频繁模式，并将其加入到 fre_patterns 中
    E=FindFrequentEdgeTS(D, l, h,θ );
    foreach (T∈E)
        fre_patterns.addPattern(T);
    //初始化规模为 1 的频繁模式的外模式集合
    foreach (T∈E)
        InitialOuterEdgeSet(T);
    //从规模为 1 的频繁模式出发，进行两阶段的模式挖掘
    foreach (T∈E)
        MiningTreeTS(T);
}

MiningTreeTS(T){
    ExtendOuterEdges ( T ); //对当前的树模式 T 进行第二类外模式扩展
    foreach(T'∈CP(T)outer){ //检测 T 的所有外模式
        if (ValidTreeExtension(T') ){
            //发现了一个新的频繁树模式 T'
            fre_patterns.addPattern(T');
            //对 T'进行第一类外模式扩展
            BuildOuterPatternSet(T');
            //对 T'进行递归地处理
            MiningTreeTS(T');
        }
    }
    //初始化 T 的内模式集合
    InitialInnerEdgeSet ( T );
    //以 T 为生成树构造（有环）图模式
```

```
    MiningGraphTS(T);
}

MiningGraphTS(G){
    foreach (G'∈CP(G)inner){//检测 G 的所有内模式
        if (ValidGraphExtension(G') ){
            //发现了一个新的频繁图模式 G'
            fre_patterns.addPattern(G');
            //对 G'进行内模式扩展
            BuildInnerPatternSet (G');
            //对 G'进行递归地处理
            MiningGraphTS(G');
        }
    }
}
```

最后需要再次强调的一点是，上述的算法框架对于一般的标记图集上的拓扑结构挖掘都是适用的，在下一节中我们将根据 PPI 网络的特点对算法进行改进。

4.3 根据PPI网络的特点对算法进行改进

PPI 网络自身的特点为我们改进上述算法提供了可能，这一节里我们来讨论这个问题。

我们在定义问题时已经作了特别说明，即一般来说，在 PPI 网络图集 D 上进行平频繁拓扑结构的挖掘时，我们将支持度阈值 θ 设为$|D|$，换句话说，模式图需要在每一张数据图中至少有一次出现（因为我们关心的是所有物种间共有的保守结构）。

因此，虽然我们前面在定义 PPI 网络挖掘问题时将频繁拓扑结构模式视为无标记图，但是现在在支持度为$|D|$的前提下，我们可以重新把模式视为标记图。具体来说，设 D_{min} 是 D 中的顶点数最少的一张数据图，G 是 D 的任意一个频繁拓扑结构，则 $\forall u \in V(G)$, 定义 u 的标记为 $l_{V(G)}(u)=l_{V(Di)}(Occ_{G,g,Dmin}(u))$, g 为 G 在 D_{min} 中的某个出现。显然这样做会带来一个问题：当 G 在 D_{min} 中出现的次数超过一次时我们以 G 的哪个出现 g 作为对 G 进行标记的依据呢？为了越过这个障碍，我们规定，每个 G 只能在 D_{min} 中出现一次。换句话说，如果无标记拓扑结

构 G 在 D_{min} 中有两个出现 g',g'',则我们将 g'和 g''视为两个不同的有标记拓扑结构 G'和 G''在 D_{min} 中的出现，满足在忽略 G'和 G''的标记信息时，成立 $G' \cong G'' \cong G$。于是，我们现在可以把模式挖掘的过程想象为先以 D_{min} 为标记依据挖掘有标记的模式，然后忽略模式的标记信息进行同构判断将（无标记情况下）同构模式的出现记录表合并，从而得到最终的无标记的模式。显然，这样做并不会对最后的结果造成影响，但是却可以给我们进行模式增长以及重复模式的发现带来很大的好处。

经过上述的转化，我们事实上可以基于 D_{min} 来进行模式发现和扩展的工作，换句话说，由于每个模式都对应于 D_{min} 中的唯一一个出现（可能出现对应一个以上出现的情况，因为模式中的一条边可能会与一条以上的独立路径相对应，这种情况下我们取最先发现的那个出现与模式进行对应），因此模式的出现记录表中可以只存储模式在除 D_{min} 之外的图中的出现，而把在 D_{min} 中的出现作为模式本身的一个属性单独存放。这样一来，我们在模式扩展时就不再需要通过遍历出现记录表和相应顶点的(l,h)路径表的方式来寻找可能的扩展了。事实上，我们在找到了所有的频繁边之后，只需要保留 D_{min} 中所有顶点的(l,h)路径表，而且只需要保留与那些频繁边对应的路径及其在除 D_{min} 之外的图中的相应出现就可以了。在模式扩展时，我们通过查找 D_{min} 中相应顶点的(l,h)路径表就可以得到候选的扩展模式了。

另一方面，当经过前面的转化后，我们可以给通过给模式定义新的规范标记的方式来加快重复模式的发现。由于 D_{min} 的顶点是唯一标记的，因此基于 D_{min} 的拓扑结构模式必然也是唯一标记的。下面我们根据这一特点分别给出树模式挖掘阶段和图模式挖掘阶段的两种不同的规范标记。

4.3.1 树模式挖掘阶段的规范标记

设 T 是树模式挖掘阶段的一个频繁模式，则 T 是一棵自由树的形式。下面我们要给出关于自由树的一个重要性质，在这之前，先介绍一个概念。

定义 25　自由树的中心（Center）

设 T 是一棵自由树，$\forall u \in V(T)$, 定义 $max\_dist(u)=max\{dist(u,v) | v \in V(T), v \neq u\}$。其中 $dist(u,v)$表示顶点 u 和 v 之间的距离（连接 u 和 v 的路径的长度，由于 T 中没有回路，故 p 是唯一的），即 $dist(u,v)=|p|$，$p.from=u$ 且 $p.to=v$。

若 $c \in V(T)$, 且满足 $max\_dist(c)=min\{max\_dist(u) \mid u \in V(T)\}$，则称 c 为 T 的一个中心。

性质 5 设 T 是一棵自由树，则 T 至少有一个中心，但最多有两个中心。

证明：

由自由树的中心的定义知 T 至少有一个中心是显然的。另一方面我们很容易就能找到有两个中心的自由树 T（如图 5）。现在我们证明 T 最多有两个中心。

假设 T 有两个中心，我们设 c_i,c_j,c_k 为 T 的任意三个中心。由于 T 是自由树，因此 c_i 和 c_j 之间必然有一条唯一的简单路径 $p(c_i,c_j)$相连接。下面我们证明必有 $|p(c_i,c_j)|=0$，即 $p(c_i,c_j)$没有内点，从而 c_i 和 c_j 是相邻的。

