第一章　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，并提出一种基于双哈希的精确查询算法，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*[?]。最后是实验结果与分析。

1.1　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

1.1.1　链表法

1.1.2　双哈希法

1.1.3　常用字符串哈希函数

1.2　基于路径的查询算法

本方法同 *GraphGrep* 算法[?]，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。

1.2.1　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。

1.2.2 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1-1。

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

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1.1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 1.1 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1.2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 1.2 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。

1.2.3 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。

1.2.4 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 1.1 (包含逻辑[?])**.** 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 $(g' \not\subset g_2) \Rightarrow (g_1 \not\subset g_2)$

1.2.5 路径合成

1.2.6 子图同构

1.2.6.1 UllMANN 算法

1.3 实验结果与分析

1.3.1 实验环境

1.3.2 实验数据分析