WEB 图的表示与压缩

摘要

Abstract

目录

第一章、绪论

1.1、引言

互联网上包含了海量的网页，而网页和一般文本的重要区别是在页面内容中包含相互引用的链接(Link)，如果将一个网页抽象成一个节点，而将网页之间的链接理解为一条有向边，则可以把整个互联网抽象为包含页面节点和节点之间联系的有向图，称之为 WEB 图[2]。

WEB 图是对互联网的一种宏观抽象，其微观构成元素是一个单独的网页。对于某个单独的页面 A 来说，在其内容部分往往会包含指向其他网页的链接，这些链接一般称为页面 A 的出链(Out Link)。因为页面 A 是在一个网状结构中，所以不仅网页 A 会指向其它页面，也会有很多其它页面指向网页 A，那么这些指向网页 A 的链接称为网页 A 的入链(In Link)。所有网页的出链，入链和网页本身就构成了一个大的 WEB 图[2]。

对 WEB 图的分析在很多方面都有应用[4]，例如网页的排名，过滤垃圾网站，检测站点之间的连通性，检测镜像站点等等。尤其在搜索领域[3]，当搜索引擎需要计算一个网页的排名时，一个很重要的评价因素就是网页的入链个数，一个网页的入链个数越多，那么这个网页就越重要。但是要计算一个网页的出链个数很容易，计算它的入链个数就相对难得多，因为不知道哪些页面指向了该页面。要

计算某个页面的入链个数，这就需要知道其它节点是否包含了指向这个页面的链接，WEB 图的结构和性质很好的满足了这个要求。

在 2011 年 9 月初，谷歌搜索引擎索引到的 WEB 网页已经达到 440 亿个，假设平均每个 WEB 网页存在 20 个链接，表示每条链接需要五个字节，链接信息以邻接表的方式存储，那么存储搜索引擎索引到的这整个 WEB 图将需要 4.4TB 的存储空间，远远超出了当前所有计算机内存的大小。为了在有限的内存中存储包含更多节点的 WEB 图，就需要对 WEB 图进行压缩表示。

1.2、背景知识

在现代信息社会里，计算机科学和技术的进步使得各种形态的数字化信息的数量和规模正以极快的速度在增长。截止目前，WEB 上数据正以每天三百万个页面的速度持续增长，页面总数已经超过 500 亿，并且将继续随着时间按指数进行增长。

数据挖掘是从海量的数据中自动，高效地提取有用知识的一种新兴的数据处理技术，包括分类，聚类，关联规则挖掘，特征与偏差，时序模式发现，趋势分析等。近年来，因特网的飞速发展与广泛应用，使得 WEB 上的信息量以惊人的速度增长，为数据挖掘提供了丰富的数据源和研究课题。WEB 挖掘就是利用数据挖掘技术，从与互联网相关的资源和用户浏览行为中抽取感兴趣的、有用的模式和隐含的信息。对 WEB 图的挖掘研究实际上是 WEB 挖掘中的一支，目前对 WEB 图的挖掘要面对的一个重要问题是 WEB 图越来越大，而计算机内存的增长速度缓慢，因此无法在内存中存储足够多的节点以直接进行 WEB 图的有关计算，因此需要寻找 WEB 图的压缩算法，使得能够在有限的内存中加载更多的节点。

1.2.1、WEB 图中的幂分布

通过之前的研究人员对 WEB 图的研究，总结出了 WEB 图的以下性质。

(1)、WEB 图中每个节点包含的平均链接个数为 7.2[1]；

(2)、节点的出度服从 b=2.75 的幂分布，即节点的出度为 i 的概率与$i^{-2.7}$成正比[1]；

(3)、节点的入度服从 b=2.15 的幂分布，即节点的入度为 i 的概率与$i^{-2.1}$成正比[1]；

(4)、WEB 图中的强连通分量服从 b=2.54 的幂分布[1]；

(5)、在一个站点用的 WEB 页面个数的数量服从 b=2.2 的幂分布[1]；

(6)、一天之内访问某个站点的用户数量服从 b=2.07 的幂分布[1]；

(7)、某个用户每天点击的链接数服从 b=1.5 的幂分布[1]；

(8)、网页的 PageRank 排名值服从 b=2.1 的幂分布[1]。

其中 WEB 图中节点的出度和入度的所服从的幂分布如下图所示：

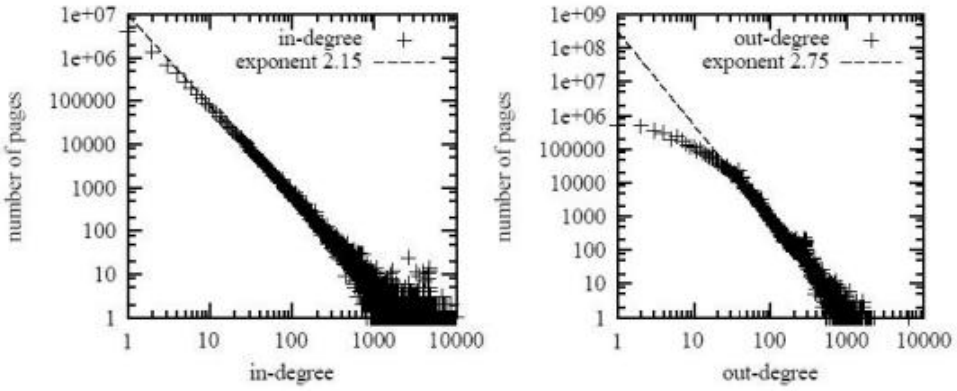

图 1.1、Web 图中节点的出度和入度的分布

1.2.2、WEB 图的领结模型

2000 年，Andrei Zary Broder 根据对 WEB 图的分析给出了 WEB 图的结构中一个很著名的模型，由于他给出的模型外形很像一个领结，因此称之为 WEB 图的领结模型[1]。

在该模型中，他将 WEB 图分为六部分，分别是 SCC，IN，OUT，Tendrils，Tubes 和 Disconnected。其中 SCC 是整个 WEB 图中的最大强连通分量，可以认为是整个 WEB 图的核，它由存在于 WEB 图中的许多相互链接的页面组成，占整个 WEB 图的 27.5%；第二部分是 IN 分量，它是由那些包含有到核心的链接但由核心并不能链接到的页面组成，占整个 WEB 图的 21.5%；第三部分是 OUT 分量，它是由那些从核心能够链接到但并不包含到核心的链接的页面组成，占整个 WEB 图的 21.5%；第四部分称之为 Tendrils，它是由那些能由上述三部分到达或者能够到达上述三部分，但是却不属于任何一个部分的页面组成；第五部分是 Tubes，它是由那些由 IN 中的链接指向，但是却又包含到 OUT 中的链接的页面组成，它与 Tendrils 一起占整个 WEB 图的 21.5%；第六部分 Disconnected 由既不由上述五个部分的链接指向，也不存在指向上述五个部分的链接的页面组成，占整个 WEB

图的 8%。

1.3、WEB 图压缩的相关概念

WEB 图并不同于普通的图，它们都具有某些特殊的性质，也正因为具有了这些性质，才使得可以对 WEB 图可以进行有效的压缩，主要特性列举如下：

(1)、邻接表的相似性[6]。新增的页面的链接集合往往和某个已存在的页面的链接集合很相似，这样就造成了在一个 WEB 站点或者某个域内部，存在很多链接集合比较相似的页面。

(2)、域和 URL 的局部性[6]。在一个域或者主机上的页面大部分页面的链接都指向在同一个域上的其他页面，接近 3/4。

(3)、链接的连续性[6]。在一个页面中的所有链接的 URL 在字典顺序中是比较接近的，即它们的 URL 都存在一定程度的相似。

WEB 图的压缩包括 URL 的压缩和 WEB 图中链接关系的压缩[5]，在本论文中主要讨论的是链接关系的压缩，压缩的目的主要是为了减少 WEB 图中每条边占用的位(bit)长度，使得能够在有限的内存空间中容纳更多的节点或者更大节点数量的 WEB 图，这样基于 WEB 图的某些运算就能在比较短的时间内完成。既然有 WEB 图的压缩，那么肯定也有 WEB 图的解压缩，不过更加具有实用性的却是在压缩 WEB 图之后进行的随机访问节点信息的操作。评价一个 WEB 图压缩方案好坏的标准主要有两个，一个是压缩之后每条边占用的位长度，另一个就是节点的随机访问时间。

1.4、本文主要工作

本文主要工作是提出一种改进的 WEB 图压缩算法，使得能够在维持比较良好的压缩效果的前提下，能够压缩更大数量节点的 WEB 图，同时针对基于压缩后的 WEB 图的操作，提出一套解压缩与查询节点信息的算法，使得能够在比较短的时间之内，查询出或者解压缩出压缩后 WEB 图中某个节点的详细信息，并真实的 WEB 图数据进行测试，对提出的算法进行验证与分析。

第一章介绍了 WEB 图的概念和 WEB 图的压缩在搜索领域的定位和作用，同时分析了使得 WEB 图能够被压缩的特殊性质，以及 WEB 图压缩相关的某些概念。

第二章分析了 WEB 图压缩的现状，简要介绍了主流的几种压缩 WEB 图的方法，以及它们的优点和不足，并且与我的方法进行了简单的比较。

第三章详细介绍了如何对 WEB 图进行压缩表示，包括使用到的其它算法以及进行某些设计的原理。

第四章详细介绍了，在对 WEB 图进行压缩之后，如何对压缩后的 WEB 图进行解压缩和查询的方法，并且针对解压缩和查询节点信息的过程中存在的问题，给出了加速解压缩和查询过程的方案。

第五章主要是实验结果的验证与分析，包括对 WEB 图的压缩和对压缩之后 WEB 图进行解压缩与查询的实验测试结果以及针对实验结果给出了某些方面的分析。

第六章主要是对论文的总结，分析了该压缩方案中存在的优点和目前存在的不足，并且对该方案的不足之处提出了某些改进的思路。

第二章、WEB 图压缩的研究现状

2.1、几种主要的 WEB 图压缩算法介绍

2.2.1、哈夫曼编码

在一个 WEB 图中，某些节点经常被作为其它节点的链接节点，那么在存储这些节点时，如果给这些节点分配较短的编码，那么显然是可以减少存储 WEB 图所需要的空间的。因此可以根据所有节点的入度进行哈夫曼编码，则入度高的节点将获得比较短的编码，入度低的节点获得较长的编码，然后以邻接表的方式存储整个 WEB 图，则可以对 WEB 图实现简单的压缩。这样单纯的使用哈夫曼编码的压缩效果可以达到每条边占用 14-15 位的长度[2]。

