班　级 031111

学　号 03111002

西安电子科技大学

本科毕业设计论文

题　　目　大规模图数据库的相似性搜索

学　　院　计算机学院

专　　业　计算机科学与技术

学生姓名　贾新禹

导师姓名　霍红卫

目　录

第一章　引言

1.1　研究背景与意义

随着科学技术的进一步发展，我们正逐步从信息时代走入数据时代[11]，全球的数据量正在以一种前所未有的方式增长着。数据的迅速增长，在给人们带来便捷信息的同时，也带来了一个巨大的挑战——面对日益复杂的数据，传统的查询方法不再有效，无法快速检索出相关联数据。面对大量有意义的数据，无奈于查询手段的限制，只能将其简化再进行处理。现在的大数据现状就好似守着一座金山，却不知如何开采。为了进一步挖掘有效信息，加速查询速度，提高信息价值，各种数据查询技术便应运而生。

其中最为热门的就是图数据库的查询。图作为计算机科学中的一个数据结构，其数据表达能力较强，可以很好得表示

第二章　背景知识

2.1　图基本定义

定义 2.1 (标号图[6])**.** 一个可以被四元组 $G = (V, E, \Sigma, \lambda)$ 表示的图称为**标号图** ***(labeled graph)***，其中 V 为有限的节点集合，E 为有限的边集合 $\in V \times V$，Σ 是标号集合，λ 是一个标号函数用于给各个节点与边分配标号 $\lambda : V \cup E \to \Sigma$。

如图2.1就是一个包含六个节点的标号图。需要注意的是标号与标识的区别，标号是图的固有属性，标识只是为了方便使用人为添加的记号。在图2.1中，v_i 就是标识，而 A, B, C, D 则是标号。

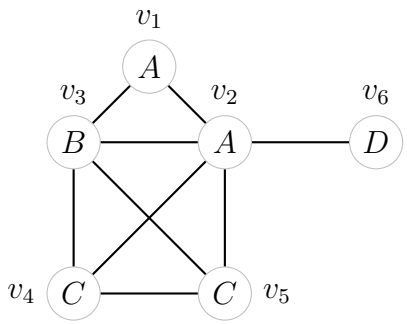


图 2.1 标号图

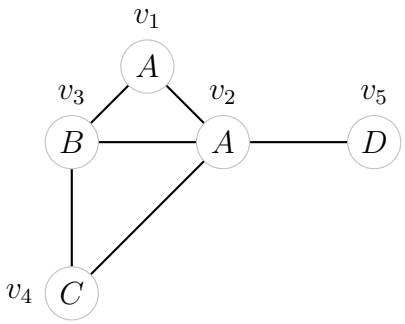


图 2.2 图2.1的子图

定义 2.2 (子图)**.** 如果一幅图 $G = (V, E, \Sigma, \lambda)$ 和另一幅图 $G' = (V', E', \Sigma', \lambda')$ 有1-1 映射的关系 $f : V \to V'$，那么图 G 就是 G' 的**子图** ***(subgraph)***，用 $G \in G'$ 表示。f 可以有这么几种

- 对于所有 $v \in V, \lambda(v) = \lambda'(f(v))$
- 对于所有 $(u, v) \in E, (f(u), f(v)) \in E$
- 对于所有 $(u, v) \in E, \lambda(u, v) = \lambda'(f(u), f(v))$

换言之，如果一幅图和另一幅图的节点标签，边关系，边标签能一一对应上，那么这副图就是另一幅图的子图。如图2.2就是图一个2.1的子图，节点标签，边关系及边标签均可一一对应，只有标识可以不同。

定义 2.3 (超图)**.** 如果图 G 是图 G' 的子图，则 G' 是 G 的**超图**。对于图2.1和图2.2,图2.2是图2.1的子图，所以图2.1是图2.2的超图。