反证之，如果$|p(c_i,c_j)| \neq 0$，则至少存在某个顶点 c，满足 $c \in IVS(p(c_i,c_j))$。设 $v_i,\ v_j \in V(T)$, 且 $dist(c_i,v_i)=max\_dist(c_i),\ dist(c_j,v_j)=max\_dist(c_j)$，因为 c_i 和 c_j 都是 T 的中心，因此 $dist(c_i,v_i)= dist(c_j,v_j)$。

设 $v \in V(T)$且满足 $dist(c,v)=max\_dist(c)$,则必有 $dist(c,v)< dist(c_i,v_i)$或 $dist(c,v)< dist(c_j,v_j)$。这是因为如果 $dist(c,v) \geq dist(c_i,v_i)$且 $dist(c,v) \geq dist(c_j,v_j)$,则必有 $v \neq v_i$, 且 $d(c_i,v)>d(c,v) \geq dist(c_i,v_i)$, 与 v_i 的选择矛盾；或者 $v \neq v_j$, 且 $d(c_j,v)>d(c,v) \geq dist(c_j,v_j)$，与 v_j 的选择矛盾。然而，无论是 $dist(c,v)< dist(c_i,v_i)$还是 $dist(c,v)< dist(c_j,v_j)$，都是与 c_i 和 c_j 是 T 的中心的假设相矛盾的。这说明存在 $c \in IVS(p(c_i,c_j))$的假定是不能成立的，故$|p(c_i,c_j)|=0$，c_i 和 c_j 必是相邻的

经过上面的论证我们知道 $c_k \notin IVS(p(c_i,c_j))$。因为 c_k 也是 T 的中心，所以 $max\_dist(c_k)=max\_dist(c_i)=max\_dist(c_j)$。又由于 T 是自由树，故必成立一下两种情况之一：

(1)T 中存在唯一的一条连接 c_i,c_k 的简单路径 $p(c_i,c_k)$,且 $c_j \in IVS(p(c_i,c_k))$；

(2)T 中存在唯一的一条连接 c_j,c_k 的简单路径 $p(c_j,c_k)$,且 $c_i \in IVS(p(c_j,c_k))$；

若(1)成立，则 $dist(c_k,v_j)>dist(c_j,v_j)=max\_dist(c_j)=max\_dist(c_k)$，矛盾；若(2)成立，则 $dist(c_k,v_i)>dist(c_i,v_i)=max\_dist(c_i)=max\_dist(c_k)$，矛盾。所以 c_k 不可能是 T 的中心。由 c_i,c_j,c_k 的任意性即知 T 中不可能有 3 个及以上的中心存在，即 T 最多有两个中心。

下面让我们来看一个实际的例子：

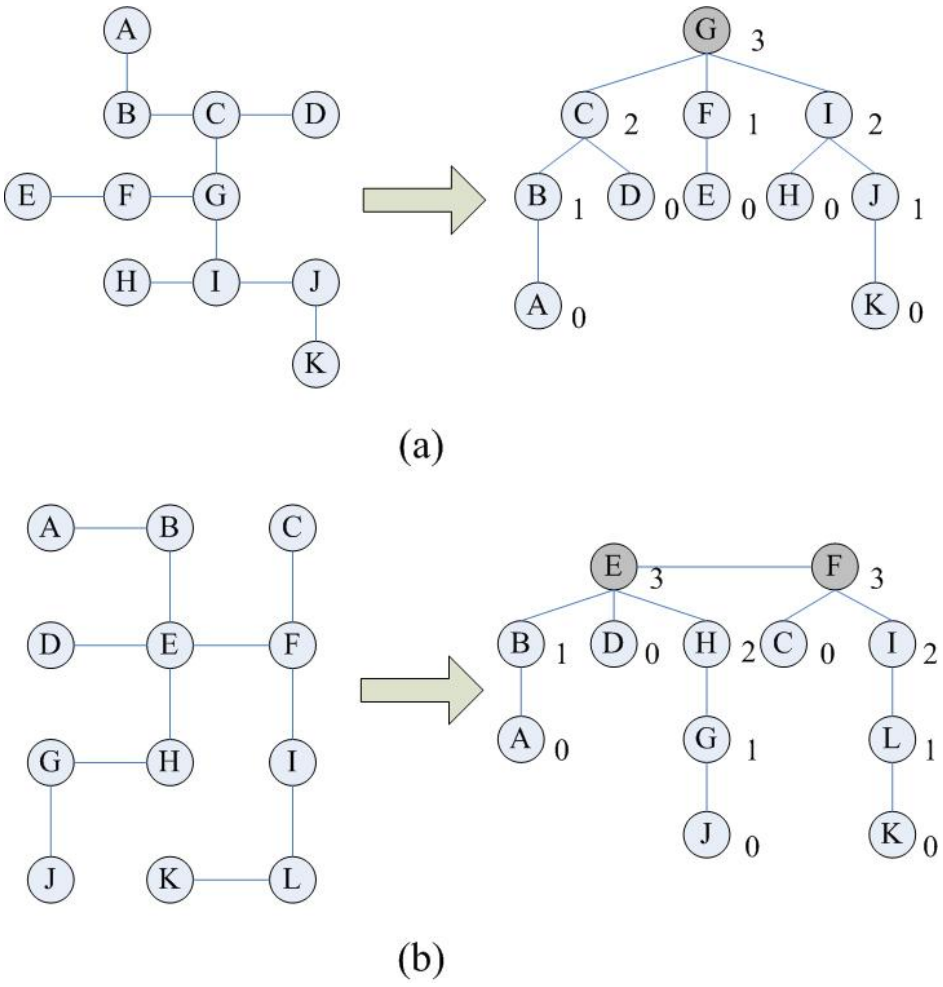


图 5 自由树的中心

如图 5 所示，(a)中的自由树有一个中心 G，而(b)中的自由树有两个中心 E 和 F。我们可以通过这样的方式来寻找自由树 T 的中心：$\forall u \in V(T)$，若 $d(u)=1$，则定义 $m(u)=0$，否则定义 $m(u)=max\{ m(v) \mid v \in N_T(u)\}+1$。若节点 c 满足 $m(c)=max\{ m(v) \mid v \in V(T)\}$，则 c 为 T 的一个中心。具体的算法如下：

算法 11 寻找自由树 T 的中心

```
FindCenters(T){
    C=∅;
    Mark(T); //计算 T 中顶点的 m 值
    max=-1; //得到 T 中顶点中最大的 m 值
    foreach(v∈V(T)){
        if (m(v)>max)
            max=m(v);
    }
    foreach(v∈V(T)){
```

```
        if(m(v)==max) // v 为 T 的一个中心
            C=C∪{v};
    }
    return C;
}

Mark(T){
    foreach(v∈V(T)) //初始化 T 中各顶点的 m 值为-1
        m(v)=-1;
    S=∅;
    m=0; //初始化 m 值为 0
    bool changed=false;
    while( true){
        foreach(v∈V(T)){ //寻找 T 中当前度数为 1 的顶点
            if (d(v)==1)
                S=S∪{v};
        }
        if (S!=∅)
            changed=true;
        foreach(u∈S){ //设定当前度数为 1 的顶点的 m 值
            m(u)=m;
            foreach(v∈N_T'(u)){ //相当于从 T 中去除顶点 u
                d(v)=d(v)-1;
                N_T'(v)=N_T'(v)-{u};
            }
            d(u)=0;
        }
        if(!changed){ //若没有发现当前度数为 1 的顶点
            foreach(v∈V(T)){
                /*如果仍有未被标记的顶点，则该顶点事实上是 T 的中心，
                这种情况只会在 T 有唯一的中心时发生。*/
                if(m(v)==-1)
                    m(v)=m;
            }
```

```
                break; //退出 while 循环
            }
            S=∅;
            m=m+1;
        }
    }
```

在图 5 中，我们也给出了基于上述算法得到的各顶点的 m 值，可见自由树的中心确实具有最大的 m 值。

基于性质 5，我们给出树模式挖掘阶段的模式的规范标记如下：

定义 26 树模式挖掘阶段的模式的规范标记

设 T 是树模式挖掘阶段的一个模式，记 $L(T)$为 T 的所有顶点标记所组成的集合。在 $L(T)$上定义偏序关系$\prec$为词典序，并记定义了$\prec$的 $L(T)$为 $L(T,\prec)$。根据 T 的中心数目分两种情况来定义 $c(T)$。

（1）若 T 有唯一的中心 r，则以 r 为根建立一棵有根树 T'，使得有根树的每一层互为兄弟的节点之间按节点标记的词典序$\prec$由小到大排列。设ε表示空标记，L_b为一个特殊的标记，满足 $L_b \notin L(T,\prec), \forall T$。用 L_i 表示 T'的第 i 层，$\forall v \in L_i$，若 v 没有孩子（即 v 为叶节点），则给 v 增加一个孩子节点 $child(v)$,规定 $child(v)$ 的标记为 L_b。T 的规范标记 $c(T)$定义为，