2.2.2、Connectivity Server

Connectivity Server[5]是一个以 WEB 图作为其模型的特殊数据库，主要用来进行基于 WEB 图的某些计算与分析，例如网站的连通性，网页的 PageRank 值，查

询结果的改善以及镜像站点的检测等等与 WEB 页面相关的操作。Connectivity Server 以 Link Database 作为其数据库，Links Database 可以为 Connectivity Server 提供对链接数据的快速访问。Link Database 总共包含三个不同的版本，版本一主要以连接表的形式存储 WEB 图，且并没有利用 WEB 图具有的相似性和局部性，它的每条链接占 32 位，在当时每个节点的平均链接个数为 17，因此以这种方式，表示每个节点的信息，平均需要 68 个字节。在版本二和版本三中，由于使用了很多现存的很多著名的压缩技术，例如哈夫曼编码，字典方法等，并且利用了 WEB 图的相似性和局部性，因此对 WEB 图的压缩获得了比较好的效果。版本三能够将原来的每条链接占用的 32 位减少到 6 位以下，并且仍然提供了很快的解压缩速度。

Connectivity Server 主要由三个部分组成，URL Database，Host Database 和 Link Database。其中 URL Database 主要存储节点的 URL，指纹（根据节点的 URL 产生的一个 64 位的哈希值）和节点的 URL-id（一个 32 位的整数）。URL-id 是一系列从 1 到 N 的连续整数，在给节点指定 URL-id 之前需要根据节点的出度将节点分配到三个集合，第一个结合中所有节点的出度都大于 254，第二个集合中所有节点的出度都在 24 到 254 之间，第三个集合中的节点的出度都小于 24，然后给每个集合中的节点根据它们的 URL 按照字母表进行排序。给节点分配 URL-id 的原则是先按照集合分配，然后再按照集合中的 URL 顺序进行分配，集合 1 中节点出度最高，因此分配 1 到 R 给集合 1 中的节点，集合 2 中节点的出度居中，因此分配 R+1 到 S 给集合 2 中的节点，然后给集合三中的节点分配 S+1 到 T 的 URL-id。这样就给每个节点都指定了一个 URL-id，那么 WEB 图中每个节点的关系就可以用这些 URL-id 来进行表示。Host Database 主要用于存储 URL 中的主机地址那一部分，主要用于 URL 的压缩。

Link Database 主要用于将节点的 URL-id 同该节点出链的 URL-id 和入链的 URL-id 进行映射。版本一中，只是对节点的链接关系进行了简单的映射，并没有使用压缩技术，因此，研究意义不大；版本二中，对邻接表中的节点使用了 Delta 编码(Delta code)，半字节编码(nybble code)等等技术，并且对起始数组进行了压缩，使得每条边的压缩效果达到了 8.5 位，相对版本一的 32 位，可见压缩效果是很明显的；版本三通过使用链内压缩(Interlist compression)技术，更加减少了重复链接造成的冗余，使得表示每条边的长度降到了 6 位以下。

2.2.3、WebGraph Framework

在 WEB 图的压缩方案中，目前为止最有名的就是这一种了，它既能保证良好的压缩效果，也保证了比较短的随机访问时间。这个方案使用了很多 Link Database 的第三个版本中的压缩技术，例如最著名的链内压缩(Interlist compression)，并且还使用了很多现代的压缩技术，使得对 WEB 图的压缩之后，每条边仅仅占用 3.08 位[5,6]。

WEBGraph Framework 主要由以下几部分组成：

(1)、一套扁平的编码方法方式，称为ζ编码，包括元编码(Unary code)，伽马编码（Gamma code），Delta 编码和 Golomo 编码等。这套编码方式非常适合于对 WEB 图进行编码和存储，并且整合了在一定指数范围内的幂分布结构，使得可以很好的吻合 WEB 图中的各种性质。

(2)、一套压缩 WEB 图的算法，可以用来压缩间隔和引用链，间隔编码和ζ编码能够对节点进行高压缩率的表示。这个算法的结果可以通过一系列参数进行控制，以便在压缩效果和随机访问时间之间根据需求进行定制。

(3)、一套支持在实际上不解压缩 WEB 图的情况下，随机访问 WEB 图中节点的算法，并且通过使用懒惰技术(Lazy techniques)来推迟对 WEB 图的解压缩到必须需要进行解压缩的时候才进行真正的解压缩。

(4)、一个基于 WebGraph Framework 方案，遵守 GPL 协议，使用 java 和 C++语言的完整的，文档化的实现，并且提供了清晰简单的接口。

(5)、一系列包含非常大的 WEB 图的数据集，其中的数据集都是由 UbiCrawler 网络爬虫按照一定的算法从互联网上抓取的真实数据，如果 WEB 图是按照 UbiCrawler 的爬虫算法进行抓取并且进行某些预处理，则通过 WebGraph Framework 进行压缩一定能保证比较好的效果。

2.2、我的方案与现存方案的比较

现存的很对 WEB 图的压缩算法，虽然都能够获得比较好的压缩效果，但是却都存在某些限制，而且仍然还有可以继续改善压缩效果的空间。

我的压缩方案主要有以下几点改进：

(1)、使用滑动窗口技术寻找合适的参考节点，改善算法的性能；

(2)、使用参考编码挖掘 WEB 图的相似性；

(3)、使用块压缩技术对 bit 串进行压缩，提高压缩效果；

(4)、使用间隔压缩技术挖掘 WEB 图的连续性；

(5)、使用差分压缩技术，挖掘 WEB 图的局部性；

(6)、使用类 Golomb 编码将大数据转化为小数据，使得数据分布的更加紧密，减少表示每个节点所需要的平均长度；

(7)、使用γ编码表示小的数据，提高压缩效果；

(8)、使用 Run-Length&Golomb 编码压缩差异部分，改善压缩效果。

(9)、使用哈夫曼编码对某些可能大的数据进行编码，减少他们占用的空间，提高压缩效果。

我在论文中给出了一套 WEB 图的压缩方案，该方案能够对很大的 WEB 图进行有效的压缩，前提是 WEB 图只需要满足基本性质即可，目前能达到的压缩效果为每条边仅仅占用 3.1 位左右，对于某些图甚至能够达到 1.24 位。在给出 WEB 图的压缩方案的同时也给出了 WEB 图的解压缩以及随机访问节点的方案，并且根据其中存在的问题，给出了解压缩加速算法，使得随机查询时间可以保持在一个可以接受的范围内。

第三章、WEB 图的压缩

3.1、WEB 图的压缩方案简介

在论文的 1.2 节已经介绍过 WEB 图不同于普通的图的几个特殊性质，其中与 WEB 图的压缩相关的两个重要性质就是邻接表的相似性以及域和 URL 的局部性。为了叙述方便，在论文以后的部分都会直接称呼为相似性和局部性。为了进行 WEB 图的压缩，假设所有节点的 URL 已经按照字母表顺序排好序，并且给每个 URL 分配了一个 ID，这个 ID 既可以表示这个 URL，也可以表示这个节点，以后节点间的所有关系都用节点 ID 进行表示。由于 WEB 图的相似性，因此在一个 WEB 图中存在很多不同节点的链接指向同一个节点，使用一般的方法储存 WEB 图时，就会将每两个节点之间相同的链接重复存储一次，这样就会导致在存储后

的 WEB 图文件中存在大量冗余链接，由于 WEB 图的节点个数一般很大，这样就会额外浪费很多空间。WEB 图的压缩的一个主要目的就是要减少压缩后 WEB 图中这些冗余链接的存在。

根据 WEB 图具有的相似性，可以使用一种称为参考编码(Reference Code)[2]的方式为每个节点编码，以此来减少编码过程中的冗余。为了对节点进行参考编码，需要为每个节点选择一个最合适的参考节点，使得参考编码的得到的结果占用的空间最少。为了为每个节点选择一个合适的参考节点，同时也为了保证算法的性能，引入了滑动窗口技术[6]，通过滑动窗口技术，可以在该节点之前的 W 个节点之中为该节点选择一个最合适的节点，作为它的参考节点。

在计算出节点的参考节点之后，就可以对每个节点进行参考编码，通过参考编码将节点的表示分割为两个部分，一部分是相对于参考节点的一个位向量，另一部分是当前节点的链接集合与参考节点的链接集合的一个差集，简称为差异部分。然后，通过对位向量进行块压缩，对差异部分进行间隔压缩，差分压缩等等技术来对 WEB 图中的节点以及整个 WEB 图进行压缩表示。