定义 2.4 (顶点的度[10])**.** 一个顶点 u 的**度数**是与它相关联的边的数目，记做 $degree(u)$。$degree(v_i) = |E(v_i)|, v_i \in V(i = 1, 2, ..., n)$, 图2.1中 v_2 的度 $degree(v_2) = 5$，图2.2中 v_2 的度为 $degree(v_2) = 4$。

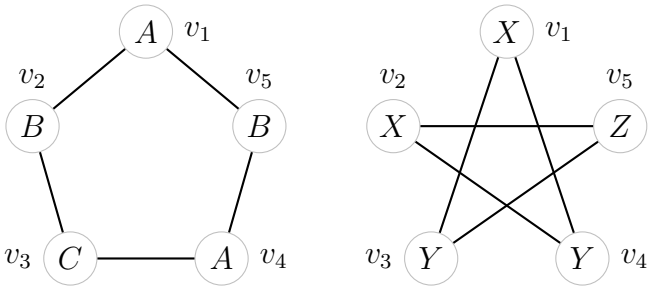
定义 2.5 (图的尺寸[12])**.** 一般由图中节点数 $|V(G)|$ 定义。因此图2.1的**尺寸** ***(size)*** 为 6，图2.2为 5。

定义 2.6 (路径)**.** 在图 $G(V, E)$ 中，若从顶点 v_i 出发，沿着一些边经过一些顶点 $v_{p1}, v_{p2}, ..., v_{pm}$，到达顶点 v_j, 则称顶点序列 $(v_i, v_{p1}, v_{p2}, ..., v_{pm}, v_j)$ 为从顶点 v_i 到 v_j 的一条**路径** ***(path)***。如在图2.1中，(v_1, v_2, v_5) 就是一条 v_1 到 v_5 的路径。

定义 2.7 (图的同构)**.** 给定图 g 和图 g', 若 g' 满足 $g' \equiv_{iso} g$, 则称 g 与 g' 是**同构图**。$g' \equiv_{iso} g$ 同构，当且仅当存在一个双射函数 $f : V(g) \leftrightarrow V(g')$，其满足下列条件:

- 对于所有 $v \in V, \lambda(v) = \lambda'(f(v))$
- 对于所有 $u, v \in V, (u, v) \in E \Leftrightarrow (f(u), f(v)) \in E$
- 对于所有 $(u, v) \in E, \lambda(u, v) = \lambda'(f(u), f(v))$

如图2.3, 图 G_1 和图 G_2 就是一组同构图, 顶点标号 $A \leftrightarrow X, B \leftrightarrow Y, C \leftrightarrow Z$,$G_1$ 中的边与 G_2 的边也形成双射关系



(a) G_1 (b) G_2

图 2.3 图的同构

2.2　图存储表示方式

图存储表示方法有很多种，常用的有 2 种：**邻接矩阵** *(Adjacency Matrix)* 和**邻接表** *(Adjacency List)*。本节还将介绍一种新颖的图表示方法**简化包表示** *(Reduced Bag Representation)*[6]。

2.2.1　邻接矩阵

在邻接矩阵存储方法中，除了一个记录各个顶点信息的顶点数组外，还有一个表示各个顶点之间关系的矩阵，称为**邻接矩阵**。设 $G(V,E)$ 是一个具有 n 个顶点的图，则图的邻接矩阵是一个 $n \times n$ 的二维数组，用 $Edge[n][n]$ 表示，定义为:

$$Edge[i][j] = \begin{cases} 1, if(i,j) \in E \\ 0, else \end{cases} \tag{2-1}$$

增加邻接图

2.2.2　邻接表

邻接表就是将同一个顶点发出的边连接在同一个称为边链表的单链表中。边链表的每个节点代表一条边，称为边节点。每个边节点有两个域：该边终点的序号以及下一个边节点的指针。

增加邻接

2.2.3　包表示法

包表示法是王教授等人在 *G-Hash*[6] 中提出的一种方法。这种方法的主要思想是将图中的各个节点特征提取出来形成一个列表，然后将这些特征就作为每个节点的标识。这样整幅图就变成了一个字符串的包，也就是一组字符串。通常，我们用这个节点的标号和其邻接节点的不同标号数目为特征。例2.1所示就是一个常用的包表示方法

