HyperANF:
Approximating the Neighbourhood Function of Very Large Graphs on a Budget

基本思路

参考论文：*ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs*

$\mathcal{M}$(x, h)：点集合V中到点x距离<=h的点集合

$\mathcal{M}$(x, 0) = {x} for all x ∈ V

FOR each distance h DO

  $\mathcal{M}$(x, h) = $\mathcal{M}$(x, h − 1) for all x ∈ V

  FOR each edge (x, y) DO

    $\mathcal{M}$(x,h) = $\mathcal{M}$(x, h) ∪ $\mathcal{M}$(y, h − 1)

$$B(x,r) = \bigcup_{x \to y} B(y, r-1)$$

基本基数计算方法

基数

- 集合中不同元素的个数

常用计算基数方法

- B树
  - 查找，添加删除效率高，不好合并，空间复杂度高
- Bitmap
  - 查找效率高，易合并，空间复杂度缩减1/32

基本估算方法

一个简单的基数估算方法

假设

1,随机生成n个0-1的服从均匀分布的数字

2,随便重复其中一些数字，重复的数字和重复次数都不确定

3,打乱这些数字的顺序，得到一个数据集

估算方法

max/min

优点

算法简单直观

缺点

数据量小的时候准确率低

数据要求均匀分布

估算方法LC(Linear Counting)

参考论文：*A linear-time probabilistic counting algorithm for database applications*

LC的基本思路是：设有一哈希函数H，其哈希结果空间有m个值（最小值0，最大值m-1），并且哈希结果服从均匀分布。使用一个长度为m的bitmap，每个bit为一个桶，均初始化为0，设一个集合的基数为n，此集合所有元素通过H哈希到bitmap中，如果某一个元素被哈希到第k个比特并且第k个比特为0，则将其置为1。当集合所有元素哈希完成后，设bitmap中还有u个bit为0。则：

$$\hat{n} = -mlog\frac{u}{m}$$

为n的一个估计，且为最大似然估计（MLE）。

如果m比n小太多，则很有可能所有桶都被哈希到了，此时u的值为0，LC的估计公式就不起作用了（变成无穷大）。因此m的选择除了要满足上面误差控制的需求外，还要保证满桶的概率非常小。

$$m > (e^t - t - 1)/(\varepsilon t^2)\text{，其中}t = {}^{n}/_{m}$$

精度要求越高，则bitmap的长度越大。随着m和n的增大，m大约为n的十分之一
优点：算法简单，易于合并
缺点：依然有很高的空间复杂度

估算方法LLC(LogLog Counting)

参考论文：*Log log Counting of Large Cardinalities*

基本思想：

设a为待估集合（哈希后）中的一个元素，由上面对H的定义可知，a可以看做一个长度固定的比特串（也就是a的二进制表示），设H哈希后的结果长度为L比特，我们将这L个比特位从右到左分别编号为1、2、…、L。
设$\rho(a)$为a的比特串中第一个“1”出现的位置，显然$1 \le \rho(a) \le L$，取$\rho_{\max}$为所有$\rho(a)$的最大值，则：$\hat{n} = 2^{\rho_{max}}$

满桶控制：

上述分析给出了LLC的基本思想，不过如果直接使用上面的单一估计量进行基数估计会由于偶然性而存在较大误差。因此，LLC采用了分桶平均的思想来消减误差。具体来说，就是将哈希空间平均分成m份，每份称之为一个桶（bucket）。对于每一个元素，其哈希值的前k比特作为桶编号，其中$2^k = m$，桶编号相同的元素分配到同一个桶中，分别计算m个桶的最大的第一个“1”的位置，然后求平均，则：$\hat{n} = 2^{\frac{1}{m}\sum M[i]}$

举例：

假设H的哈希长度为16bit，分桶数m定为32。设一个元素哈希值的比特串为“0001001010001010”，由于m为32，因此前5个bit为桶编号，所以这个元素应该归入“00010”即2号桶（桶编号从0开始，最大编号为m-1），而剩下部分是“01010001010”且显然$\rho(01010001010) = 2$，索引桶编号为“00010”的元素最大的ρ即为$M[2]$的值

误差：$m > (\frac{1.30}{\varepsilon})^2$

估算方法HLLC(HyperLogLog Counting)

参考论文：*HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*

基本思想：

HLLC的一个改进是使用调和平均数替代几何平均数。注意LLC是对各个桶取算数平均数，而算数平均数最终被应用到2的指数上，所以总体来看LLC取得是几何平均数。由于几何平均数对于离群值（例如这里的0）特别敏感，因此当存在离群值时，LLC的偏差就会很大，这也从另一个角度解释了为什么n不太大时LLC的效果不太好。这是因为n较小时，可能存在较多空桶，而这些特殊的离群值强烈干扰了几何平均数的稳定性。

因此，HLLC使用调和平均数来代替几何平均数，调和平均数的定义如下：

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} x_i}$$

调和平均数可以有效抵抗值的扰动，使用H后，估计公式变为：

$$\hat{n} = \frac{\alpha_m m^2}{\sum 2^{-m}}$$

其中：

$$\alpha_m = (m \int_0^{\infty} \left(\log_2 \left(\frac{2+u}{1+u} \right) \right)^m du)^{-1}$$

误差：$m > (\frac{1.04}{\varepsilon})^2$

估算方法HLLC(HyperLogLog Counting)

Algorithm 1 The Hyperloglog counter as described in [5]: it allows one to count (approximately) the number of distinct elements in a stream. α_m is a constant whose value depends on m and is provided in [5]. Some technical details have been simplified.

```
0   h : 𝒟 → 2^∞, a hash function from the domain of items
1   M[–] the counter, an array of m = 2^b registers
2       (indexed from 0) and set to −∞
3
4   function add(M: counter, x: item)
5   begin
6       i ← h_b(x);
7       M[i] ← max{M[i], ρ⁺(h^b(x))}
8   end; // function add
9
10  function size(M: counter)
11  begin
12      Z ← (∑_{j=0}^{m−1} 2^{−M[j]})^{−1};
13      return E = α_m m² Z
14  end; // function size
15
16  foreach item x seen in the stream begin
17      add(M,x)
18  end;
19  print size(M)
```

Algorithm 2 The basic HyperANF algorithm in pseudocode. The algorithm uses, for each node $i \in n$, an initially empty HyperLogLog counter c_i. The function union$(-,-)$ maximises two counters register by register.

```
0   c[–], an array of n HyperLogLog counters
1
2   function union(M: counter, N: counter)
3       foreach i < m begin
4           M[i] ← max(M[i], N[i])
5       end
6   end; // function union
7
8   foreach v ∈ n begin
9       add v to c[v]
10  end;
11  t ← 0;
12  do begin
13      s ← ∑_v size(c[v]);
14      Print s (the neighbourhood function N_G(t))
15      foreach v ∈ n begin
16          m ← c[v];
17          foreach v → w begin
18              m ← union(c[w], m)
19          end;
20          write ⟨v, m⟩ to disk
21      end;
22      Read the pairs ⟨v, m⟩ and update the array c[–]
23      t ← t + 1
24  until no counter changes its value.
```