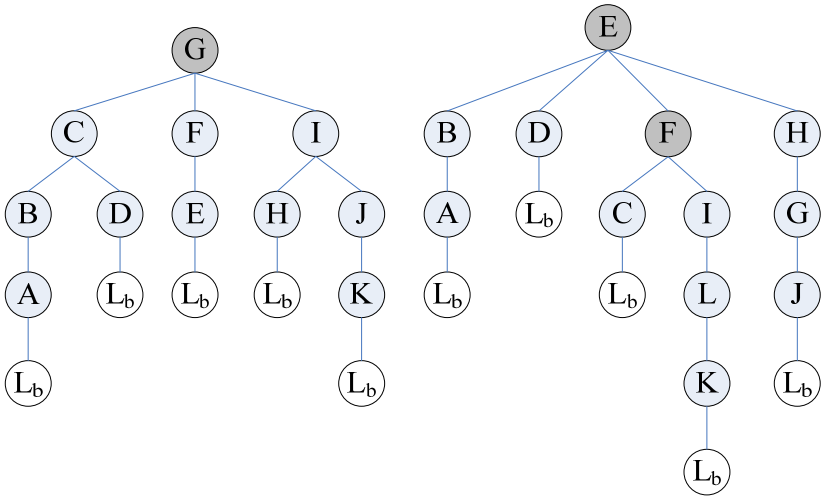
$c(T)=c(L_0)c(L_1)\ldots c(L_{h(T')})$。设第 i 层的顶点数目为 $n(i)$，则 $c(L_i)=l(v_{i1})t_{i1}l(v_{i2})t_{i2}\ldots l(v_{in(i)})t_{in(i)}$，$v_{ik}$为第 i 层的第 k 个顶点（$k\in\{1,2,\ldots,n(i)\}$），$l(v_{ik})$表示 v_{ik} 的标记，t_{ik} 的定义为：$t_{ik}=\varepsilon$，若 v_{ik} 与 $v_{i(k+1)}$为兄弟节点（$k\neq n(i)$）或者 $l(v_{ik})=L_b$；否则 $t_{ik}=L_b$。

（2）若 T 有两个中心 r_1,r_2,则若 $r_1\prec r_2$，取 $r=r_1$，否则取 $r=r_2$。然后将 r 视为 T 的唯一中心，按照与（1）中相同的方式来定义 $c(T)$。

把上述定义运用到图 5 中，对于图 5(a)，(b)中的自由树，我们可以分别得到如图 6 中(a)，(b)所示的有根树，从而得到对应的规范标记为：

$C(T_{(a)})=GL_bCFIL_bBDL_bEL_bHJL_bAL_bL_bL_bL_bKL_bL_bL_b$

$C(T_{(b)})=EL_bBDFHL_bAL_bL_bCIL_bGL_bL_bL_bLL_bJL_bKL_bL_bL_b$



(a) (b)

图 6　图 5 中自由树的规范标记

具体计算该规范标记的算法可以用伪码表示为：

算法 12　计算树模式 *T* 的规范标记

```
ComputeTreeCanonicalLabel(T){
    queue Q;
    C=FindCenters(T); //找到 T 的中心
    RootedTree T'; //构造与 T 对应的有根树 T'
    if (|C|==1)
        T'=ConstructTreeWithRoot(C[0],T);
    else{
        if (C[0]≺C[1])
            T'= ConstructTreeWithRoot(C[0],T);
        else
            T'= ConstructTreeWithRoot(C[1],T);
    }
    //通过对 T'进行层次遍历的方式来得到 T 的规范标记 c(T)
    Q.push(T'.root);
    Q.push(NULL);
    while(!Q.empty()){
        u=Q.front();
        Q.pop();
        if (u==NULL)
            c(T).add(Lb);
```

```
            else{
                c(T).add(l(u));
                foreach(v∈Children(u) ) //Children(u)为 u 的所有孩子组成的集合
                    Q.push(v);
                Q.push(NULL);
            }
        }
    }

    ConstructTreeWithRoot(r, T){ //以 r 为根构造与 T 对应的有根树 T'
        RootedTree T';
        new RootedTreeNode r';
        l(r')=l(r);
        r'.parent=NULL;
        hash_table visited;
        ConstructTree(r',r, visited,T);
        T'.root=r';
        return T';
    }

    ConstructTree(u',u, visited,T){
        visited.add(u); //标记 u 为已访问过
        foreach(v∈N_T(u) ){
            if(!visited.contains(v)){ //若 v 尚未被访问过
                new RootedTreeNode v';
                l(v')=l(v);
                v'.parent=u';
                u'.addChild(v'); //设定 u 和 v 为父子关系
                ConstructTree(v',v,visited,T); //对 v 进行递归
            }
        }
    }
```

定理 1 定义 25 中定义的 *c(T)*是 *T* 的一个规范标记。

证明：

设 T_1 和 T_2 是树模式增长阶段的两个频繁模式，我们需要证明的是 $T_1 \cong T_2$ 当且仅当 $c(T_1)=c(T_2)$。

$\Rightarrow$：若 $T_1 \cong T_2$，则由 T_1 和 T_2 构造的有根树 T_1'和 T_2'也满足 $T_1' \cong T_2'$，因此 $c(T_1)=c(T_2)$。

$\Leftarrow$：若 $c(T_1)=c(T_2)$, 事实上我们可以通过下面的算法从 $c(T)$出发构造唯一的一棵有根树 T'，因此有 $T_1' \cong T_2'$。设 T_1''和 T_2''为去掉 T_1'和 T_2'中所有标记为 L_b 的节点后得到的有根树，则显然有 $T_1'' \cong T_2''$。现在我们把 T_1''和 T_2''视为自由树，显然有 $T_1 \cong T_1''$，$T_2 \cong T_2''$，因此 $T_1 \cong T_2$。

从 $c(T)$出发构造唯一的一棵有根树 T'的算法如下：

```
ConstructTreeFromLabel(c(T)){
    T';
    T'.root=r;
    l( r)= c(T)[0];
    queue Q;
    Q.push(r);
    i=2; //跳过第一个 Lb 标记
    while(!Q.empty()){
        u=Q.front();
        Q.pop();
        while(c(T)[i]!=Lb){
            new v;
            l(v)= c(T)[i];
            Q.push(v);
            u.addChild(v); //v 为 u 的一个孩子节点
            i=i+1;