3.2、参考编码

由于 WEB 图中相似性的存在，可以使用一种参考编码(Reference Code)[2]的方式为每个节点编码，以此来减少编码过程中相同链接造成的冗余。简单的描述就是寻找两个比较相似的节点 A 和 B，以 A 为参考节点，将 B 的链接集合拆分为两部分，一部分是节点 A 与节点 B 同时包含的链接，数学表示$\mathrm{similar(B)} = \mathrm{links(A)} \cap \mathrm{links(B)}$，另一部分就是存在于节点 B 的链接集合但是却不存在于节点 A 的链接集合中的链接，数学表示就是$\mathrm{difference(B)} = \mathrm{links(B)} - \mathrm{links(A)}$。其中节点 A 与节点 B 相同的部分，可以参照节点 A 中链接的顺序，将$\mathrm{similiar(B)}$转化为一个位向量 vector(B)，vector(B)的位长度等于参考节点 A 的出度 outdegree(A)，位向量中为 1 的位表示节点 B 也包含节点 A 中这个位置的某个链接，为 0 的位则表示不包含此位置的链接。这样就把节点 B 中相对于节点 A 的冗余链接减少到了一个 1 个 bit 位。如下例所示，假设：

links(A)={1,3,4, 5,7,8,9,11,12}，

links(B)={1,4,6,7,9,10,11,12,13,15}；

那么：

reference(B)=A，

similar(B)={1,4,7,9,11,12}，

difference(B)={6,10,13,15}，

vector(B)=101010111。

通过节点 reference(B)的详细链接信息、vector(B)和 difference(B)就可以准确的表示节点 B。

在给节点 B 进行参考编码之后，参考编码的结果将由三部分组成，reference(B)，vector(B)，和 difference(B)，在节点的链接已经按照要求排好序的情况下，参考编码的算法如下所示，其时间复杂度为$O(M+N)$，其中 M 和 N 分别为参考节点和被编码节点的链接个数。

```
ReferenceCode (node, reference)
1   j ← 1
2   for i ← 1 to node.length
3         do
4         while j < reference.length && node[i] > reference[j]
5               do  vector.clr(j), j ← j + 1
6         if j = reference.length
7         then
8               for k ← i to node.length
9               do  defference.add[node[k]]
10              break
11        if  node[i] = reference[j]
12        then
13              vector.set(j), j ← j + 1
14        else
15              difference.add(node[i])
16  for  k ← j to reference.length
17        vector.clr(k)
18  return vector difference
```

3.3、通过滑动窗口选择参考节点

为了给每个节点进行参考编码，需要为每个节点选择一个最合适的参考节点，这样可以使得压缩后的 WEB 图具有更好的压缩效果。为每个节点寻找最合适的参考节点，需要把 WEB 图转换为关联图(Affinity Graph)[2]，关联图也是一个有向图，它的节点是 WEB 图中的节点加上一个虚拟根节点 root，有向边的起点为参

考节点，终点为被编码的节点，有向边的权值表示以参考节点编码某个节点的开销，这样就将寻找最合适参考节点的过程转换为在关联图中寻找一棵最小树形图(The Directed Minimum Spanning Tree 或者 Optimum Tree)[2]的问题了。虽然在关联图中计算出的最小树形图中包含了每个节点的最合适的参考节点，但是这个方案存在一定的局限性，因为 WEB 图的节点个数巨大，在将 WEB 图转换为关联图的过程中，时间复杂度为$\theta(N^2)$，花费的时间将很长，而且在关联图中计算最小树形图的过程也需要花费很长的时间和很大的内存空间，因此需要对寻找参考节点的方案进行改进。为了解决这个问题，引入了滑动窗口技术。滑动窗口技术的原理是，假设某个节点的最合适参考节点在该节点之前的 W 个节点中，那么寻找参考节点的时候就只需要在前 W 个节点进行选择，这样就将时间复杂度降低到了$\theta(WN)$，而且通过实际测试，即便这个参考节点可能不是最优的，但是它带来的整体压缩效果与最好的参考节点带来的压缩效果相差不大，而运行时的时间以及占用的内存空间却减少了许多。由于寻找的参考节点受到窗口大小 W 的限制，因此理论上来说，W 越大，寻找的参考节点就越合适。

为了在前 W 个节点中寻找最合适的节点作为参考节点，那么这就需要一个规则来进行准确的判断，判断规则可以使用下一节介绍 cost 函数[2]，cost(A,B)表示使用节点 B 作为参考节点编码节点 A 的开销，如果节点 B 的信息已知，那么也表示对节点 A 使用参考编码表示时，编码节点 A 所需要的 bit 长度。在进行判断前需要为 WEB 图引入一个虚拟节点 R，也称为根节点，为了方便该节点同时也可以使用 root 进行表示。假设当前节点为 A，那么$\mathrm{cost(A,R)}$表示不使用参考编码编码节点 A 所需要的开销，即直接对节点 A 进行表示所需要的 bit 长度。假设滑动窗口内的节点为$W = \{w_1, w_2 \dots\dots, w_k\}$，那么需要在$W \cup \{R\}$内选择一个节点 B 使得$\mathrm{cost(A,B)}$最小。如果$B \epsilon W$，那么节点 A 的参考节点就是 B，否则，参考节点为 R。整个算法的伪代码如下：

```
FindReference(graph, win)
1   for i ← 2 to graph.nodecount
2   do
3       result[i].target = i
4       w = min(win, i)
5       limit = cost(graph.node[i], root)
6       for j ← 1 to w
7       do
8           weight[j] = cost(graph.node[i], graph.node[i − w + j])
```

```
9       min = weight[1]
10      index = 1
11      for k ← 2 to w
12      do
13          if weight[k] < min
14          then
15              min = weight[k]
16              index = k
17      if min ≥ limit
18      then
19          result[i].source = 1
20      else
21          result[i].source = i − w + index
22  return result
```

3.4、cost 函数

在使用滑动窗口寻找参考节点的过程中，需要使用到一个很有用的函数 cost(i, j)，如果节点 j 表示被编码的节点，节点 i 为参考节点，那么cost(i, j)可以按照如下方式计算[2]：

$$\mathrm{cost}(i,j) = \mathrm{outdegreee}(j) + \lceil \log n \rceil * (|N(i) - N(j)| + 1)$$

公式 3.1、cost 函数

现解释为什么如此设计 cost 函数。对某个节点 i 使用参考编码后，结果将由三部分组成，一部分是长度等于参考节点 j 出度的位向量 vector(i)，另一部分是 difference(i)，还有一部分是节点的参考节点 reference(i)。假设 WEB 图存在 n 个节点，若要表示每个节点，则每个节点的平均二进制编码长度至少为⌈log n⌉位。公式中的outdegree(j)就是参考编码中位向量的长度，$\lceil \log_n \rceil * (|N(i) - N(j)| + 1)$就是 difference(i)+reference(i)的长度。因此有向边的权值既可以表示以参考节点编码某个节点的开销，也可以表示在给定参考节点的情况下，进行参考编码之后，表示当前被编码节点所需要的 bit 串的长度。虽然 cost 函数计算出来的值与实际进行参考编码后所得到的的 bit 串的长度存在一定的差别，但是通过实际测试，它仍然是一个很优秀的判定规则。

当节点 j 为根节点 root 时，outdegreee(j)为 0，$\lceil \log_n \rceil * (|N(i) - N(j)| + 1)$为以 root 作为参考节点编码节点 i 后节点 i 的位长度，同时也可以表示不使用参考编码，而是直接表示节点 i 所需要的位长度，加 1 是为了保存 root 参考节点，方便以后进行运算。由于节点 i 为 root 时没有意义，因此不进行讨论。

通过研究滑动窗口技术，可以发现找出的所有参考节点与被编码节点之间的关系组成了一个树，树的根节点是虚拟节点 R，这棵树相对于通过关联图计算出来的最小树形图，在参考节点与被参考节点的关系上来说是一棵次优的树。因为对节点进行参考编码也可以表示在指定了参考节点之后，完整表示被编码节点的所需要的 bit 串的长度，而关联图的最小树形图的所有边的总权值是最小的[2]，因此如果按照最小树形图进行参考编码，则完整表示所有节点的信息所需要的总位长度也是最少的，因此按照最小树形图的结构进行参考编码可以保证整个 WEB 图的压缩效果达到最好。由于通过滑动窗口计算出来的树在某些节点关系上相对于最小树形图是一颗次优的树，因此它带来的压缩效果也将会是很优秀的，在实际的测试过程中也证明了这一点。

3.5、块压缩技术

一个 bit 串，假设为 10011101110，使用 Run-Length 编码[22]可以表示为 $\{(1,1),(2,0),(3,1),(1,0),(3,1),(1,0)\}$，由于连续的 1 和连续的 0 都是间隔出现的，因此该 bit 串可以表示为{1,1,2,3,1,3,1}，块的个数为 6，最前面的一个 1 表示该 bit 串是以 1 开头的，如果为 0 则表示该串是以 0 开头的，可以用 block_flag 表示。如果 bit 串以 1 开头，那么接下来的那一个块一定都是 0，由于通过 bit 串的长度、块的个数和除去最后一个块后其余块的信息已经可以将原来的 bit 串还原，因此在保存时可以不用保存最后一个块的信息，这样可以节省存储空间。例如 bit 串 10011101110 的最终表示{1,1,2,3,1,3}，块的个数为 6，bit 串的长度为 11。

由于 WEB 的局部性和连续性，在使用参考编码之后的得到的 vector 向量中一般会存在比较长的连续的 1 或者连续的 0，这样就很适合使用块压缩技术来对该 vector 进行压缩。而且 vector 的长度即为参考节点的出度，这样就不再需要额外的空间来存储 vector 的长度了。

3.6、间隔压缩

根据 WEB 图的局部性和连续性，在使用参考编码之后得到的差异部分中，也存在部分部分连续的链接。假设当前节点的 ID 为 17，进行参考编码之后，得到的差异部分为D = {5,12,13,14,20,51,52,56,80,66538}，那么{12,13,14}，{51,52}

就是连续其中连续的链接，可以对这些连续的链接使用间隔压缩，来减少表示WEB 图的节点所需要的空间[6]。通过间隔压缩，又可以将差异部分分为两个部分，一部分是连续出现的链接集合，称为连续链接集合；一部分是没有连续出现的链接，可以称为剩余链接。对于连续链接集合，可以用两个参数表示，连续出现的第一个链接和每个间隔链接的长度，可以用 (first,length)进行表示。例如{12,13,14}可以表示为(first,length)=(12,3)，为了利用 WEB 图的局部性和支持差分压缩带来的好处，需要使用一些技术将(first,length)中的值进行某些转化。

对 D 进行间隔压缩后得到两部分内容{{12,13,14},{51,52}}和{5,20,56,80,66538}，对{{12,13,14},{51,52}使用(first,length)表示为{(12,3),(51,2)}，根据 WEB 图的局部性和连续性，一般第一个 first 值与节点 ID，相邻的两个 first 值都相差不会很大，这样就可以通过节点 ID、节点 ID 与第一个 first 值的差值和每两个连续的 first 值的差值来表示每个(first,length)对。算法的伪代码如下：

```
IntervlasEncoding(intervlas, nodeid)
1   for i ← intervlas.length downto 2
2   do
3       intervals[i].first −= intervlas[i − 1].first − intervlas[i].length − 1
4       intervlas[i].length −= 1
5   start_flag = (intervlas[1].first ≥ nodeid)? true: false
6   intervals[1].first = abs(intervals[i].first − nodeid)-1
7   intervlas[1].length −= 1
8   return start_flag
```