例 2.1. 表2.1是图2.1的包表示。我们以节点标号和其邻接节点各个标号的个数作为特征，因此有五个特征：本身标号，A,B,C,D 分别的个数。所以加上第一列的标识，表一共有六列。而一共有六个不同节点，所以有六行。我们以 v_3 为例，详细说明下抽取特征的步骤。首先 v_3 的标号是 B, 所以 Label 为 B, 然后和 v_3 邻接的共

Nodes	Label	#A	#B	#C	#D
v_1	A	1	1	0	0
v_2	A	1	1	2	1
v_3	B	2	0	2	0
v_4	C	1	1	1	0
v_5	C	1	1	1	0
v_6	D	1	0	0	0

表 2.1 图2.1的包表示

有四个节点，分别是 v_1, v_2, v_4, v_5，其标号分别为 A, A, C, C 所以 $\#A = 2, \#C = 2$ 其余为 0。节点 v_3 的包表示就是“B,2,0,2,0”, 如表2.1所示。需要注意一点的是，包表示不仅仅这一种表示方法，对于选取什么样的特征并没有限制。不过特征必须是离散的，这样才可以用字符串表示。

2.3 图查询类型

根据图之间的包含关系，可将图查询分为子图查询和超图查询。根据图查询的精确程度，可以将其分为精确查询和相似查询。

定义 2.8 (子图查询[5])**.** 子图查询的问题为: 对于给定的一个查询图，返回图数据库中所有查询图的超图。给定一个图数据库 GD 和一个查询图 q，子图查询的目的是找出在 GD 中所有包含 q 或者 q 的同构的图的集合，最后返回该超图集合 Q。$Q = g | g \in GD \wedge q \subseteq g$。

定义 2.9 (超图查询[2])**.** 超图查询的问题为: 对于给定的一个查询图，返回图数据库中所有查询图的子图。给定一个图数据库 GD 和一个查询图 q，超图查询的目的是找出在 GD 中所有以 q 为超图的图的集合，最后返回该子图集合 Q。$Q = g | g \in GD \wedge q \supseteq g$。

图2.4就是同一个查询 q 在同一数据库中子图查询和超图查询的不同结果。

定义 2.10 (精确查询)**.** 图精确查询的问题是这样定义的: 对于给定数据库 $G = g_1, g_2, ..., g_n$, 查询图 q, 返回 G 中与 q 具有子图同构的图集合。具体可以分为子图查询和超图查询。

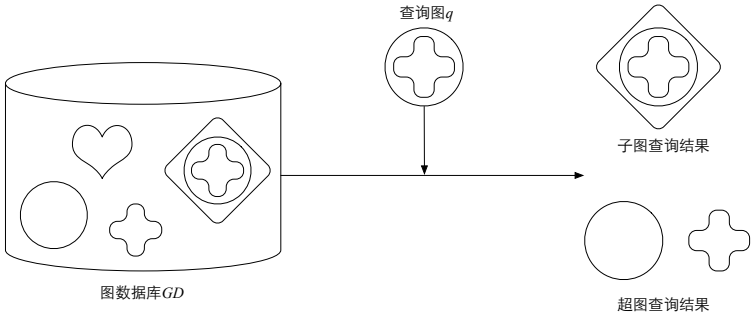


图 2.4 子图查询与超图查询

定义 2.11 (近似查询)**.** 图近似查询，又称相似性搜索，是这样定义的：对于给定数据库 $G = g_1, g_2, ..., g_n$, 查询图 q, 返回 G 中与 q 距离小于预设阀值的图集合。