        }
        i=i+1; //跳过 Lb 标记
    }
}
```

4.3.2 图模式挖掘阶段的规范标记

定义 27 图模式挖掘阶段的模式的规范标记

设 G 是图模式挖掘阶段的一个频繁模式，记 $L(G)$为 G 的所有顶点标记所组

成的集合。类似地，在 $L(G)$上定义偏序关系$\prec$为词典序，并记定义了$\prec$的 $L(G)$为 $L(G,\prec)$。设 G 的顶点数为 n，$l(v)$为 G 中顶点 v 的标记，将 $V(G)$中的顶点排列为 $v_1,v_2,\ldots,v_n$，使得 $l(v_1)\prec l(v_2)\prec\ldots\prec l(v_n)$，构造矩阵 $A(G)=(a_{ij})_{n\times n}$，使得 $a_{ij}=l(v_i)$，若 $i=j$；否则若$(v_i, v_j)\in E(G)$，则 $a_{ij}=1$，若$(v_i, v_j)\notin E(G)$，则 $a_{ij}=0$。取 A 的主对角线及其以下的元素组成的下三角阵 A'，设 $A'(G)=(\alpha_1,\alpha_2,\ldots,\alpha_n)$，$G$ 的规范标记 $c(G)$定义为：$c(G)=c(\alpha_1)c(\alpha_2)\ldots c(\alpha_n)$，$c(\alpha_i)=a_{ii}a_{i(i+1)}\ldots a_{in}$，$i\in\{1,2,\ldots,n\}$。

例如，对于图 3 中的模式 X，$c(X)=A111B01C1D$。

具体计算该规范标记的算法可以用伪码表示为：

算法 13 计算图模式 G 的规范标记

```
ComputeGraphCanonicalLabel(G){
    n=|V(G)|;
    sort V(G) so that V(G)={ v1,v2,…,vn}, l(v1)≺l(v2)≺…≺l(vn);
    for(i=1; i≤n; i=i+1){
        c(G).add(l(vi);
        for(j=i+1; j≤n; j=j+1){
            if( (vj∈NG(vi) ) //如果(vi,vj)∈E(G)
                c(G).add("1");
            else
                c(G).add("0");
        }
    }
}
```

定理 2 上面定义的 $c(G)$是 G 的一个规范标记。

证明：设 G_1 和 G_2 是图模式增长阶段的两个频繁模式，我们需要证明 $G_1\cong G_2\Leftrightarrow c(G_1)=c(G_2)$。显然，$G_1\cong G_2\Leftrightarrow A(G_1)=A(G_2)$，由于 $A(G_1)$和 $A(G_2)$是对称矩阵，故$A(G_1)=A(G_2)\Leftrightarrow A'(G_1)=A'(G_2)$, 而根据 $c(G)$的定义，显然又有 $A'(G_1)=A'(G_2)\Leftrightarrow c(G_1)=c(G_2)$。因此 $G_1\cong G_2\ \Leftrightarrow c(G_1)=c(G_2)$。

4.4 实际算法的时间代价分析

这一节里，我们结合 4.3 节中提出的几点改进对 4.2 节中给出的算法框架进行一个简单的时间代价的分析。由于整个数据集中频繁模式的数量是未知的，所以我们下面将讨论集中在对某个有效的当前模式进行模式扩展并得到所有候选

模式的时间代价上。这主要分为两部分的工作：对当前模式进行有效性（同构）判断，构造当前模式的外（内）模式集合。和前面一样，我们分树模式挖掘和图模式挖掘两个阶段分别进行分析。

4.4.1 树模式挖掘阶段的时间代价

设 T 是树模式挖掘阶段的一个的频繁模式。

（1）计算 T 的规范标记 ***ComputeTreeCanonicalLabel(T)***的时间代价可以分为以下几步来衡量：

寻找 T 的中心 ***FindCenters(T)***，时间代价为 $O(V(T)^2)$;

构造与 T 对应的有根树 T' ***ConstructTreeWithRoot(r, T)***，时间代价为 $O(V(T)+E(T))$;

通过对 T'进行层次遍历来得到 $c(T)$，时间代价为 $O(V(T)+E(T))$;

所以 ***ComputeTreeCanonicalLabel(T)*** 的时间代价为 $O(V(T)^2)+ O(V(T)+E(T))= O(V(T)^2)+O(V(T))= O(V(T)^2)$。

（2）构造 T 的外模式集合的时间代价可以分为以下两步来衡量：

进行第一类外模式扩展 ***BuildOuterPatternSet (T)***，时间代价为 $O(|CP(T.parent)_{outer}| \times |T.occurrence\_list| \times max\{|H.occurrence\_list| \mid H \in CP(T.parent)_{outer}|\})$。

进行第二类外模式扩展 ***ExtendOuterEdges (T)***，时间代价为 $O(|T.occurrence\_list| \times max\{|\ PL_{(l,h),X}(\ t.path.to\ )| \mid t \in T.occurrence\_list,\ X=D(t.graph\_id)\})$。

所以构造 T 的外模式集合的时间代价为上面两步之和。

4.4.2 图模式挖掘阶段的时间代价

设 G 是图模式挖掘阶段的一个的频繁模式。

（1）计算 G 的规范标记 ***ComputeGraphCanonicalLabel(G)***的时间代价为 $O(V(G)^2)$;

（2）构造 G 的内模式集合 ***BuildInnerPatternSet (G)***的时间代价为 $O(|CP(G.parent)_{inner}|\ \times |G.occurrence\_list| \times max\{|H.occurrence\_list| \mid H \in CP(G.parent)_{inner}|\})$。

第五章 实验结果与分析

我们用 C++对第四章中给出的算法进行了实现，编译环境为 Microsoft VC++ .NET 编译器，操作系统环境为 WindowsXP。实验用的 PC 机的配置情况为，CPU: AMD2600+，1.91GHz；Memory Size : 512MB。

在这一章里，我们将进行一系列实验，从而对算法的实际性能进行一个更深刻的了解和评估。

5.1 实验数据说明

实验数据集分为三组，一组是自定义的模拟 PPI 网络特点的人工数据集，以下简记为 *AD*；另外两组是真实的 PPI 网络数据集，以下简记为 *RD1* 和 *RD2*。为了简单起见，同时为了更清楚地描述实验结果，设$|AD|=|RD1|=|RD2|=2$。我们将会看到，即使是如此简单的数据集，也已经能够充分地说明问题的复杂程度了。