通过对每个(first,length)对进行转化，可以使用较小的数字来表示每个(first,length)，这也是差分压缩的一种应用。转换后每个 first 都大于或者等于 0，每个 length 都大于 0，由于每个 length 都不会很大，那么每个 length 就很适合使用γ编码进行表示。最终，{{12,13,14}，{51,52}}转化后结果为{false,{4,2},{35,1}}，其中 false 表示 intervals[1].first 小于 nodeid，在还原 intervals[1].first 是会用到这个标志。

3.6、差分压缩

3.6.1、差分编码

差分压缩是通过对有序序列进行差分编码来实现的。差分编码(differential

encoding)，又称增量编码或者间隔编码，相对于存储或者传输完整信息，差分编码存储或者传输序列化资料之间的差异部分，并且因此而得名。在差分编码中，差异储存在称为“delta”或“diff”的离散文件中。由于序列之间相邻的元素改变通常很小（平均占全部大小的 2%），差分编码能大幅减少信息的重复和冗余。一连串独特的差分信息在空间上要比未编码的信息具有更好的空间效率。

差分编码的简单例子是储存序列化资料之间的差异（而不是储存资料本身）：例如对于信息“2, 4, 6, 9,10”，不是存储信息本身，而是存储为“2, 2, 2, 3, 1”。即第一个元素不变，以后只存储当前元素相对于前一个元素的差值。差分编码单独使用用处不大，但是结合其它编码方案，在序列式数值常出现时可以帮助压缩资料。例如，对于上例，信息为 “2,4,6,9,10”，给每个节点编码，那么每个节点至少需要$\lceil \log_2 5 \rceil = 3$位，则完整表示整个序列需要$3 \times 5 = 15$位，但是如果使用差分编码，则每个节点只需要$\lceil \log_2 3 \rceil = 2$，完整表示整个序列需要$2 \times 5 = 10$位。

使用差分编码有一定的限制，要求序列内的元素都尽量按照递增或者递减的顺序，这样每两个元素之间的差值能保持在一定的范围内，更有助于进行压缩表示。因此对于非递增或者递减的序列，如果元素的位置并不会影响序列的意义，可以对序列先进行排序，然后对序列进行差分表示。

3.6.2、差分编码压缩剩余链接

在对差异部分使用了间隔压缩之后，每个节点中就只剩下剩余链接没有进行压缩了，使用差分编码可以对这些剩余链接进行压缩[6]。假设节点 A 的剩余链接用 rest_links(A)进行表示，节点的参考节点使用 reference(A)进行表示。假设节点 A 的节点 ID 为 nodeid(A)=17，reference(A)=7，rest_links(A)= {5,20,56,80,66538}，那么可以对 reference(A)和 rest_links(A)使用差分编码。对于 reference(A) ，由于 reference(A)一定小于 nodeid(A)，那么 reference(A)使用差分编码可以表示为 reference(A)=17-7=10。特殊情况，如果节点的参考节点为虚拟节点为 R，那么 reference(A)为 0，因为这样可以使用更少的 bit 来进行表示。对于 rest_links(A)，先对 rest_links(A)进行升序排列，然后将 rest_links(A)的第一个元素更新为 $s_1 = |s_1 - \text{nodeid}(A)|$，同时保存一个标记，用来指示$(s_1 - \text{nodeid}(A))$的正负，记为 rest_flag_diff。从第二个元素开始，每个元素更新为$s_i = s_i - s_{i-1}$-2。通过差

分编码，reference(A)与 rest_links(A)的信息最后可以表示为 reference(A)=10，rest_links(A)={12,13,34,22,66456}。

由 WEB 中链接的连续性和局部性，某个页面 URL 和它的链接的 URL 在字母表顺序是比较接近的，由于 WEB 图中的节点 ID 是按照 URL 的字母表顺序进行分配的，因此该页面代表的节点 ID 和它的有向边指向的节点的 ID 一般也是比较接近的。对节点的剩余链接使用差分编码，可以将参考节点和剩余链接中分布比较广泛和离散的节点 ID 转换为分布集中在较小范围内的数字，这样就可以使用更短的 bit 串来对每个数字进行表示。

3.7、类 Golomb 编码

虽然由于 WEB 图的局部性以及连续性，在进行间隔编码以及差分编码之后，每个 first 值以及节点的剩余链接中的值都比较小，但是也可能存在某些例外，例如上述节点 A，进行差分编码之后，在剩余链接中仍然包含一个很大的值 66456。虽然这种情况出现的并不频繁，但是由于 WEB 图中的节点个数巨大，积少成多，这样就会导致所有出现在 first 值以及剩余链接中的值会分布在一个比较大的范围内。根据实验可以得出，在这个范围内，排在前面的小数字出现的频率要高很多，而排在后面的大数字出现的频率却很低。由于节点分布的范围很大，那么平均表示每个节点所需要的 bit 数也就会相应的增多，而某些大数字出现的频率却又相对很低，这就需要进行某些处理，将一些大的数字转化为小的数字，来减小数字分布的范围。

Golomb 编码[22]是一种无损编码技术，Golomb 编码对较小的数用较短的编码，较大的数用较大的编码表示。假设要编码的数字为 x，Golomb 编码的步骤如下：

(1)、选定参数 m，令$b = 2^m$必须在压缩前固定下来，因为解压缩时必须确定 m 的大小；

(2)、令$q = \lfloor (x-1)/b \rfloor$；

(3)、令 $r = x - qb - 1$，也就是$x = qb + r + 1$；

(4)、这样 x 可有两部分的编码组成，第一部分是 q 个 1 加上 1 个 0 组成，表示 q；第二部分是用 m 位二进制数组成，第二部分的二进制值为 r。

为了将大数字转化为小数字，我使用了类似 Golomb 编码的思想，但是却并

没有完全的使用 Golomb 编码，因此我在这里称之为类 Golomb 编码。假设要编码的数字为 y，为了保证 WEB 图良好的压缩效果，可以使用类 Golomb 编码将大数字按照如下方式进行转化：

(1)、令$m = \lfloor (y/65536) \rfloor$，$r = y - 65536 * m$

(2)、因为需要在后来的编码过程中需要对 m 和 r 进行哈夫曼编码，而哈夫曼编码是前缀编码，是可以唯一解读出来的，因此假设 m 的哈夫曼编码为 b1，r 的编码为 b2。

(3)、对于任何数字 y，如果 m=0，那么在进行存储时，任何出现 y 的地方都用{0,b2}代替，即先存储一个 bit，并将该 bit 设置为 0,然后再在该 bit 后面存储 b2；如果对于数字 y，其$m > 0$，那么，在存储 y 时，任何出现 y 的地方，都用{1,b1,b2}代替，即先存储一个 bit，并将该 bit 设置为 1，然后再在该 bit 后面存储 b1，接着在存储 b2。这样，额外添加的那个 bit 就包含了如何解码该数字的信息。

在实际的压缩过程中，因为有一些值是可以保证不会大于 65535 的，所以对于这些值并没有使用这种表示方式，例如节点的出度以及节点的参考节点的值等，而且对于剩余链接，我又设计了一种与此相关的新的方案，因此只有间隔压缩部分的 first 字段使用了此种方式的编码及存储方案。

3.8、Run-Length&类 Golomb 编码

前文提到过，由于 WEB 图的局部性和连续性，在对剩余链接使用差分编码之后，大部分时候所得到的序列应该都是些比较小的数字，但是有时也有例外，例如某个节点指向了一个距离自己很远的链接，那么即便使用了差分编码，所得到的序列中至少仍然会存在一个很大的值，由于序列在进行计算时，节点是根据 ID 按照升序排列的，因此在进行差分编码之后，这个大的数字最有可能出现的地方就是剩余链接的前面或者后面，根据实际统计，出现在后面的次数要比出现在前面的次数多，因此就需要一种合适的技术来对这些情况进行处理。

假设剩余链接 rest_links(A)= ={12,13,34,22,66456}，那么可以使用 Run-Length 编码结合类 Golomb 编码对剩余链接进行变换，变换的思想如下：

(1)，根据 Run-Length 编码的思想对序列进行变换，将大于 65535 分为 1 组，小于等于 65535 的分为一组，同时设置一个标记，用来标记剩余链接的第一个元素

是否大于 65535，记为 rest_flag_golomb。对 rest_links(A)进行变换的结果为 rest_flag_golomb=false，rest_links_length={4,1}，意思即为前面四个元素小于等于 65535，第五个元素大于 65535。

(2)、将大于 65535 的数字通过类 Golomb 编码转换为(m,r)对，66456 将被转换为(1,921)。

(3)、将剩余序列中每个大于 65535 的数字替换为其对应的(m,r)对，这样{12,13,34,22,66456}就被转换为{12,13,34,22,1,921}。

(4)、此时，通过 rest_flag_golomb，rest_links_length 以及 rest-links 就可以对信息进行还原。但是由于大数字出现在剩余链接的尾部的次数更多，为了尽量改善压缩效果，还需要对 rest_links_length 进行某些变换。变换的伪代码如下：

```
ConvRestLinksLength(rest_links_length)
1    size = 1
2    for i ← 1 to rest_links_length.size
3    do
4        size += rest_links_length[i]
5    for j ← 1 to rest_links_length.size
6        rest_links_length[i] = size − rest_links_length[i]
7        size −= rest_links_length[i]
```

在伪代码中初始化 size 为 1 是为了方便使用γ编码对转化后的 rest_links_length 中的值进行编码，通过计算可以得出，在进行转化之后 rest_links_length 的最后一个元素一定为 1。在转化之后，rest_links_length(A)={2,1}，2 和 1 都很适合使用γ编码进行表示，对 rest_links_length 进行转化的目的是尽量减小每个 rest_links_length 中的值。