因此，近似查询中相似性度量方法和近似阀值决定了结果集的大小。阀值越大，候选集越多；阀值越小，候选集越少。当阀值为零是，其结果应该与精确查询一致。

常见的相似性度量方法有两种，一种方法是*编辑距离 (edit distance)*. *编辑距离*就是我们将图 G 通过一系列操作 (如增删点边，重新标号等) 变换为另一个图 G' 所需的操作数。我们可以通过给不同操作分配不同的权值，然后计算权值和作为距离，这样可以让距离更为符合我们实际需求。虽然编辑距离是一种非常直观的图相似性测度方法，但是我们很难计算 (实际上，这是个 NP-hard 问题)。还有一种方法是*最大公共子图 (maximal common subgraph)*[8]，这里就不详细说明了。

第三章　经典图查询算法

本章将详细介绍几种经典的图查询算法。由于基本的图查询模式均为“过滤-验证”，所以要提高查询效率，只能从两方面优化。一是过滤阶段优化，二是验证阶段优化。众所周知，图是结构画的数据，目前并没有一个统一高效的索引机制可以实现较好的查询结果。因此大多数学者都会在过滤阶段选用不同的索引方式来得到更好的查询结果。而在验证阶段，由于图同构是个 NP-hard 问题，所以验证会消耗大量时间。如何快速检测同构，或将同构转为其他非复杂多项式的一半问题也是学者们关心的一个热点课题。本章介绍的几种查询方法均为过滤阶段的优化。

3.1　图精确查询算法

本节将详细介绍子图查询中的 *GraphGrep* 算法[1] 和 *gIndex* 算法[9]。

3.1.1　GraphGrep 算法

2002 年 ShaSha 教授等人提出了 *GraphGrep* 算法[1] 。GraphGrep 算法是基于特征索引方法中的第一个经典算法，它采取典型的“过滤 -验证”框架，GraphGrep 算法中只讨论过滤阶段。由于基于结构的算法中对图的顺序扫描代价太高，GraphGrep 提出基于路径的过滤方法，来减少候选集大小。

其算法的主要流程是这样的:

1. 构造索引: 首先将节点和边利用哈希存成两个二维表作为未来筛选用特征之一，然后枚举图数据库中所有长度不大于 l_p 的路径 $v_0, v_1, v_2, ..., v_k, (k \leq l_p, \forall i \in [1, k-1], (v_i, v_i+1) \in E)$，然后将这些路径与图的关系存为一个二维表作为索引。表的每一行代表每一个图，每一列为一个路径，利用简单哈希确定路径对于行号，每一个单元格代表在该图包含该路径几条，如表3.1所示。

2. 解析查询: 如同图数据库，首先将节点和边利用哈希存成两个二维表，然后枚举查询中的所有长度不大于 l_p 的路径，哈希存在一个列表中。

Key	g_1	g_2	g_3
h(CA)	1	0	1
...			
h(ABAB)	2	2	0

表 3.1 GraphGrep 索引

3. 数据库过滤:

   (a) 利用节点信息过滤: 如果查询图中的某一节点在数据库中一图里未出现，则此图一定不会包含查询图，因此可以删去。

   (b) 利用边信息过滤: 同节点过滤，如果某条边未出现，则一定不会包含查询图，可删去。

   (c) 利用路径个数进行过滤: 如果查询图中某一路径个数大于数据库中一图的此路径个数，则此图一定不会包含查询图，可以从候选集中删去。

4. 子图查询: 利用标号进行路径合成，从候选集中删去所有未能成功合成的候选图。剩下的就是 GraphGrep 算法返回的候选集。后续再加以图同构验证即可得到最终子图查询结果。

成过程

由于 GraphGrep 算法是基于路径的，编写十分简单，但是路径数目过多，造成索引集很大，建立十分费时，子图查询过程中也有大量运算量，因此效果有限。

3.1.2 gIndex 算法

2004 年，Yan 教授等人提出了 *gIndex* 算法[9]。gIndex 算法是以频繁子图结构作为索引特征的子图查询算法。它采用*动态支持度 (size-increasing support)* 和*区分度片段*两种方法来优化算法，获得性能更好的索引。

Index 详

其算法主要流程如此:

1.