1）对 *AD* 的描述：

我们以图 3 中的两张图 Y_1 和 Y_2 作为基础来建立 *AD* 的两张数据图 AD_1 和 AD_2。由于真实的 PPI 网络是唯一标记的，因此我们首先需要对 Y_1 和 Y_2 的顶点进行重新标记来使得 AD_1 和 AD_2 的顶点标记唯一。另外，如第三章所介绍的那样，真实 PPI 网络之间顶点的是否可以匹配是由评分表来决定的，因此我们还需要构造评分表。为了简单起见，我们设 Y_1 和 Y_2 中凡是具有相同标记的顶点都是可以匹配的，并且我们统一所有的相似度值为 20，而在涉及 *AD* 的任何一个实验中，我们设定评分表的相似度阈值为 δ =20。下面我们给出经过上述步骤后得到的 AD_1 和 AD_2 以及相应的评分表 $SL(AD_1,AD_2)$。（图 7）

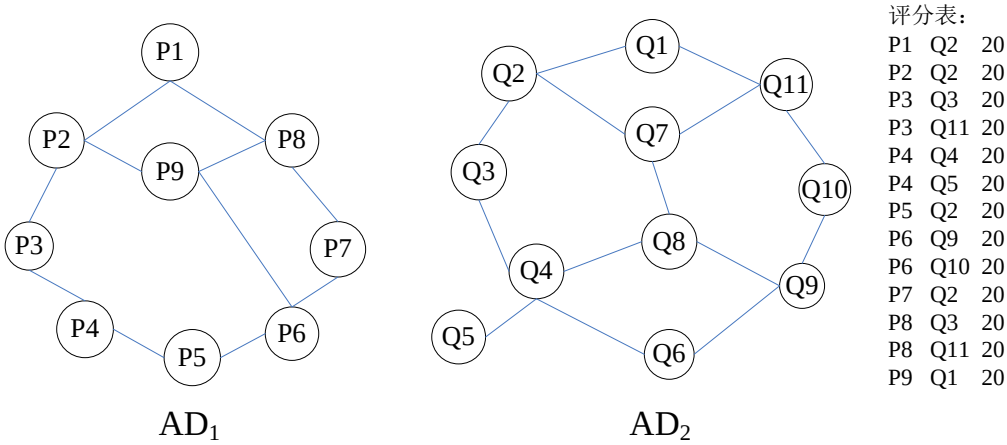


图 7 人工数据集 *AD*

2）对 *RD1* 的描述：

取 $RD1_1$ 为图 1*(a) Drosophila Melanogaster*（果蝇的 PPI 网络图），有 404 个顶点, 481 条边；取 $RD1_2$ 为图 1*(c)Saccharomyces Cerevisiae*（酵母的 PPI 网络图），有 2187 个顶点, 4837 条边。评分表 $SL(RD1_1,RD1_2)$通过 BLAST 计算获得，关于 BLAST 的介绍将在第六章中进行，这一点前面也已经提到过。最终获得的评分表的大小为 330KB，包含 18745 个可匹配顶点对。为了有针对性地设置实验参数，我们统计了 $SL(RD1_1,RD1_2)$中相似度值的分布情况（图 8）：

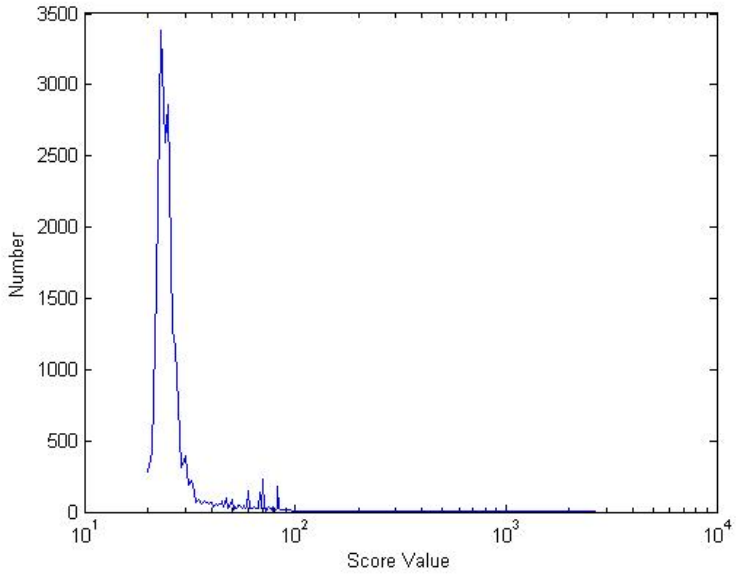


图 8　　$SL(RD1_1,RD1_2)$中相似度值的分布

3）对 *RD2* 的描述：

取 $RD2_1$ 为图 1*(b)Caenorhabditis elegans*（蛔虫的 PPI 网络图），有 1722 个顶点, 2384 条边；取 $RD2_2$ 同样为图 1*(c)*。评分表 $SL(RD2_1,RD2_2)$同样通过 BLAST 计算得到。获得的评分表的大小为 249KB，包含 14138 个可匹配顶点对。$SL(RD2_1,RD2_2)$中相似度值的分布情况如图 9 所示：

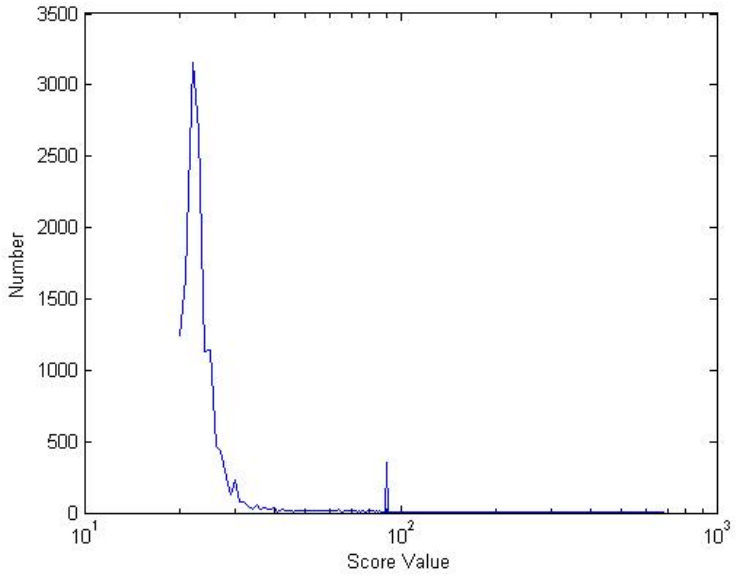


图 9　　$SL(RD2_1,RD2_2)$中相似度值的分布

从图 8 和图 9 我们看到，评分表中的相似度值的分布是不均衡的，绝大部分的相似度值在 50 以下，超过 100 的相似度值非常少。而这些高相似度的顶点对反映了对应蛋白质之间的高相似度，恰好是我们进行挖掘时所应特别关注的，所以在进行真实 PPI 网络图集的挖掘时相似度阈值δ不宜取得过低。

5.2 实验结果描述与分析

为了方便起见，在以下的实验中，我们将采用以下的一些符号来表示相关的实验结果：

t_{fe}:　　挖掘频繁边（规模为 1 的模式）所花的时间，单位为秒*(s)*，精确到 0.1*s*；

n_{fe}:　　找到的频繁边的数目；

t_{fp}:　　树模式和图模式挖掘阶段所花的时间，单位为秒*(s)*，精确到 0.1*s*；

n_{fp}:　　找到的频繁模式（包括频繁边）的数目；

S_{max}：　　找到的规模最大的频繁模式的大小（边数）。

实验参数 l, h,θ,δ 的含义见前面章节的说明，正如前面所说的那样，θ值统一取为θ=|*D*|=2。另外在统计模式的数量时，我们统计的是带标记模式的数量，而不考虑忽略标记后这些模式之间的同构性；在统计模式挖掘的时间时，我们也不考虑忽略标记后合并同构模式所需要的时间。这是因为带标记模式的挖掘是整个算法的核心部分，而最后的同构模式处理则只不过是模式挖掘过程结束后的一个后续工作。

我们首先使用 *AD* 对算法进行了测试以保证算法的正确性。然后我们在不同

的数据集上进行了一系列实验：

实验 1 不同δ值对实验结果的影响（固定 l=1，h=2）

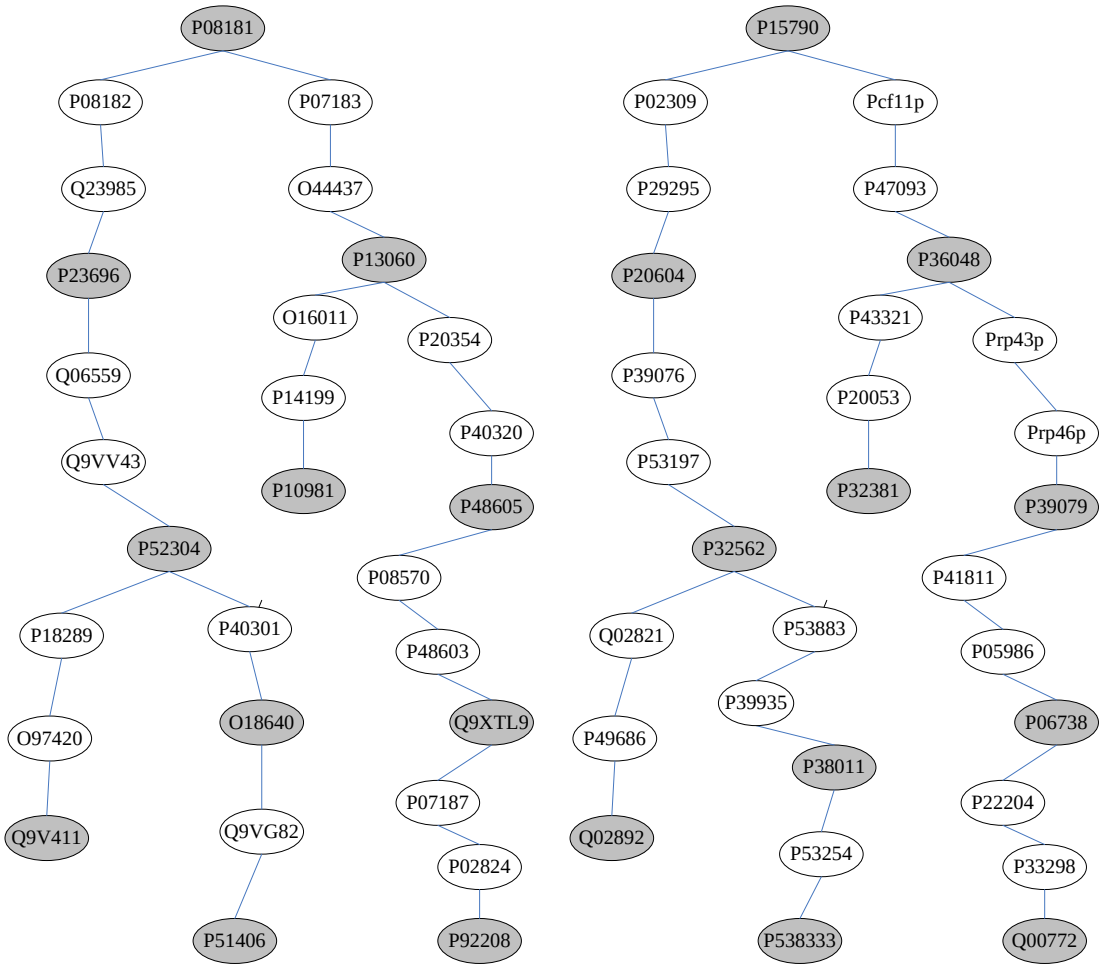
1）表 1 显示了在数据集 *RD1* 上进行的部分实验结果：

δ值 / 结果	300	250	200
t_{fe}	3.9	5.2	8.6
n_{fe}	7	19	40
t_{fp}	0.1	35.4	1581
n_{fp}	10	94	520
S_{max}	2	6	10

表 1　　数据集 *RD1* 上的实验结果

可以看到当δ=200 时，t_{fp}已经相当巨大。事实上，当试图再次减小δ时，就我们的 PC 配置而言，将由于内存不足而导致程序无法运行完，我们只好放弃进一步的尝试。

图 10 展示了我们在 *RD1* 上找到的一个规模最大的模式（10 条边）分别在 $RD1_1$和 $RD1_2$中的某个出现。它对我们前面提出的一个猜想在一定程度上作出了验证，即真实 PPI 网络图集中许多频繁模式是以自由树的形式存在的，从而两阶段挖掘的框架在 PPI 网络的实际应用中是合理的。



(a) Occurrence in $RD1_1$ (b) Occurrence in $RD1_2$

图 10 *RD1* 中发现的某个规模为 10 的模式在 $RD1_1$ 和 $RD1_2$ 中的出现

2）表 2 显示了在数据集 *RD2* 上进行的部分实验结果：

δ 值 / 结果	130	120	110	100	91
t_{fe}	6.2	8.6	10.9	13.8	17.5
n_{fe}	10	17	20	24	24
t_{fp}	0.8	2.3	3.4	24.2	49.5
n_{fp}	15	47	57	145	166
S_{max}	2	5	5	7	7

表 2 数据集 *RD2* 上的实验结果

但是当我们设定 δ=90 时，出乎意料的是，我们在半个小时之内都没有看到程序结束，而且最终竟然也遇到了内存不足的问题。这说明程序对于 δ 的取值是

非常敏感的。那么，从δ=91 变化到δ=90 时究竟发生了什么事情呢？

我们统计了δ=91 和δ=90 时相应的$SL(RD2_1,RD2_2,\delta)$中$RD2_1$和$RD2_2$可匹配的顶点数目，得到了以下结果：

δ=91 时，$RD2_1$中有 22 个可匹配的顶点，$RD2_2$中有 115 个可匹配的顶点；

δ=90 时，$RD2_1$中有 23 个可匹配的顶点，$RD2_2$中有 439 个可匹配的顶点。

我们惊奇地发现，$RD2_1$中只是增加了一个可匹配的顶点，$RD2_2$中的可匹配顶点数目却增加了将近 3 倍。那么$RD2_1$中究竟增加了哪个顶点呢？经过比较我们发现所增加顶点的标记为 Q18688。我们检索了$SL(RD2_1,RD2_2)$中所有相似度值为 90 的匹配对，发现在所有这样的 349 个匹配对中竟然有 343 个匹配对的第一个顶点为 Q18688。难怪一旦这个顶点被排除后$RD2_2$中的可匹配顶点数会如此明显地减少了。这已经可以解释我们为什么会遇到前面的问题了，因为一旦这个顶点被考虑，则我们在模式挖掘的任何一步一旦扩展到这个顶点，就必须考虑所有与之相匹配的顶点的所有可能扩展情况，因此时间代价会迅速增长。而如果这些可能的匹配确实是真正的匹配，则存储的空间代价也会迅速增长（空间代价的问题后面还要讨论）。