假设 rest_links(B)={2,3,7,10,12,16}，那么使用 Run-Length&类 Golomb 编码之后的结果为 rest_flag_golomb=false,rest_links_length={1},rest_links={2,3,7,10,12,16}。

3.9、Elias γ 编码

Elias γ编码[20,21]是 Elias 编码中最简单的一种编码。要对自然数$x \in N = \{1,2,3,\dots\}$编码，在该自然数的二进制码前面加上$\lfloor \log x \rfloor$个 0 。注意$\lfloor \log x \rfloor + 1$为将 x 写作二进制所需的位数。例如，对于$x = 13$，其二进制表示为 1101，$\lfloor \log 13 \rfloor = 3$，因此，13 的Elias γ 码字为 $C\_\gamma(13) = 0001101$。

简单研究一下解码过程，很容易验证这种构造是一个瞬时码(instantaneous

code)。解码器首先计算出该码字开始的 0 的数目，不妨说 n，如果 n 为 0，然后解码的数为 1 ；如果 n 非 0 ，那么解码器读出接下来的n + 1个 bit，并将这 n+1 个 bit 还原成相应的数字。

γ编码对于编码小数字能够达到比较好的效果，为了尽量保证良好的压缩效果，对 WEB 图的节点表示中确定比较小的数字，我都采用的γ编码，包括块压缩后的块长，间隔压缩后的间隔数，间隔压缩后每个间隔中的 length 字段，Run-Length&类 Golomb 压缩后的 rest_links_length 中的值。

3.10、哈夫曼编码

哈夫曼编码[19]是一种给信息进行编码的方式，是一种用于无损数据压缩的熵编码(权编码)算法。哈夫曼编码使用变长的编码表对信息进行编码，其中变长编码表是通过对信息的出现频率进行评估得到的，出现频率高的信息使用较短的编码，反之，出现频率高的信息使用较长的编码，这便使得编码之后信息序列的平均长度、期望值最低，从而达到压缩数据的目的。

由于哈夫曼编码中任一信息的编码都不是其它信息编码的前缀，因此，每个编码都能够唯一的表示一个信息，同时这也使得对编码之后的信息的解码过程是比较容易的。

在对 WEB 图进行压缩的过程中，有几个地方需要使用到哈夫曼编码，包括节点的出度，节点的参考节点，使用间隔压缩后的 start 字段，以及使用最后的剩余链接中的值。由于节点的出度以及参考节点的值一定小于 65536，那么这些部分就不用再存储额外的 bit 位来标识该数字的大小了，同理，剩余链接中也不再需要额外的 bit 位。

假设 WEB 图中节点的个数不超过2^{32}，由于使用了差分编码、类 Golomb 编码以及 Run-Length&类 Golomb 编码，这就将那些分布在很大范围内的数字转化为分布在 0-65535 之间，这样有两个好处，一是可以保证良好的压缩效果，而是减少了额外的辅助数据结构所占用的空间，在实际的测试过程中，不论 WEB 图的节点个数多大，由于哈夫曼编码所产生的辅助数据结构的大小始终不超过 700K，这就使得该压缩方案能够对很大的 WEB 图进行压缩，而仍然能够保证良好的压缩效果。

3.11、压缩后节点的表示

通过以上各小节，可以得出对 WEB 图中的节点进行压缩表示时，需要如下的数据结构来表示每个节点：

```
NodeEncoding
{
    out_degree;          //节点的出度
    reference;           //节点的参考节点
    block_count;         //块压缩后块的个数
    block_flag;          //块标记
    blocks;              //块压缩后得到的块
    intervals_count;     //间隔压缩后得到的间隔个数
    start_flag;          //开始标记
    intervals;           //间隔压缩后得到的间隔，每个间隔包括 start 和 length
    rest_flag_diff;      //差分压缩的标记
    rest_flags_golomb;   //Run-Length&类 Golomb 编码后的标记
    rest_links_length;   //剩余链接中每块的长度值
    rest_links;          //Run-Length&类 Golomb 编码后的剩余链接
};
```

其中 out_degree，reference，intervals 中的 start 字段，和 rest_links 中的数据可能比较小，也可能相对比较大，因此对这些部分都使用哈夫曼编码进行表示，由于 block_count，blocks，intervals_count，intervlas 中的 length 字段和 rest_links_length 一般都确定是一些小数字，因此这些部分使用γ编码进行表示，block_flag，start_flag，rest_flag_diff 和 rest_flag_golomb 都用一个 bit 位来进行表示。

通过 reference，block_count，block_flag 和 blocks 能够还原进行参考编码后得到的向量 vector，通过 intervals_count，start_flag 和 intervals 能够对使用了间隔编码的连续链接集合进行还原，通过 outdegree，rest_flag_diff，rest_flag_golomb，rest_links_length 和 rest_links 则能够对剩余链接信息进行还原。

3.12、存储压缩后的 WEB 图

WEB 图的压缩不仅包括压缩 WEB 的过程，也包括对压缩后的 WEB 图执行解压缩和通过压缩后的 WEB 图查询节点信息的过程，因此在设计压缩方案的过程中，我们需要考虑以后的解压缩和查询过程，尽量也为这些操作提供方便，以提高解压缩和查询的速度。

在上一节中，已经给出了表示节点信息的数据结构，节点的该数据结构能够完整的表示节点的信息，为了更好的支持 WEB 图的解压缩和节点的信息查询，选择按照如下方法存储压缩后的 WEB 图：

(1)、节点按照顺序存储，为了减少空间占用，将压缩后 WEB 图的最小单位分割到 bit。即按 bit 顺序存储节点 A 的 outdegree，reference 直到 rest_links 之后，然后从下一个 bit 开始继续存储节点 B 的 outdegree，reference 直到 rest_links 等等信息，直到该子 WEB 图中的最后一个节点为止；

(2)、在存储间隔压缩中的间隔信息时，假设 start 部分小于 65536，则先存储一个 bit，并将该 bit 设置为 0，然后存储 start 的编码，如果 start 部分大于等于 65536，则先存储一个 bit，并将该 bit 设置为为 1，然后存储进行类 Golomb 编码之后的得到的(m,r)对的编码。

(3)、在存储 rest_links 与 rest_links_length 的时候，需要按照 Run-Length 编码的方式进行存储，例如 rest_flag_golomb=false、rest_links={12,13,34,22,1,921}、rest_links_length={2,1}，那么存储时先存储一个 bit，并将该 bit 设置为 0，接下来存储 2 的γ编码，然后依次存储 12，13，34，22 对应的哈夫曼编码，接着在存储 1 的γ编码，然后依次存储 1，921 对应的哈夫曼编码。

(4)、为了进行解压缩和查询操作，将 WEB 图压缩过程中进行哈夫曼编码的所需要的辅助数据信息保存到一个文件中；

(5) 、为了支持随机访问压缩后 WEB 图中节点的信息，将压缩后 WEB 图中每个节点信息在压缩文件中的起始位置保存在一个位置信息文件中。

由于使用了类 Golomb 编码和 Run-Length&Golomb 编码，使得分布在很广范围内的数字分布在了一个比较小的范围内，对这个小范围内的节点进行哈夫曼编码，可以减少编码每个节点所需要的位数，同时，也能显著的减少辅助数据结构所占用的空间。因此只要 WEB 图的节点个数不大于2^{32}，在压缩之后，保存哈夫曼编码信息的辅助文件的大小将始终小于 700K。

将压缩 WEB 图的最小单位分割到位(bit)，是为了尽量减少空间的浪费，使得压缩后的 WEB 图尽量占用少的空间，同时可以将整个压缩后的 WEB 图看成一个很长的位向量，使得以后对 WEB 图进行解压缩和查询操作时更加的方便。 但是由于计算机内部数据类型的限制，在 32 操作系统上，压缩后的 WEB 图超过 512MB 之后，进行随机访问将需要占用更多的内存空间[6]，而且压缩 WEB 图后得到的

位置信息文件也将显著的增大。

保存压缩后 WEB 图中节点的位置信息文件，是为了在解压缩或者随机访问节点信息的过程中，可以快速的定位节点的压缩信息在压缩文件中的位置，这样就可以显著提高解压缩和查询的速度，同时也使得压缩后的 WEB 图能够支持节点的随机访问操作。由于每个节点的位置信息长度都是等长的，信息存储方式类似于一个数组，因此在需要查询节点的起始位置时，可以通过节点的 ID 在O(1)的时间内获得，不用顺序查找，显然这样也能提高解压缩和查询节点信息的速度。

第四章、WEB 图的解压缩与查询

4.1、WEB 图的解压缩与随机访问

在压缩 WEB 图的过程中，需要对每个节点进行参考编码，编码之后该节点的压缩信息都是相对于参考节点的信息，只有在完全知道参考节点信息的情况下，才可以通过该节点的压缩信息解码出节点的完整信息。在寻找最合适参考节点的过程中，使用的是通过滑动窗口在该节点前的 W 个节点和虚拟节点 R 之中选择一个使得 cost 函数的值最小的节点作为参考节点，并根据计算出来的参考节点进行参考编码，纵观 WEB 图中所有节点与其参考节点的关系，可以确定它们也组成了一棵树，树的根节点为虚拟节点 R，为了方便，以后就称这棵树为参考链。

WEB 图的解压缩指的是将压缩后的 WEB 图文件和其它相关文件通过与压缩 WEB 图时相反的方式，将压缩后的 WEB 图进行解压缩还原，并且要求解压缩得到的 WEB 图的每个节点都和原来未压缩时的 WEB 图一模一样。WEB 图的查询指的是，给定一个节点 ID，通过与解压缩 WEB 图类似的方式，从压缩后的 WEB 图中，提取出该节点的完整信息。通过分析可得，查询是解压缩的一个中间过程或者子过程，按照一定的顺序查询每个节点的过程就是解压缩 WEB 图的过程。

在对 WEB 图进行解压缩和查询节点信息的过程中，通过节点的压缩信息在压缩后的 WEB 图文件中的偏移位置，可以快速的提取出该节点的压缩信息，但是这个信息时相对于参考节点的，并不是该节点的完整信息，为了解码出该节点的完整信息，需要先解码出该节点的参考节点的完整信息，由此需要沿着节点和参考节点组成的树形结构向上重复该过程直到根节点，至此，从该节点反向到达