3.2　图相似性搜索

本节将着重介绍两种相似性搜索方法，分别是 *G-Hash* 算法[6] 和 *C-tree* 算法[3]。

3.2.1　G-Hash 算法

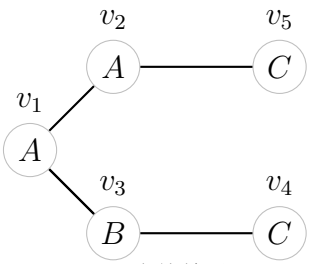
G-Hash 算法是 Wang 教授等人于 2009 年提出的一种图相似性搜索方法。相比较其他近似搜索方法，这种方法更加稳定高效。G-Hash 开创性地采用了*简化包表示 (Reduced Bag Represent)* 来表示每个节点特征，并表示成字符串，Hash 存储作为索引。并利用*小波匹配核函数 (Wavelet Graph matching kernels)* 来计算节点间的相似度，从而得到和查询图最为相似的 K 个图。但是 G-Hash 算法的准确度和速度完全取决于相似度度量函数，因此一个好的相似性度量方法尤为重要。

算法主要流程如下:

1. 索引构建: 将图数据库中每幅图用简化包表示，然后将每个节点特征变为字符串，利用 Hash 存成索引。如例3.1所示, 就是一个图建成索引过程。需要注意的是在例子中，我们只有三个标号，所以特征矩阵只有四列。但是在实际中当我们选取节点标号数目作为特征时，需要先统计出图数据库中有的所有标号，这样才好做归一化存储。

   例 3.1. 图3.1就是一个 G-Hash 构建索引的实例。图3.1(a) 是需要存储的一幅图。我们选取节点标签和邻接节点各个标号的个数作为度量特征，所以一共有四个特征。首先统计各个节点周围各个标号的数目，结果如图3.1(b) 所示。然后用小波函数提取出实际特征值。举例而言，对于 v_3 在用 $h = 0$ 的小波函数提取后，局部特征是 $[B, 2, 0, 1]$。将这些表示特征值的字符串利用 Hash 找到对应位置，然后将对应节点标号填到此位置，最后生成的哈希表如图3.1(c) 所示。而整个数据库生成的哈希表就如图3.1(d) 所示。

2. 查询过程: 将查询图也同数据库中图一样用简化包表示。然后计算其中每个节点字符串和图数据库中每个字符串的相似性，乘以每个图中此字符串出现的次数，得到这个节点和每幅图的相似度，求和即可得到总的相似度。排序得到最相近的 K 个。



(a) 一个简单图 P

Nodes	Label	#A	#B	#C
v_1	A	1	1	0
v_2	A	1	1	1
v_3	B	2	0	1
v_4	C	0	1	0
v_5	C	1	0	0

(b) 节点特征矩阵

索引键	哈希值
A,1,1,0	v_1
A,1,1,1	v_2
B,2,0,1	v_3
C,0,1,0	v_4
C,1,0,0	v_5

(c) 图 P 对应的哈希表

索引键	G_1	G_2	G_3
A,1,1,0	1	4	0
A,1,1,1	0	1	0
B,2,0,1	2	0	6
C,0,1,0	0	5	1
C,1,0,0	3	0	1

(d) 图数据库哈希表示例

图 3.1 一幅简单示例图

图是 G—Hash 查询的一个完整例子。从 XXXX

添加 G-Hash+ 中那副图

此方法还支持动态增删图数据，如例3.1因为其用字符串作为索引键，所以如果增删的图不改变原有的特征情况，在例3.1中就是标号的个数，那么在添加时需要添加数据库中的一列，在删除时也只需要删除一列即可。不需重建整个索引。