另外需要注意的是，相似度分布在相似度为 90 的地方产生了一个突变（这从图 9 中也可以很明显地看出来），因此这相似度为 90 的 349 个顶点的增加对$SL(RD2_1,RD2_2,\delta)$的规模的影响是相当显著的，这也能部分地解释我们遇到的问题。

3）为了进一步地探寻不同δ值的设置对实验结果产生的影响的规律性，我们在*RD2* 上将δ值在区间[90,200]上以 10 为步长进行了一系列实验，得到了以下的变化曲线图：（由于δ=90 时程序无法运行出结果，因此以下各曲线的第一点都是对应于δ=91 时的值）

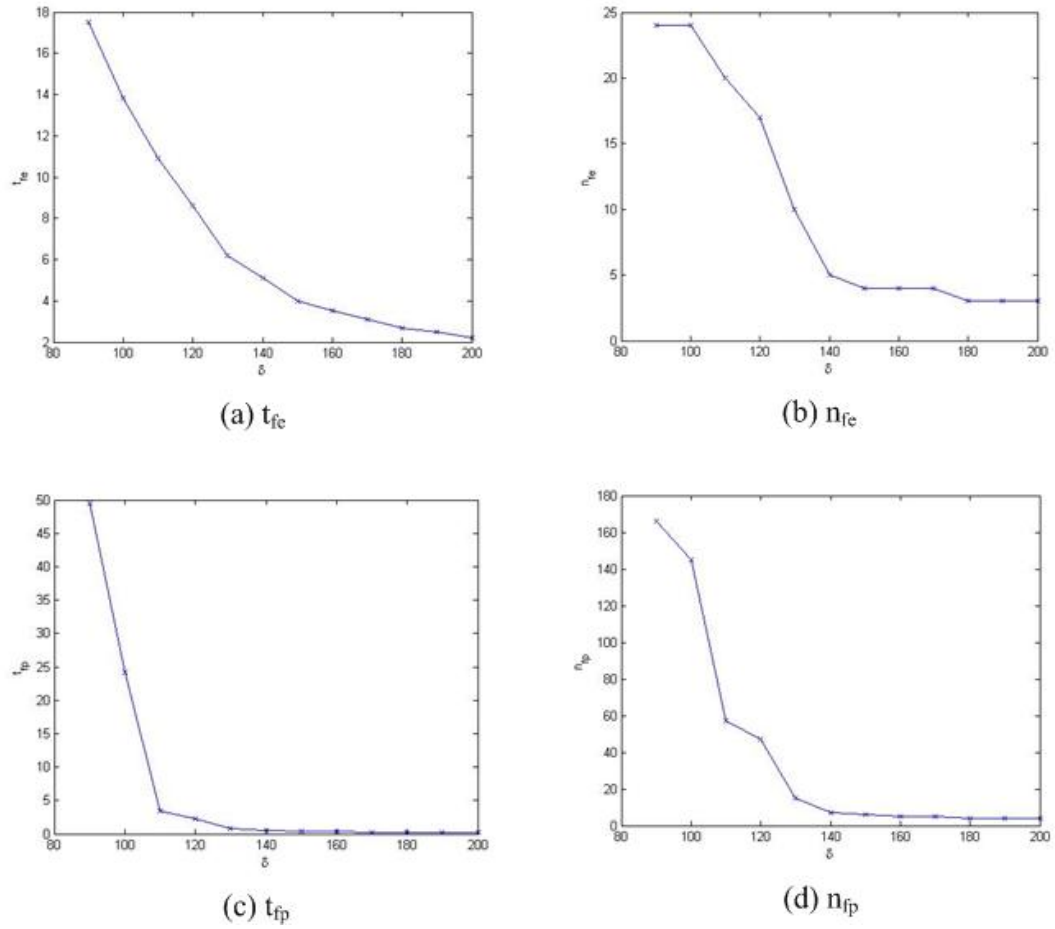


图 11　t_{fe}, n_{fe}, t_{fp}, n_{fp} 随 δ 值的变化情况

我们发现，随着 δ 值的增加，t_{fe} 呈逐渐下降的趋势，而 n_{fe}, t_{fp}, n_{fp} 先迅速下降至某个值，然后下降趋势变得非常缓慢。这是因为 t_{fe} 的计算直接依赖于 δ-评分表的大小，而一般来说，δ 值越大，δ-评分表的规模就越小，所以 t_{fe} 也随之减小。而 n_{fe}, t_{fp}, n_{fp} 的大小取决于实际的匹配情况，但实际的匹配情况一定程度上取决于 δ 值的分布情况，因此其对于 δ 值的变化的反应是不均衡的：δ 值的某些增加可能会造成某些重要顶点的失配从而造成频繁模式数量的急剧减少（如我们在 2）中所分析的情况）；而另一方面，在某些范围内 δ 值的变化却可能对实际的匹配情况没有任何影响，从而不影响频繁模式的数量和挖掘时间。

通过实验 1 我们至少可以得到以下结论：

在固定 l，h 的前提下，

1）虽然一般来说，δ 值越大，δ-评分表的规模越小，但是不同的 δ 值处的相同变化量对 δ-评分表规模变化的影响却有着很大的差异。

2）算法的实际运行时间与数据图的规模有直接关系，但是评分表的规模对之影

响更大。比较 *RD1* 和 *RD2* 我们不难发现，$RD2_1$ 的规模要比 $RD1_1$ 大许多，但是由于 *RD2* 的评分表规模要比 *RD1* 的评分表规模小不少，因此在同样的 PC 配置条件下，如果取同样的δ值的话，在程序可以正常结束的前提下 *RD2* 上的挖掘要明显快于 *RD1* 上的挖掘。这是因为评分表的规模决定了候选模式的数量，主导了挖掘的主要过程（模式枚举），从而对算法的运行时间起了根本性的支配作用。

实验 2 不同 *l* 和 *h* 值对实验结果的影响（固定δ值）

1）表 3 显示了在 *RD1* 和 *RD2* 上得到的部分实验结果：

参数 / 结果	*RD1*,*l*=2,*h*=3, δ=350	*RD2*,*l*=2,*h*=3, δ=295	*RD2*,*l*=2,*h*=3, δ=290	*RD2*,*l*=2,*h*=3, δ=287
t_{fe}	15.4	10.3	10.8	11.6
n_{fe}	21	4	8	8
t_{fp}	100.8	0.6	1.6	2.2
n_{fp}	77	6	15	15
S_{max}	4	2	3	3

表 3　　*RD1* 和 *RD2* 上的实验结果

很显然，上述实验数据并不符合我们的初衷，我们的本意是在固定δ值下对不同的 *l* 和 *h* 值进行实验。事实上，我们曾经按照这个设想进行了许多实验。（比如在 *RD2* 上，我们曾试图固定δ=100，*(l,h)*分别取(1,3)，(2,3)，(3,4)进行实验）但是很遗憾，这些实验都无法得到结果，问题还是同样：内存不足。上面给出的是在数十次实验中的几次程序能够正常终止时得到的结果。从这有限的结果数据中我们可以看到一个明显的特点，即δ的取值非常大。从实验 1 我们已经知道δ的取值直接决定δ-评分表的规模，而δ-评分表的规模对算法性能有决定性影响。提高δ值意味着减小δ-评分表的规模，因此我们大部分实验最终无法得到结果的根本原因就是在相同δ值下，*l* 值或 *h* 值的增加导致了问题规模的大幅增加。这主要是因为增大 *l* 值或者 *h* 值使得*(l,h)*路径的数量大大地增加了，（最坏情况下，在某个范围内，路径数量是随着路径长度呈指数增长的，特别是在 PPI 网络这样的大图中，这种增加更为突出）从而使得候选模式的数量也大大地增加了。当然，我们可以肯定，如果 *l* 值取得非常大，那么路径的数量肯定不会很多（因为图的规模再怎么大也总是有限的），但是此时寻找这些*(l,h)*路径的时间代价已经无法令人忍受了（看一下第四章中构造图中所有顶点的*(l,h)*路径表的算法以及考虑一下真实 PPI 网络图的规模就知道了）。

2） 为了能够寻找到在固定δ值的条件下，不同 *l* 和 *h* 值对算法性能的影响的规律性，我们另外在 *AD* 上进行了一系列实验。*AD* 的优势在于规模小，因此不会出现内存不足的问题，而且就上述实验目的而言，在 *AD* 上进行实验不会对我们发现其中的规律性造成影响。

表 4，表 5 和表 6 给出了相关的试验结果（δ=20，5.1 节已经作了说明）：

参数 / 结果	*l*=1, *h*=2	*l*=1, *h*=3	*l*=1, *h*=4	*l*=1, *h*=5	*l*=1, *h*=6
t_{fe}	0.0	0.1	0.2	0.3	0.4
n_{fe}	24	26	30	30	30
t_{fp}	0.7	1.7	3.8	6.4	10.0
n_{fp}	112	128	138	138	138
S_{max}	5	5	5	5	5

表 4　*AD* 上的实验结果 1（固定 *l*=1，*h* 由小到大变化）

参数 / 结果	*l*=1, *h*=6	*l*=2, *h*=6	*l*=3, *h*=6	*l*=4, *h*=6	*l*=5, *h*=6
t_{fe}	0.4	0.4	0.3	0.3	0.2
n_{fe}	30	30	30	29	28
t_{fp}	10.0	2.9	0.2	0.1	0.0
n_{fp}	138	51	30	29	28
S_{max}	5	2	1	1	1

表 5　*AD* 上的实验结果 2（固定 *h*=6，*l* 由小到大变化）

参数 / 结果	*l*=1, *h*=2	*l*=2, *h*=3	*l*=3, *h*=4	*l*=5, *h*=6	*l*=7 *h*=8
t_{fe}	0.0	0.1	0.1	0.2	0.2
n_{fe}	24	26	29	28	6
t_{fp}	0.7	0.4	0.0	0.0	0.0
n_{fp}	112	43	29	28	6
S_{max}	5	2	1	1	1

表 6　*AD* 上的实验结果 3（*l, h* 均由小到大变化）

通过实验 2 我们至少可以得到以下结论：

在固定δ的条件下，

1）在真实的 PPI 网络中无论 *l* 值还是 *h* 值发生变化都会对算法性能产生较大的影响。