根节点的整个过程中，沿途所有节点的参考编码的信息都被提取出来，为了获得每个节点的完整信息，需要根据每个节点的参考节点和该节点相对于参考节点的位向量，由根节点向下解码每个节点的完整信息，直到被查询的节点。因此，查询节点信息的过程可以概括为，由该节点沿着参考编码的树形结构反向提取沿途每个参考节点的信息直到根节点，然后根据参考编码的相关信息沿着参考编码组成的结构向下还原直到被查询的节点，至此该节点的信息被完全解码出来。整个过程需要花费的时间与从根节点到被查询节点的距离成正比。

4.2、DecodeTree

由于在节点的实际压缩格式表示中，有某些部分使用的是哈夫曼编码，在进行解压缩和随机访问操作时需要准确地解码这些信息。通过分析哈夫曼树可知，如果有 N 个叶子节点，那么整个哈夫曼树中节点的个数一定为$2 * \mathrm{N} - 1$，而且哈夫曼树的左孩子对应的编码 0，哈夫曼树的右孩子对应 1，从根节点按照编码向下查找，编码为 0 移动到左子树，编码为 1 移动到右子树，直到哈夫曼树的叶子节点为止，其中叶子节点的信息即为该哈夫曼编码对应的值，这样就可以通过哈夫曼树在$\Theta(log\ N)$的时间复杂度内将某个哈夫曼编码对应的信息解码出来。

通过压缩后 WEB 图后的产生的与哈夫曼编码相关的辅助数据文件，可以将该文件中对应的数据与该数据对应的哈夫曼编码还原成一棵哈夫曼树，称为解码树(DecodeTree)，解码树的构造算法如下：

```
CreateDecodeTree(nodes)
1    size ← 2 * nodes.size − 1
2    root ← new TreeNode[size]
3    root[1].info ← MAX
4    root[1].left_child ← NULL
5    root[1].right_child ← NULL
6    index ← 2
7    for i ← 1 to node.size
8    do
9        ptr ← root[1]
10       for j ← 1 to nodes[i].vector.length
11       do
12           pre ← ptr
13           if nodes[i].vector.test(j)
14           then
15               ptr ← ptr.right_child
```

```
16          else
17              pt ← ptr.left_child
18          if ptr = NULL
19          then
20              newnode ← root[index]
21              index ← index + 1
22              newndoe.left_child ← NULL
23              newnode.right_child ← NULL
24              node.info ← MAX
25              if nodes[i].vector.test(j)
26              then
27                  pre.right_child ← newnode
28              else
29                  pre.left_child ← newnode
30              pre ← newnode
31      ptr.info ← nodes[i].info
32  return root
```

在构造完解码树之后，可以根据节点在压缩文件中的起始位置信息对节点信息进行解码，要求解码算法返回从位向量中的指定位置开始进行解码得到的信息和下一个信息在位向量中的起始位置，方便进行下一次的解码，解码算法伪代码如下：

```
Decode(root,vector,position)
1   ptr ← root
2   index ← positon
3   while ptr != NULL && ptr.info = MAX
4   do
5       if vector.test(index)
6       then
7           ptr ← ptr.right_child
8       else
9           ptr ← ptr.left_child
10      index ← index + 1
11  if ptr = NULL
12  then
13      error "decode error"
14  else
15      return ptr.info index
```

4.3、rest_links 的解码

在开始解码节点的 rest_links 部分时，节点的 rest_links 信息之前的信息已经解码出来，此时就可以计算出 rest_links 中包含的真实链接的个数。假设 rest_links 部分包含的真实链接个数为 size，L1、L2、 ……、Ln 为在压缩后的图中解码出来

的 rest_links_length 的长度，并且 Ln 一定为 1。RL1、RL2……、RLn 为真实的进行了 Run-Length 编码剩余链接中后每个块的长度。例如在进行 WEB 图的存储之前，假设 rest_flag_golomb=false、rest_links={12,13,34,22,1,921}、rest_links_length={2,1}，那么$L1=2$，$L2=1$，$RL1=4$，$RL2=1$。由于为了方便进行γ编码，在进行 Run-Length&类 Golomb 编码时将链接的个数加 1，因此在这里也需要将 size 加 1。在对 size 加 1 之后，就可以得出以下等式：

(1)、SIZE=size+1；

(2)、RL1=SIZE-L1；

(3)、RL2=SIZE-RL1-L2=SIZE-(SIZE-L1)-L2=L1-L2

(4)、RL3=SIZE-RL1-RL2-L3=SIZE-(SIZE-L1)-(L1-L2)-L3=L2-L3

……

(5)、RLn=SIZE-RL1-RL2-…..- Ln=L(n-1)-1

这个公式在对压缩后的 WEB 图中的 rest_links 部分进行解码时候是很有用的，因为它可以计算出真实的每个块的实际长度，这样就不会出现访问越界，导致解码错误的情况。

4.3、节点信息的解压缩

从节点在 WEB 图压缩文件中的起始位置起，第一个要解码的是节点的出度 outdegree，接着是节点的参考节点 reference，通过节点的参考节点又可以解码出参考节点的出度$outdegree_{ref}$，接下来解码出 block_count，block_flag，通过$outdegree_{ref}$，block_count，block_flag 以后和连续的 blocks 可以解码出进行参考编码之后的位向量 vector，通过统计 vector 中 1 的个数，可以计算出差异部分中链接的个数。然后对间隔部分进行解码就可以解码出节点的间隔部分的信息。通过节点的出度，相对于参考节点的 vector 中 1 的个数，以及对间隔部分解码后得到的链接个数，就可以计算出剩余链接的个数，再对剩余链接部分解码就可以得出节点相对于参考节点的全部信息。提取每个节点相对于参考节点的信息的过程如上所述，然后结合上一节的解压缩与查询方法就可以解压缩出某个节点甚至整个 WEB 图的完整信息。

假设 WEB 图中节点与参考节点组成的部分树形结构如图 4.1 所示，那么解码

节点 D 的完整信息的过程如下：

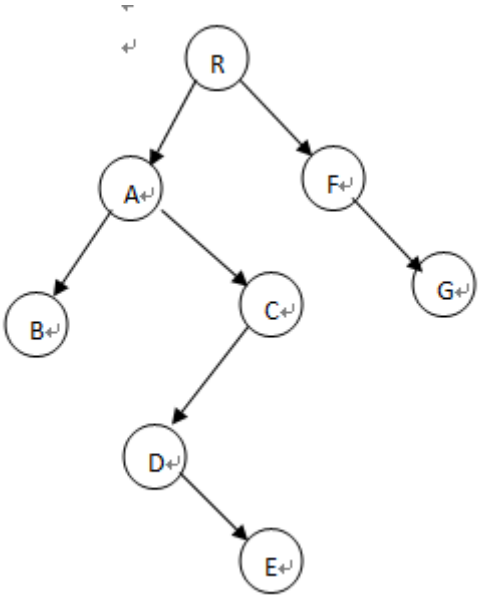

图 4.1、压缩过程中最小树形图部分结构

(1)、根据节点 D 的压缩信息在压缩文件中的起始位置，结合哈夫曼编码构建而成的 DecodeTree，解码并还原出出节点的出度，参考节点；

(2)、根据解压缩出的参考节点 C，解码出节点 C 的出度；

(3)、然后依次解码并还原出节点 D 相对于参考节点 C 的位向量，以及节点 D 的连续链接集合和剩余链接等信息；

(4)、重复执行上述过程，解压缩出节点 C 的相应信息；

(5)、重复执行上述过程，解压缩节点 A 的相应信；

(6)、由于节点 A 的参考节点为 R，因此此时得到的节点 A 的信息就是节点 A 的完整信息。

(7)、根据节点 A 的完整信息以及节点 C 相对于节点 A 的位向量，计算出节点 C 的完整信息。

(8)、根据节点 C 的完整信息以及节点 D 相对于节点 C 的位向量，计算出节点 D 的完整信息。

至此，节点 D 的完整信息被解码出来。

4.3、解压缩加速算法

通过分析 WEB 图的解压缩和查询过程可知，每次要查询一个节点的信息，

都需要从该节点反向解压缩每个节点相对于参考节点的信息至根节点，然后在从根节点向下还原沿途的每个节点的完整信息，直到被查询的节点。目的是只查询一个节点，但是却将该根节点到该节点的路径中的所有节点信息解压缩出来，而且，如果目的不是解压缩 WEB 图，只是查询某些节点的信息，为了节省存储空间，已经被查询出来的节点可能并没有被保存，那么下次查询其它相关节点时，需要将这些节点都重新查询一遍，无法重用之前查询出的结果，因此，整个解压缩算法的效率并不是很高，例如在上一节对节点 D 的查询过程中，节点 A 和节点 C 的完整信息都在查询节点 D 的过程中已经被完整的查询出来。如果，查询出节点信息并没有被保存起来，那么下次查询节点 E 的信息时，又需要对节点 D，节点 C 和节点 A 进行一次新的查询，导致需要做很多重复的操作，降低了查询的效率。

通过对节点与节点的参考编码构成的树进行分析可知，查询一个节点信息所花费的时间与根节点到该节点的距离是成正比的，要想减少解压缩和查询的时间，需要减少反向向根节点方向查询的次数。有两种可行的方法，一种是尽量减少每个节点距离根节点的距离，以此来减少反向查询的次数；另一种方法是，每次反向解压缩时不用查询到根节点，而是查询到一个中间节点，就能将节点的信息完全查询出来。第一种方法，由于需要改变节点与其参考节点构成的树的结构，在论文中称为参考链的整形与优化，第二种方法，由于需要保存中间查询结构，且需要使用一定的替换算法，类似计算机内存中的 Cache 调度，因此在论文中称之为 Cache 加速算法。