3.2.2 Closure tree 算法

C-tree

第四章　基于二次哈希开链法的图精确查询

由于传统算法在利用哈希表存储索引中多采用简单哈希，容易产生冲突问题，导致构建索引效率较低。本章在路径索引的基础上，探究了不同哈希方法对算法速度的影响，并提出一种基于二次哈希开链法的精确查询算法，来减少图查询中的过滤阶段的耗时，以提高查询速度。本章将先介绍下现有的哈希算法，然后详细介绍基于路径的查询方法包括作为此算法验证方法用的子图同构算法 *ULLMANN*[4]。最后是实验结果与分析。

4.1　常用哈希方法

本节将介绍几种常用的哈希方法及字符串哈希函数。

补充

4.1.1　开放定址法

4.1.2　开链法

4.1.3　再哈希法

4.1.4　常用字符串哈希函数

4.2　基于路径的查询算法

本方法同 *GraphGrep* 算法[1]，都是基于路径的精确子图查询算法。算法基本流程如下：(1) 遍历图数据库中图的路径，(2) 利用双哈希构建索引，(3) 遍历查询路径，(4) 利用基本索引特征做先验剪枝，(5) 路径合成进一步筛选候选集，(6) 子图同构确定最终结果。下面我们将分小节详细说明这些步骤。

4.2.1　数据库路径遍历

首先，对于每个数据库我们设定一个路径长度的上限 l_p，l_p 越大意味着可记录的路径越长，索引集合也会相应增加。随后对于数据库中的每幅图，我们用深度优先搜索遍历每个节点，遍历最大深度为 l_p，并记录遍历过程中经过的每一条路径，存成一个列表，用于下一步构建索引。

4.2.2 二次哈希开链法索引构建

双哈希是再哈希的一种，其利用两个哈希函数来构造哈希探测序列，大大降低了地址冲突概率。但是传统上，双哈希方法实质上也是开放定址法的一种，也就是如果所需节点数目大于 Mod 时，将完全无法表示。这完全不符合图数据库实际情况，因为作为一个通用算法，我们无法预先确定数据库中最大图的节点数。因此，本文提出了一个变形的双哈希算法，即**二次哈希开链法**来进行哈希定址。

二次哈希开链法就是只进行一次双哈希，然后再有冲突就利用开链法解决。双哈希函数公式如4-1。

$$h(key) = (h_1(key) + h_2(key))\%Mod \tag{4-1}$$

其中 h_1, h_2 为两个不同的哈希函数，Mod 为哈希函数取模值，一般为一个小于但最接近存储空间大小的素数。当 $h_1(key)$ 发生冲突时，再用 $h_2(key)$ 的值作为偏移量来进行探测。如果再有冲突则进行开链法，存成链表。

根据二次哈希开链法的特点，本文设计了一种将路径字符串映射到哈希表的算法，如算法4.1所示。而访问时函数则不需重新设计，直接用开链法原有函数即可。

算法 4.1 二次哈希编码

输入： 路径字符串 $path\_string$

输出： 哈希编码 $code$

```
1: function HASH(path_string)
2:     code ← h1(key)%Mod
3:     if code 有冲突 then
4:         code ← (code + h2(key))%Mod
5:     end if
6:     return code
7: end function
```

在二次哈希开链法中哈希函数可以自选，不过我们推荐选取算法4.2中的两个函数作为 h_1, h_2，经过实验这两个函数对于字符串哈希这两个效果最好。

BKDRHash 运算简单，速度快，所以作为第一次哈希函数。*APHash* 不易冲突，所以作为第二次。

算法 4.2 哈希函数

输入: 字符串 $String$

输出: 哈希编码 $code$

function $h_1(String)$ ▷ BKDRHash
    $char \leftarrow (string.first)$
    $code \leftarrow 0$
    while $char \neq 0$ **do**
        $code \leftarrow code << 6 + char$
        $char \leftarrow (char.next)$
    end while
    return $(code \& 0 \times 7FFFFFFF)\%Mod$
end function
function $h_2(String)$ ▷ APHash
    $char \leftarrow (string.first)$
    $code \leftarrow 0$
    $i \leftarrow 0$
    while $char \neq 0$ **do**
        if i 为偶数 **then**
            $code \leftarrow (code \oplus ((code << 7) \oplus char \oplus (code >> 3))$
        else
            $code \leftarrow (code \oplus (\sim ((code << 11) \oplus char \oplus (code >> 5)))$
        end if
        $char \leftarrow (char.next)$
    end while
    return $(code \& 0 \times 7FFFFFFF)\%Mod$
end function