2）当固定 l 时，t_{fe}, n_{fe}, t_{fp}, n_{fp} 将随着 h 的增加而增加；当固定 h 时，t_{fe}, n_{fe}, t_{fp}, n_{fp} 将随着 l 的增加而减小。

3）当 l, h 都逐渐增加时，t_{fe} 呈逐渐增大的趋势，而 n_{fe} 呈现先增大后减小的趋势，t_{fp}, n_{fp} 的变化取决于 n_{fe} 和具体的数据图的特点。虽然从表 6 来看似乎 t_{fp} 和 n_{fp} 呈递减趋势，但由于第 2 列至第 5 列的 S_{max} 太小，我们无法肯定这一趋势是否是由于实验数据本身的特殊性所造成的。

最后，我们来对算法运行的空间代价作一个简单的分析。我们已经多次提到，在我们前面的实验过程中，我们经常遇到内存不足而导致程序无法正常结束。为了了解其中的原因，我们在挖掘过程结束后输出了所找到的规模最大的那些模式以及它们在数据图中的所有出现。我们得到了以下一些结果：

RD1, l=1, h=2, δ=200, 输出文件的大小约为 25.9MB。

RD2, l=1, h=2, δ=91, 输出文件的大小约为 11.5MB；

δ=200 和 δ=91 分别为我们固定 l=1, h=2 进行实验时程序能够正常结束和不能够正常结束的临界值，输出文件的规模是令人吃惊的，因为我们输出的仅仅是那些规模最大的模式。为了弄清情况，我们检查了上述两个文件，发现在第一个文件中只有 5 个规模为 10 的模式，但每个模式的出现分别有 31241，209，24789，11565，15773 次；而在第二个文件中，我们发现也只有 3 个规模为 7 的模式，但每个模式的出现分别有 91，12520，38769 次。不少模式的出现次数是惊人的，而考虑到这些模式的所有子模式都是频繁模式，就不难理解为什么会出现内存不足的问题了。但是就我们的挖掘任务而言，我们无法在找到频繁模式后就丢弃它的出现记录表，因为最终模式在数据图中的出现才是我们所真正关注的。导致模式出现如此之多的根本原因仍然是 δ-评分表的规模。因此就目前而言，我们暂时无法在实践中避免空间代价很大的问题，因为我们确实需要这样大的空间来存储我们的频繁模式以及它们的出现记录表。如果我们想更进一步的话，我们必须考虑更高效的压缩存储方式或者对挖掘任务的定义作一些修改，这些在第七章中还会提到。

第六章 算法在Pathway系统中的实际应用

在这一章里，我们将结合 Pathway 图频繁模式挖掘系统来看看算法的实际应用价值。6.1 将对 Pathway 系统进行一个概要的介绍；6.2 节具体地讨论一下 Pathway 系统的设计与实现方面的情况，包括系统的架构以及各主要模块功能与流程的简要描述；6.3 将给出算法的实际应用效果。

6.1 Pathway系统简介

Pathway 是一个用于解决 PPI 网络频繁模式挖掘问题的系统，它所要挖掘的频繁模式即为本文所定义的图的拓扑结构。设计该系统的目的在于将 PPI 数据预处理的过程以及频繁模式挖掘的过程结合到一起，并对挖掘的结果用图形化的方式表示出来，从而实现大规模 PPI 网络挖掘的完全自动化。因此，从功能上说，Pathway 系统可以划分为三个模块，即数据预处理模块，挖掘算法模块和图形化表示模块。其中，挖掘算法模块是整个系统的核心部分。然而，在对 Pathway 系统的第一个实现版本进行测试时，我们发现其挖掘算法模块的效率完全无法满足实际的需要。因此展开本文所讨论的研究工作的一个初衷就是设计一个能够满足实际需要的算法来替换原来的挖掘算法，从而使 Pathway 系统真正具有实用价值。现在看来，我们已经在一定程度上使问题得到了解决。当然，为了使系统更加完善，我们今后还需要对算法进行改进，进一步提高其效率，可行的改进策略将在第七章予以讨论。

6.2 Pathway系统的设计与实现

6.1 中已经提到，Pathway 系统从功能上来看主要分为三个模块：数据预处理模块，挖掘算法模块和图形化表示模块。在这一节里，我们首先给出系统的整体架构图，然后分别针对这三个模块的设计与实现进行简要介绍。

6.2.1 系统的整体架构

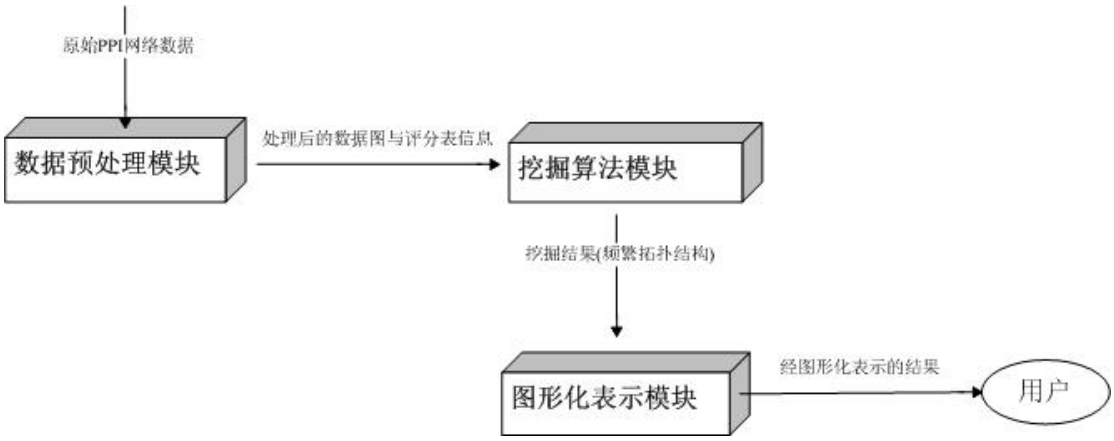


图 12 Pathway 系统的整体架构

6.2.2 数据预处理模块

数据预处理模块包括两部分的功能：1）原始 PPI 网络数据格式的转化；2）顶点蛋白质之间同源相似度的计算。

6.2.2.1 原始 PPI 网络数据格式的转换

从 DIP 等 PPI 网络数据库中得到的数据有多种格式，有 XML 格式的，也有文本格式的。即使同为文本格式，不同年代的档案数据也存在着细微的差别。蛋白质标识符有多套，来源于不同实验的数据可能使用不同的标识，当这些实验结果合并成整个物种 PPI 网络数据时，不同的标识系统就会混杂在一起。因此，需要首先对这些混杂的格式进行转换，使用统一的标识系统，从而得到一致的格式。

在 Pathway 系统的实现中，将所有的格式转换成简单的文本格式，一个文件对应于一个物种的 PPI 网络。文件的第一行表示在此 PPI 网络中出现的所有蛋白质，蛋白质标识符之间以空格隔开；从第二行起每行包含用空格隔开的两个蛋白质标识符，表示两个蛋白质之间的相互作用。另外，每个蛋白质的标识符需要与蛋白质序列数据库中的标识符相一致，因为后面我们需要使用 BLAST 工具软件来计算两个蛋白质之间的同源相似性。

6.2.2.2 顶点蛋白质之间同源相似度的计算

统一了 PPI 网络数据的表示格式后，在进行挖掘之间，我们还需要得到相应

的评分表的信息。评分表的每一行表示了两个蛋白质之间的同源相似度信息。前面已经提到过，我们将使用 BLAST 工具软件来计算蛋白质之间的同源相似度。具体的计算过程如下：

首先根据蛋白质的标识符到蛋白质序列数据库（如 swissprot）获得要比对的物种的所有蛋白质序列，放入一个文件，用这个文件使用 formatdb（BLAST 工具集中的一个程序）构建数据库。然后用这个文件作为查询输入文件在新建的数据库中使用 BLAST 工具集中的有关程序（如 blastall）进行同源搜索，得到报告文件。最后解析这个报告文件，便可以得到蛋白质序列之间的同源相似度值。对于没有分值的蛋白质对，则分配一个固定的分值，这个分值可以由用户指定（如在第五章的一系列实验中，我们指定这个分值为 20）。

6.2.3 挖掘算法模块

这一模块主要用来封装实际的模式挖掘算法。为了便于算法的更新和替换，设计时将这一模块完全地独立了出来，模块的输入输出都采取文件读写的方式来进行，从而基本与前后模块之间处于无相互耦合的状态。由于我们前面已经花了巨大的篇幅讨论了我们算法的设计与实现，因此这里不再赘述。

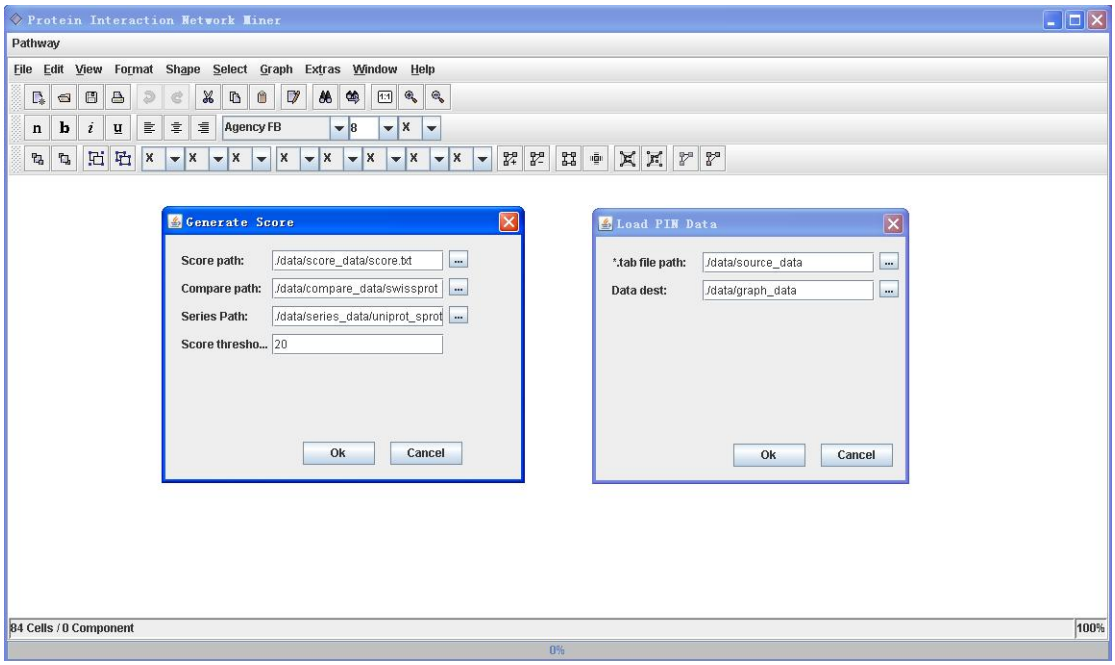
6.2.4 图形化表示模块

在 Pathway 系统中，图形化表示是其的一项重要功能，这一模块需要解决的主要问题是：对于一张给定的图，如何将它在有限的屏幕空间上合理地进行布局？我们从图 1 中已经看到，如果给定的图的规模非常大，那么这是一个十分复杂和困难的问题。事实上，也有许多人已经或正在研究这个问题，出现了不少关于图的布局算法。由于图的布局问题与本文所研究的问题无关，因此我们这里不再展开讨论。目前在 Pathway 系统中，我们只需要将最后产生的模式进行图形化的表示，而一般而言模式的规模不会很大，因此 Pathway 系统采取了一些相对简单的启发式策略来处理这个问题。当然，在今后对 Pathway 系统进行完善时，我们会考虑植入更有效的图布局算法，也正是出于这样的考虑，目前在图形化表示模块中，涉及图的布局算法的部分被设计为一个单独的方法，以便于日后的扩展和替换。