4.4、参考链的整形与优化

使用滑动窗口技术计算节点的参考节点时，由于后面的节点参考前面的节点或者根节点，那么在整个 WEB 图中就可能存在一些很长的参考链，当对这些参考链底部的节点进行解压缩操作时，将需要花费很长的时间。为了减少反向查询的次数，通过分析可知可以通过将某些节点的参考节点调整为根节点以减少它下面所有节点到根节点的距离来减少查询时花费的时间，如图 4.2 所示，将节点 D 的参考节点由节点 C 调整为根节点，这样就能显著缩短节点 D，节点 E 和节点 H 的查询时间。整形与优化的要求是既要保证稳定、良好的压缩效果，又要减少解

压缩和查询的时间，因此需要设计特定的算法。

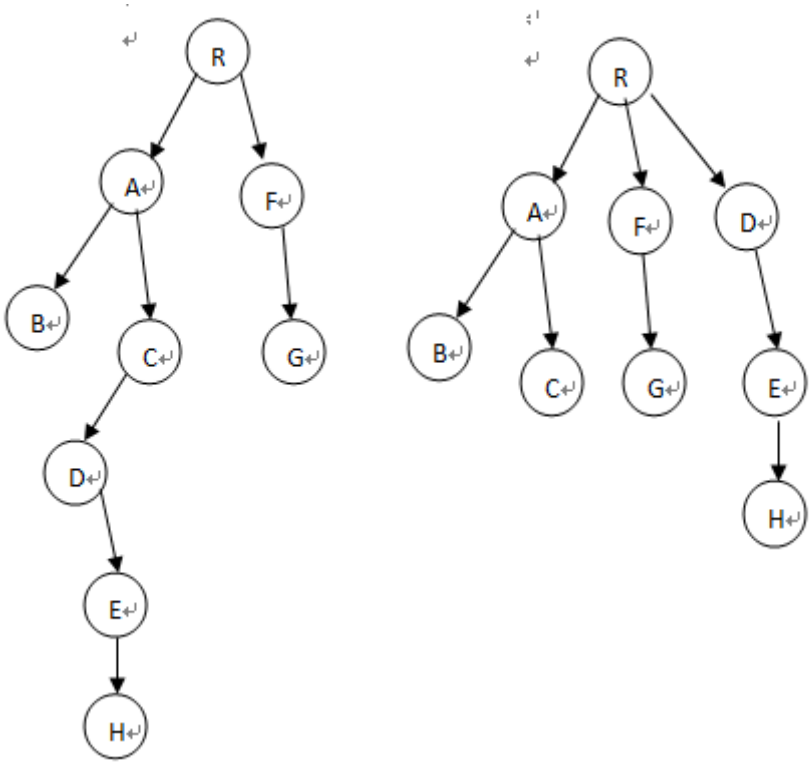

图 4.2、对最小树形图整形与优化前后

为了将某些节点的参考节点调整为根节点，需要对在参考链中超出一定距离之外的某些节点，尽量挑选最合适的进行调整。通过分析可知，在参考链中距离根节点越远的节点，越需要进行调整，同时对于节点 i，假设其参考节点为节点 j，那么$\mathrm{cost(i,root)}$与$\mathrm{cost(i,j)}$的差值越小的点也越适合进行调整。但是在调整的过程中存在一定的优先级关系，如果以到根节点的距离作为主要的因素，那么将很有可能会影响压缩效果，如果以$\mathrm{cost(i,j)}$与$\mathrm{cost(i,root)}$作为主要因素，则可能存在部分点仍然距离根节点较远，为了保证良好的压缩效果，实验中，选择以$\mathrm{cost(i,j)}$与$\mathrm{cost(i,root)}$的差值作为主要因素。因此可以使用如下公式进行计算：

$$\mathrm{adjust(i)} = \frac{\mathrm{cost(i,j)}}{\mathrm{cost(i,root)}}$$

公式 4.1、adjust 函数

通过分析可以，adjust(i)一定小于等于 1，对于超出指定距离外的节点可以使用此公式计算调整的参考值 adjust(i)，参考值越大表示越适合将参考节点调整为根节点，当参考值大于一定的值时，表示使用 root 节点编码节点 i 与使用节点 j 编码节点 i 的开销相差不大，为了提高解压缩效率，将参考节点调整为根节点

所付出的的代价是值得的。

4.5、Cache 加速算法

Cache，俗称高速缓存存储器，是存在于计算机 CPU 与内存之间的一个容量小，访问速度与 CPU 速度基本一致的存储器。根据程序局部性原理，正在使用的主存储器某一单元邻近的那些单元将被用到的可能性很大。因而，当中央处理器存取主存储器某一单元时，计算机硬件就自动地将包括该单元在内的那一组单元内容调入高速缓冲存储器，中央处理器即将存取的主存储器单元很可能就在刚刚调入到高速缓冲存储器的那一组单元内。于是，中央处理器就可以直接对高速缓冲存储器进行存取。在整个处理过程中，如果中央处理器绝大多数存取主存储器的操作能为存取高速缓冲存储器所代替，计算机系统处理速度就能显著提高。

由于在本算法中使用了类似计算机内存的 Cache 模型，并且主要目的是加速解压缩和查询的过程，因此称之为 Cache 加速算法。在计算机的 Cache 模型中，很重要的一个算法就是 Cache 的替换算法，因为 Cache 替换算法的好坏与 Cache 命中率直接相关，命中率越高表示 Cache 的加速效果越好。在 Cache 加速算法中也是如此，最重要的也是 Cache 替换算法，由于按照参考链进行解压缩的模型与内存访问模型并不是一样的，因此，针对内存访问使用的各种 Cache 替换算法，根据参考链进行解压缩模型并不能直接使用，需要设计使用于解压缩模型的替换算法。

通过分析按照参考链进行解压缩节点信息的过程可以知道，如果从根节点到每个叶子节点的路径的中间位置都存在一个节点已经被查询出，并且已经被保存，假设称呼它们为中间节点。那么所有在中间节点的下边（离根节点较远的一边）的节点，在对他们进行查询时，只需要反向查询到中间节点就可以查询出全部的信息，此时中间节点相当于一个代理根节点，而所有在中间节点的上边（距离根节点较近的一边）的节点可以直接反向查询到根节点来查询出全部的信息，由于查询节点信息的时间与节点到根节点的距离是成正比的，因此可以将查询时间缩短到原来的一部分。同时，如果从根节点到某个叶子节点的路径中，每隔一段距离就存在一个中间节点的信息被查询出并且被保存，那么在对任意两个中间节点之间的节点，叶子节点到中间节点之间的节点和中间节点到根节点之间的节点进

行查询时，只需要反向查询到上一个中间节点或者根节点即可查询出全部的信息，显然，这样可以减少查询的时间。如图 4.3 所示，假设节点 C 是一个中间节点，那么在对节点 D，节点 E，节点 H 进行查询操作时，只需要沿着最小树形图结构反向查询到节点 C，然后再向下还原沿途的节点即可完全解码出节点的全部信息。因此，在 WEB 图的解压缩模型中，设计 Cache 替换算法的主要目的是使得尽量在根节点到每个叶子节点的路径的中间位置都存在已经查询出并且已保存的中间节点，同时在根节点到某个叶子节点的路径中，每隔一段距离，就有一个中间节点被查询出并且保存，以此来减少解压缩 WEB 图和查询节点信息所花费的时间。

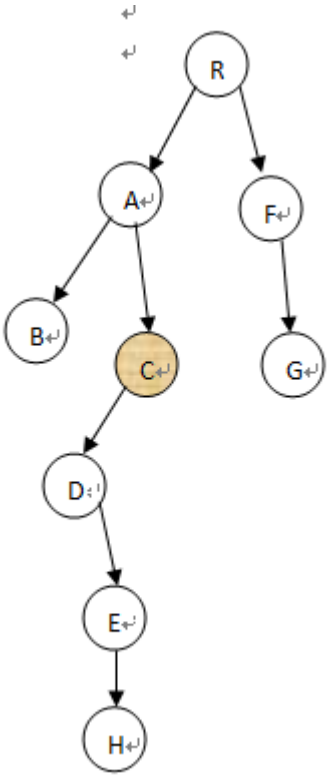

图 4.3、Cache 加速模型

在以参考链的树形结构分析整个 WEB 图时，发现了一个很有用的规律。由于参考节点是基于滑动窗口技术计算得来的，而滑动窗口技术中使用 cost 函数表示以一个节点作为参考节点编码另一个节点的开销，cost(i, j)越小表示节点 j 越适合编码节点 i，因此在最小树形图中，一般任意两个节点之间的距离（从一个节点到达另一个节点所需要的最少路径）越大，那么两个节点之间相似的链接就越少。所以，可以以节点之间相似的链接的个数作为设计 Cache 替换算法的依据。在实验过程中，使用的 Cache 替换算法如下：

```
InsertCache(cache, size, limit, cachenode)
1   if cache.size < size
1   then
2       cache.insert(cachenode)
3       return
4   value ← 0
5   for i ← 1 to cache.size
6   do
7       temp ← Similarity(cachenode, cache[i])
8       if temp < limit
9       then
10          if temp > value
11          then
12              value ← temp
13              index ← i
14      else
15          return
16  if i = cache.size
17  then
18      cache[index] = cachenode
```

在内存中分配一块不大的内存充当 Cache 的角色，当对压缩后的 WEB 图执行节点的查询操作时，在从节点反向解压缩节点信息到根节点的过程中，计算节点到根节点的距离，假设为 length，那么再从根节点依次还原节点信息到被查询节点的过程中，将距离根节点距离为$(length + 1)/2$的位置的节点的信息保存到 Cache 中，如果 Cache 记录未满，则直接插入 Cache 中即可，如果 Cache 中记录已经达到最大限制，则通过 Cache 替换算法进行替换。在执行 Cache 替换算法时，计算新插入节点与所有节点的相似度，如果在 Cache 中存在节点与新插入节点的相似度大于一个指定的限制值，那么说明在 Cache 中已经存在和节点比较相似的节点，通过前面的分析，可以认为是与被插入节点的距离很短的节点，那么新节点将不会被插入到 Cache 中；如果新节点与所有节点的相似度都小于某一限制值，那么将选择一个最相似的进行替换，因为两个节点越不相似，可以认为他们之间的距离越远，选择这样的替换策略是为了使得在 Cache 中的节点在整个参考链构成的树形结构中分布的更加均匀。