通过哈希存储，我们可以很便捷得获得各图包含的路径关系表，我们将其存到文件中作为索引，分离查询与建库，进行离线查询加速查询速度。

4.2.3 查询图路径遍历

我们对查询图也像数据库中的图一样进行拆分，以 l_p 为路径最大长度遍历出所有路径。然后同样存成一个哈希表，记录着每一条路径出现了几次。为进一步查询做准备。不过和数据库路径遍历有所不同的是，查询图的遍历过程中需要记录不同路径中相同的节点，这个可以通过在路径中添加特定标签实现。

4.2.4 先验剪枝

在介绍本算法的先验剪枝步骤之前，我们先介绍一下**包含逻辑规则** ***(inclusion logic)***。

定义 4.1 (包含逻辑[7])**.** 对于给定的标号图 g_1, g_2，和 g_1 的一个子图 g'，若 g_1 是 g_2 的子图，则 g' 必定是 g_2 的子图 $(g_1 \subseteq g_2) \Rightarrow (g' \subseteq g_2)$。反之，若 g' 不是 g_2 的子图，则 g_1 也不可能是 g_2 的子图 $(g' \not\subset g_2) \Rightarrow (g_1 \not\subset g_2)$

从包含逻辑规则中，我们可以得知如果查询图 g_1 中的子图 g' 不是数据图 g_2 的子图，那么 g_2 就不可能是 g_1 的超图，因此可以放心得把 g_2 从候选集中删去。这就大大加速了“过滤 -验证”框架中过滤阶段的过滤速度。

在本算法中，我们从三个方面进行了先验剪枝来缩小索引集个数。三个方面分别是 (i) 节点，(ii) 边，(iii) 路径。如果查询图中的某个节点个数大于数据图中的，那么数据图自然不会包含查询图。如果查询图中有数据图中没有的边，那么自然这幅数据图也不合要求。如果查询图中某条路径的个数大于数据图，那么因为包含逻辑规则，这幅数据图也当从候选集中删去。

经过这三步筛选过程，候选集将会大大减少，降低了后文所述路径合成和子图同构所需时间。

4.2.5 路径合成

当完成先验剪枝后，候选集已经相对较小，但是并没有达到最好的情况。我们可以通过路径合成确定其中很多的合理性。路径合成的复杂度远远小于子图同

构，因此最后对候选集做一次路径合成可以大大降低最终复杂度。路径合成，顾名思义就是将多条路径进行合成，具体而言就是将多个有着公共节点的路径合成到一起，通过判定其是不是相同节点来筛选候选集。我们采用的是遍历的方法，只进行两两合成，然后逐一比对。如例4.1所示。

例 4.1. 假设有路径 $\bar{A}B\underline{C}\bar{A}$ 和 $\underline{C}B$，其中有相同标记的节点均为同一节点，即在本例中两个 A 是同一节点和两个 C 也是同一节点。

1. 现在假设数据库中有四幅图，其路径信息如下所示，数字代表节点标识:

$$g_1 : ABCA = (1,0,3,1)\ CB = (3,2)$$

$$g_2 : ABCA = (1,2,3,1)\ CB = (3,2)$$

$$g_3 : ABCA = (1,0,3,4)\ CB = (3,2)$$

$$g_4 : ABCA = (1,0,4,1)\ CB = (3,2)$$

2. 在路径合成前，我们就可以利用节点信息删去 g_3，因为 g_3 的 $ABCA$ 中两个 A 一个是 1，一个是 4，并非同一节点。

3. 我们将 ABCA 和 CB 合成, 我们知道其中两个 A 是统一节点，两个 C 是同一节点，而两个 B 不是。我们得到合成结果:

$$g_1 : ABCACB = (1,0,3,1),(3,2)$$

$$g_2 : ABCACB = (1,2,3,1),(3,2)$$

$$g_4 : ABCACB = (1,0,4,1),(3,2)$$

4. 显然，g_2 不满足条件，因为它的两个 B 也是一个节点，g_4 也不满足，因为其两个 C 不是同一节点。所以筛选集中只剩下 g_1。

可见，通过路径合成，我们大大缩小了候选集，降低了下一步子图同构所需的计算量，加速了整个算法。

4.2.6　子图同构

子图同构作为算法最后的验证部分，承担着对于整个算法正确性把关的责任，也同样是个需要消耗大量运算量的部分。由于子图同构是个 $NP-hard$ 问题，所

图 4.1 Ullmann 算法流程图

以目前仍没有一种快速的解法。我们选用经典算法 *ULLMANN* 算法[4] 来解决这个问题。下面我们将大致介绍下 UllMANN 算法。

ULLMANN 算法是 Ullmann 教授 1976 年提出的一种经典图同构算法，其本质是基于一个深度优先搜索树。其算法流程如图4.1所示。首先，根据查询图节点的出入度从数据图中找出候选集。随后，再根据每个节点的邻接节点对候选集进行筛选。最后，通过深度优先查询，一一遍历配对，寻找匹配点。例4.2就是一个子图同构的完整过程。

Ullmann
子

例 4.2.

4.3 实验结果与分析

4.3.1 实验环境

本文提出算法的实验环境为 CPU Intel Core i7, 主频为 1.7 GHz，内存为 8 GB 1600 MHz DDR3，硬盘为 128GB SSD，操作系统为 Mac OS X Yosemite 10.10.3；所有算法均用 C 语言在 clang 600.0.56 环境编译完成。

4.3.2 实验数据分析

实验数据全采用真实数据，为 DTP 提供的 AIDS 数据集。可从以下网址得到:`https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data`。我们从其数据库中随机抽取了 1000,2000,4000,16000 个图作为我们的查询集合。

验

第五章　基于字符串距离的图相似性搜索

参考文献

[1] Rosalba Guigno and Dennis Shasha. Graphgrep : A fast and universal method for querying graphs. 2002.

[2] Shang H, Zhu K, and Lin X. Similarity search on supergraph containment. *ICDE*, 2010.

[3] H. He and A. K. Singh. Closure-tree: an index structure for graph queries. *Proc. International Conference on Data Engineering'06 (ICDE)*, 2006.

[4] J.R.ULLMANN. An algorithm for subgraph isomorphism. *Journal Association for Commputing Machinery*, 23(1):31–42, January 1976.

[5] Kurmochi M and Karypis G. Frequent subgraph discovery. *ICDM*, 2001.

[6] Xiaohong Wang, Aaron Smalter, and Jun Huan. G-hash:towards fast kernel-based similarity search in large graph databases. *ACM*, 2009.

[7] Yan X, Yu P S, and Han J. Graph-based substructure pattern mining. *ICDE 2002*, 2002.

[8] T. Jiang Y. Cao and T. Girke. A maximum common substructure-based algorithm for searching and predicting drug-like compounds. *Bioinformatics, 24(13)*, 2008.

[9] Xifeng Yan, Philip S.Yu, and Jiawei Han. Graph indexing : A frequent structure-based approach. *ACM*, 2004.

[10] 王桂平，王衍，任嘉辰. *图论算法理论, 实现及应用*. 北京大学出版社, 2011.

[11] (英) 维克托 · 迈尔 -舍恩伯格，肯尼思 · 库克耶. *大数据时代: 生活、工作与思维的大变革*. 浙江人民出版社, 2012.

[12] 谭伟，杨书新. 图数据精确查询与近似查询的研究. 2013.