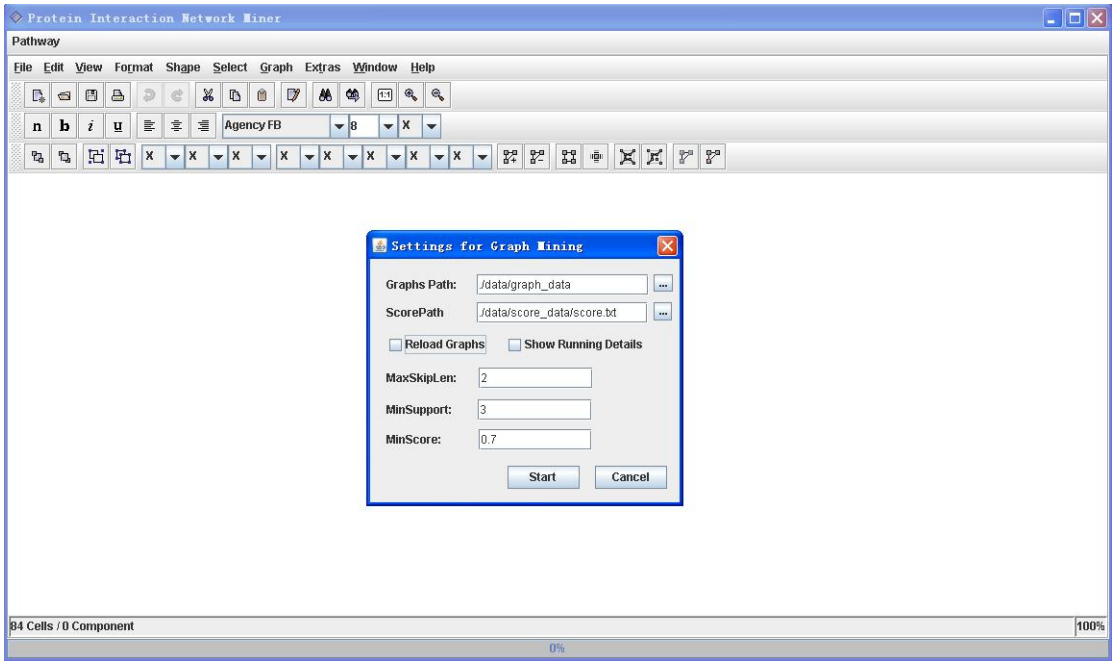
6.3 算法在Pathway系统上的实际应用效果

这一节里我们演示一下我们的算法在 Pathway 系统上实际运行效果的示例。

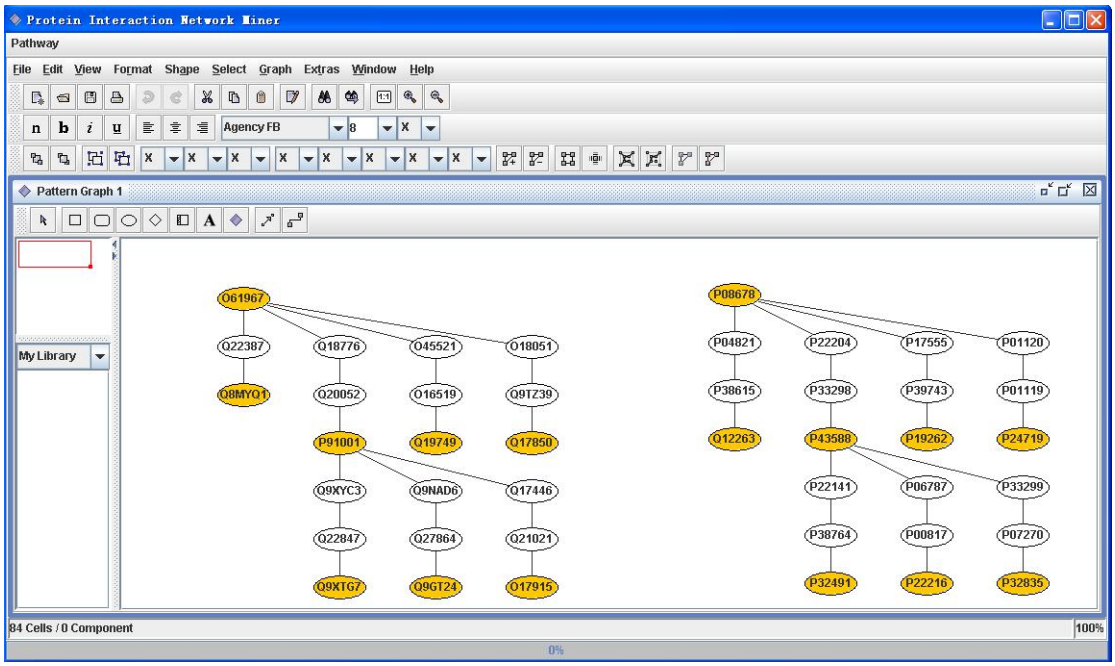
（1） 数据预处理：



（2）参数设置与模式挖掘：



（3） 结果模式的图形化表示：



第七章 总结与展望

本文对在大规模 PPI 网络图集上挖掘频繁拓扑结构的问题进行了初步的探索和研究。基于对一般标记图集上频繁拓扑结构挖掘问题的有关研究结果给出了两阶段挖掘的算法框架，同时结合 PPI 网络挖掘问题自身的特点对算法的一些重要部分进行了改进。另外，通过进行实验，对算法的性能进行了充分的测试和评估，发现了 PPI 网络挖掘中许多有价值的规律，为进一步有针对性地改进算法提供了依据。最后给出了算法在 Pathway 系统上的实际应用，证明了算法的实用价值。

但是，我们的算法无论在时间还是空间效率上都仍然有许多可以改进的地方。在时间效率上，我们可以考虑将模式扩展的过程与规范标记结合起来，即通过对规范标记的枚举来进行模式的增长，这可以有效地减少重复模式的产生。这方面已经有一些前人的工作可以借鉴。（如[8, 16]）另外，我们前面已经发现，制约我们的算法挖掘出更多有意义的模式的最大瓶颈在于巨大的频繁模式及其出现的数量。因此如何解决存储空间过大的问题显得更为迫切。这方面我们可以考虑一些更为高效的压缩存储方式，比如对路径，图进行编码等。另外就图结构本身而言往往存在某种对称性（即所谓的图的自同构现象，自同构的定义已经在第二章中介绍过），如果能够有效地利用这种对称性，那么由于相互对称的部分（即处于同一个自同构映射分区中的部分，这里同样要涉及有限置换群的知识，故不再作深入介绍）只需要存储一次，存储的空间代价肯定会大大减少。在有效地利用自同构现象来提高算法效率方面，第三章提到的 Nauty 算法已经为我们作出了典范。但是，真实 PPI 网络的对称程度究竟如何，这是我们利用自同构现象之前首先需要解决的问题。

另外，我们发现，在模式挖掘的结果中，绝大部分都是规模较小的模式，这些模式对于我们的实际应用而言并没有太大的意义，并且根据 Apriori 性质，大部分的小模式都会被包含在规模更大的模式中。因此，我们可以通过改变我们挖掘任务定义的方式来减少空间代价，从而发现更多有意义的模式。例如，我们可以只寻找那些不存在具有相同支持度的父模式的频繁模式（称为闭合频繁模式（Closed Frequent Pattern）），这方面的工作在频繁子图挖掘中已经被研究过[18]，但是在频繁拓扑结构的挖掘中似乎还处于空白阶段。

以上所提的这些问题都将成为我们未来工作的研究方向。

参考文献

[1]A. Barab´asi and R. Albert. *Emergence of scaling in random networks*. Science, 286:509–512, 1999.

[2]http://dip.doe-mbi.ucla.edu/

[3]http://www.ncbi.nlm.nih.gov/blast/

[4]M.R.Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[5]R.Agrawal and R.Srikant. *Fast algorithm for mining association rules*. In VLDB'94, 487-499, 1994.

[6]M.Kuramochi and G.Karypis. *Frequent subgraph discovery*. In ICDM'01, 313-320, 2001.

[7]A.Inokuchi, T.Washio and H.Motoda. *An apriori-based algorithm for mining frequent substructures from graph data*. In PKDD'00, 13-23, 2000.

[8]X.Yan and J.Han. *gSpan: Graph-based substructure pattern mining*. In ICDM'02, 721-724, 2002.

[9]J.Huan, W.Wang and J.Prins, *Efficient mining of frequent subgraphs in the presence of isomorphism*. In ICDM'03, 549-552,2003.

[10]S.Nijssen and J.N.Kok. *A quick start in frequent structure mining can make a difference*. In KDD'04, 647-652, 2004.

[11]R.Jin, C.Wang, D.Polshakov, S.Parthasarathy and G.Agrawal. Discovering frequent topological structures from graph datasets. In KDD'05, 606-611, 2005.

[12]S.Fortin. *The graph isomorphism problem*. Technical Report TR 96-20, Dept. of Computing Science, University of Alberta, Canada, 1996.

[13]J.R.Ullmann. *An algorithm for subgraph isomorphism*. Journal of the ACM, 23(1):31-42, 1976.

[14]B.D.McKay. *Practical graph isomorphism*. Congressus Numerantium, 30:45-87, 1981.

[15] http://cs.anu.edu.au/~bdm/nauty/

[16]H.Wielandt. *Finite Permutation Groups*. Academic Press, 1964.

[17]U.Ruckert and S.Kramer. *Frequent free tree discovery in graph data*. In ACM SAC'04, 564-570, 2004.

[18]X.Yan and J.Han. *CloseGraph: Mining closed frequent graph pattern*. In KDD'03, 286-295, 2003.