假设节点 i 为 Cache 中的节点，节点 j 为要插入的节点，那么可以按照如下公式计算相似度：

$$\text{Similarity}(i,j) = \frac{|N(i) \cap N(j)|}{|N(i)|}$$

公式 4.2、similarity 函数

其中N(i),N(j)分别为节点 i 和节点 j 的链接集合。可见相似度是一个分数，最小为 0，最大为 1，在实验过程中限制值的一般取值为 0.3~0.5 之间。

第五章、实验结果与分析

5.1、WEB 图的压缩

5.1.1、实验结果

实验数据都是在 SNAP 以及 Laboratory for Web Algorithmics 官方网站上下载，主要用于实验测试，这样得出的实现结果会更有说服力。由于在进行压缩时，这些网站给出的数据格式与我的程序所能读写的格式并不一样，因此我对这个 WEB 图数据格式都进行了转化，我使用的数据格式是使用邻接表方式加二进制存储，保存每个节点信息时，都是依次使用二进制格式保存链接个数，然后使用二进制格式再保存该节点的链接信息。实验结果如下表所示，其中 name 为该 WEB 图的名字，node 为 WEB 图的节点个数，edge 为 WEB 图的有向边的个数，max-out 为 WEB 图中节点的最大出度，avg-out 为 WEB 图中节点的平均出度，window 为选择的滑动窗口大小，.graph 为将其它 WEB 图格式数据转换为我的程序能支持的格式后的 WEB 图文件的后缀名，表示未压缩的 WEB 图，.comp 为压缩后 WEB 图文件的后缀名，表示压缩之后的 WEB 图， per-edge 为压缩后每条边占用的位数，ratio 为原始 WEB 图文件与压缩后 WEB 图文件的比，称为压缩比。

name	node	edge	max_out	avg_out
berkstan	685230	7600595	249	11.1
.graph	window	.comp	per-edge	ratio
32367KB	100	2829KB	3.049	11.411
32367KB	200	2803KB	3.020	11.547
name	node	edge	max_out	avg_out
uk-2007-05@100000	100000	3050615	3753	30.5
.graph	window	.comp	per-edge	ratio
12308KB	100	588KB	1.578	20.931
12308KB	200	585KB	1.570	21.040
name	node	edge	max_out	avg_out

cnr-2000	325557	3216152	2716	9.9
.graph	window	.comp	per-edge	ratio
13832KB	100	1069KB	2.721	12.939
13832KB	200	1068KB	2.720	12.951
name	node	edge	max_out	avg_out
eu-2005	862664	19235140	6985	22.3
.graph	window	.comp	per-edge	ratio
78508KB	100	7655KB	3.260	10.256
78508KB	200	7586KB	3.230	10.349
name	node	edge	max_out	avg_out
indochina-2004	7414866	194109311	6985	26.2
.graph	window	.comp	per-edge	ratio
787200KB	100	29345KB	1.238	26.826
787200KB	200	29292KB	1.236	26.874

图 5.1、WEB 图压缩的实验结果

5.1.2、结果分析与结论

从以上实验数据可知，对 WEB 图进行压缩之后，压缩后每条边占用差不多 2-3bit 左右，而且对于某些图甚至能够低于 2bit，其压缩效果还是很理想的。而且同一组数据，不同的滑动窗口大小产生的压缩效果也是不一样的，当滑动窗口越大时，找到的参考节点就越合适，这样就会显著减少进行参考编码之后得到的差异部分，进而获得更好的压缩效果。通过实验数据的分析，发现每组数据中 per-edge 与 ratio 的乘积都稍微大于 32，而且 avg_out 越小， per-edge 与 ratio 的成绩越大。这是因为，在未压缩的 WEB 图中，每条边都是使用 32 位来表示，因此，并且额外使用了 32 位来存储 WEB 图中节点的链接个数，当 WEB 图的平均出度很大时，存储链接个数的单元产生的影响比较小，因此，per-edge 与 ratio 的乘积也比较小，否则，per-edge 与 ratio 的乘积就会比较大。

5.2、WEB 图的解压缩与查询

5.2.1、实验结果

WEB 图的解压缩与查询主要是在上述压缩过后的 WEB 图上执行解压缩与随

机查询信息的操作，实验结果如下图，其中 name 与压缩时一样，可以用来代表某个 WEB 图，decompress 代表直接解压缩某个 WEB 图时花费的时间，decompress-ac 表示加入了 Cache 加速之后解压缩 WEB 图花费的时间，access 为未加入 Cache 加速的情况下，随机访问 10000 个节点所花费的时间，access-ac 表示加入了 Cache 加速之后随机访问 10000 个节点所花费的时间，所有的时间都精确到毫秒，所有结果都是滑动窗口限制为 100 时的测试结果。

name	node	decompress	decompress-ac	access	access-ac
berkstan	685230	355634	116589	5132	2482
uk-2007-05@100000	100000	48763	178952	4653	2235
cnr-2000	325557	170592	59256	5045	2146
eu-2005	862664	424556	148809	4987	2098
indochina-2004	7414866	3803830	1356920	5126	2315

图 5.2、WEB 图解压缩的实验结果

5.2.2、实验结果分析与结论

在对 WEB 图进行解压缩的实验中，通过五个不同的 WEB 图进行实验，可以得出在解压缩 WEB 图的过程中，在不进行 Cache 加速的情况下，随机访问一个节点需要大约花费 0.5 毫秒左右的时间。在进行 Cache 加速的情况下，解压缩时间可以缩短为原来的 1/3 到 1/2 之间。在对 10000 个节点进行随机查询的操作中，随机访问一个节点的平均时间与解压缩时相差并不大，但是添加了 Cache 之后，虽然起到了加速的效果，但是效果相对于解压缩时却差了好多，经过分析，一个很重要的原因就是查询的 10000 个节点数量过少，因此 Cache 模块的初始阶段起到的加速效果并不大，在经过一定次数的 Cache 替换之后，存在 Cache 中的节点分布相对更加均匀，因此才开始发挥 Cache 加速的作用。

第六章、总结与展望

本论文给出了对 WEB 图的进行压缩的一个可行方案，并且通过理论与实验论证了方案的可行性，最终该方案获得比较理想的压缩效果。对于压缩后的 WEB 图，为了获得更快的随机访问速度，给解压缩模型增加了 Cache 加速模块以及参考链整形与优化模块，改善了解压缩过程中会碰到的问题。但是虽然压缩效果比

较理想，而且解压缩模型中增加了优化的模块，但是节点的随机访问时间相对于某些已成熟的方案还是比较慢的，因此在解压缩方面还有很大的提升空间。对于节点的压缩表示部分，还可以结合其它的更优良的编码方法，给节点进行更好的压缩表示等等。

致谢

参考文献

[1] Broder A, Kumar R, Maghoul F, et al. Graph structure in the web[J]. Computer networks, 2000, 33(1): 309-320.

[2] Adler M, Mitzenmacher M. Towards compressing web graphs[C]//Data Compression Conference, 2001. Proceedings. DCC 2001. IEEE, 2001: 203-212.

[3] Raghavan S, Garcia-Molina H. Representing web graphs[C]//Data Engineering, 2003. Proceedings. 19th International Conference on. IEEE, 2003: 405-416.

[4] Suel T, Yuan J. Compressing the graph structure of the web[C]//Data Compression Conference, 2001. Proceedings. DCC 2001. IEEE, 2001: 213-222.

[5] Randall K H, Stata R, Wickremesinghe R G, et al. The link database: Fast access to graphs of the web[C]//Data Compression Conference, 2002. Proceedings. DCC 2002. IEEE, 2002: 122-131.

[6] Boldi P, Vigna S. The webgraph framework I: compression techniques[C]//Proceedings of the 13th international conference on World Wide Web. ACM, 2004: 595-602.

[7] Chierichetti F, Kumar R, Lattanzi S, et al. On compressing social networks[C]//Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2009: 219-228.

[8] Claude F, Navarro G. A fast and compact Web graph representation[C]//String Processing and Information Retrieval. Springer Berlin Heidelberg, 2007: 118-129.

[9] Gilbert A C, Levchenko K. Compressing network graphs[C]//Proceedings of the LinkKDD workshop at the 10th ACM Conference on KDD. 2004.

[10] Boldi P, Vigna S. Codes for the world wide web[J]. Internet mathematics, 2005, 2(4): 407-429.

[11] Boldi P, Vigna S. The webgraph framework ii: Codes for the world-wide web[C]//Data Compression Conference, 2004. Proceedings. DCC 2004. IEEE, 2004: 528.

[12] Apostolico A, Drovandi G. Graph compression by BFS[J]. Algorithms, 2009, 2(3): 1031-1044.

[13] Boldi P, Vigna S. Codes for the world wide web[J]. Internet mathematics, 2005, 2(4): 407-429.

[14] Kamvar S, Haveliwala T, Manning C, et al. Exploiting the block structure of the web for computing pagerank[J]. Stanford University Technical Report, 2003.
[15] Golovin D, Pandey S. WebStore: Efficient Storage and Access of the Web[J].
[16] Asano Y, TOYODA M, KITSUREGAWA M. Compact Encoding of the Web Graph Exploiting Various Power Distributions[J]. IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, 2004, 87(5): 1183-1184.
[17] Asano Y, Miyawaki Y, Nishizeki T. Efficient compression of web graphs[M]//Computing and Combinatorics. Springer Berlin Heidelberg, 2008: 1-11.
[18] Hernández C, Navarro G. Compressed representations for web and social graphs[J]. Knowledge and Information Systems, 2013: 1-35.
[19] Mordecai J. Golin, Claire Kenyon, Neal E. Young "Huffman coding with unequal letter costs" (PDF), STOC 2002: 785-791
[20] Elias, Peter (March 1975). "Universal codeword sets and representations of the integers". IEEE Transactions on Information Theory **21** (2): 194–203.
[21] Sayood,Khalid (2003). "Levenstein and Elias Gamma Codes". Lossless Compression Handbook. Elsevier. ISBN 978-0-12-620861-0.
[22] Golomb, S.W. (1966). , Run-length encodings. IEEE Transactions on Information Theory, IT--12(3):399—401