In-memory URL Compression

Kasom Koht-arsa
Department of Computer Engineering, Faculty of Engineering, Kasetsart University,
Bangkok, Thailand 10900
E-mail: g4265077@ku.ac.th

Surasak Sanguanpong
Department of Computer Engineering, Faculty of Engineering, Kasetsart University,
Bangkok, Thailand 10900
E-mail: nguan@ku.ac.th

Abstract

A common problem of large scale search engines and web spiders is how to handle a huge number of encountered URLs. Traditional search engines and web spiders use hard disk to store URLs without any compression. This results in slow performance and more space requirement. This paper describes a simple URL compression algorithm allowing efficient compression and decompression. The compression algorithm is based on a delta encoding scheme to extract URLs sharing common prefixes and an AVL tree to get efficient search speed. Our results show that the 50% of size reduction is achieved.

1. Introduction

Storing, searching and retrieving a large number of URLs is one of the problems in designing a large scale search engines and web spiders. Search engines use the URL to show the web page relevant to user queries, while spiders have to keep track of every URL in order to check whether a URL has been visited. To store 100 million URLs, with the average size about 55 bytes, would require at least 55x108 bytes (approximately 5 GBytes) of storage space.

While system processor and memory speeds have continued to increase rapidly, application performance is still constrained by slow disk and I/O speeds. Most disks still have an average seek time about 4 to 12 millisecond. It implies that the disk can be accessed randomly only about 80 to 250 times per second. This will degrade the performance of search engines and web spiders in storing and retrieving of URLs.

Since the memory is many orders of magnitude faster than the hard disk, storing URLs in memory is result in performance improvements. However, storing full URLs in memory is impractical because there are only few machines that offer a big size of main memory. Furthermore, most 32 bits machines/operating systems have a maximum virtual memory limit a process may have about one to three gigabytes. To overcome this circumstance, a compression method to reduce the URLs size is required.

The Archive's[1] solution to this problem is to use a giant bitmap approach. A chunk of memory is allocated for a bitmap, then initialized it to zero. For any URL, ten hash values are computes using ten different hashing algorithms. These hash values are checked against the bitmap. If there are any bits in the location pointed by those hash values that is not set, this URL has not be seen before. All the location pointed by the hash values are then set. This method relies on the hash function it uses, that may produce a "false positives." Furthermore, the method can't fulfil the search engine's requirements to retrieve the URLs back.

The Adaptive Web Caching project [2] uses a simple URL table compression that is based on a hierarchical URL decomposition to aggregate URLs sharing common prefixes and an incremental hashing function to minimize collisions between prefixes. While this method is good for constructing a forwarding table for the web cache to locate the nearest copy of a requested URL's contents, it's too coarse to be used with a search engine or a web spider. It lacks the ability to reconstruct the actual URL (needed by search engines) and may also produce a false positives.

The Connectivity Server [3] store the URLs by sorting them lexicographically and then store them as a delta-encoded text file. Each entry is stored as the difference (delta) between the current and previous URL. Since the common prefix between two URLs from the same server is often quite long, this scheme reduces the storage requirements significantly. This method requires that URLs have to be sorted first. After the compressed URLs has been constructed, no URL can be added later.

Furthermore, there is no way to search for the existence of any given URL, except to scan through the list of URLs. Hence, it is not suitable for web spiders as well.

2. Approach

Like the delta-encoding, our method compresses the URLs by only keeping the differences of URLs tails. To find the URLs different, the sorting of URLs is required. However, a new URL is on the fly discovered and it is impractical to sort out the URLs list every time a new URL is discovered. Instead of sorting all URLs, we incrementally construct an AVL tree[6] of URLs.

An AVL tree is a special kind of a binary search tree. It derives a property of a binary search tree that for any given node, the predecessor or the successor of that node must be one of its root node, or the maximum node of the left child, or the minimum node of the right child. Finding the predecessor/successor of a newly added node is much easier, because they must be one of its root node that it travel pass. That is because before the rotation of an AVL tree, the newly added node has no child.

In our method, all URLs are stored on each node of an AVL tree. A node structure is illustrated by like in Figure 1. Each node in the tree contains five fields: `RefID` is a URL identification used to reference to its predecessor, `CommonPrefix` is a number of common characters referenced to its predecessor, `diffURL` is the tail of the URL that does not common to its predecessor, `Lchild` and `Rchild` are the pointers to the node's left subtree and right subtree respectively.

RefID	CommonPrefix	diffURL	Lchild	Rchild

Figure 1 A node structure of the AVL tree

The first encountered URL, which is the root, is assigned with the `RefID` 0. The `RefID` is increment by one for every new encountered URLs. Please note that the value in the field `RefID` of the root is undefined. The common prefix is set to zero, and the full URL is stored. The next or any latter URL must be compared with every node on the path between the root node and its predecessor to find the maximum common prefix. The reference ID of a new node is then point to that node, and the common prefix is set to the number of common characters, and the remaining of the URL is then stored. This is the way the compression is done.

Figure 2 shows an example of a construction of an AVL tree from the following URLs : *http://www.sun.com/*, *http://www.sgi.com/*, *http://www.sun.com/news/*, and *http://www.sun.com/news/archive/*.

The first URL always become the root of the tree : this is *http://www.sun.com/*. Its reference URL ID is set to zero, its shared prefix is also set to zero and the whole URL is stored in the third field. The next URL, *http://www.sgi.com/* is compared to *http://www.sun.com/*. Since it is "smaller", so the URL is attached to the right subtree of *http://www.sun.com/*. This URL has a 11 bytes common prefix of *http://www.*, so it is stored as *sgi.com/* with a reference ID 0. The next URL, *http://www.sun.com/news/*, is "greater" than *http://www.sun.com/*, so we attach a new node with *news/* as the right subtree *http://www.sun.com/* with 19 bytes common prefix. The last URL, *http://www.sun.com/news/archive*, it compared with the root and is "greater" than both *http://www.sun.com/* and *http://www.sun.com/news/*, so we attach a new node with *archive/* as the right subtree of *news/*. The common prefix is 24 bytes long (*http://www.sun.com/news/*) and the node is pointed back to URL ID 2.

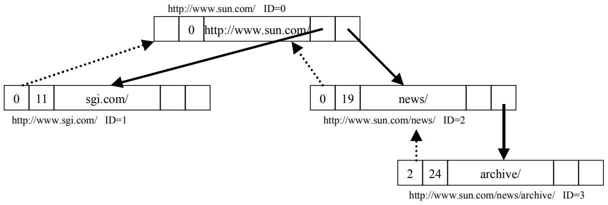

Figure 2 Compressed URL on an AVL tree

Retrieving any URL from the tree is very simple. The full URL can be re-constructed by following the path and concatenate all URL from the field `diffURL`. This approach can fulfil all the requirements of search engines and web spiders. The URLs can be added any time, the URLs can be searched, and the full URL can be retrieved.

3. Implementation

The AVL tree is implemented by three arrays. The first array, the **TreeNode**, contains a list of nodes of the AVL tree as shown in Figure 3. It contains a Left node ID, right node ID, and the height. The variable length data were stored on the second array called **CompressedURL**, as shown in Figure 4. The **CompressedURL** is accessed through the third array, the **DataPtr** as shown in Figure 5. These tree arrays must be pre-allocated. The reason behind this implementation is because the compiler can produce an optimum code if the data can be aligned in 2^n bytes boundary. Furthermore, the **TreeNode** is used only for adding a new URL and can be discarded if no URL is left. Hence, after the tree has been completely constructed, the **TreeNode** is no more required.

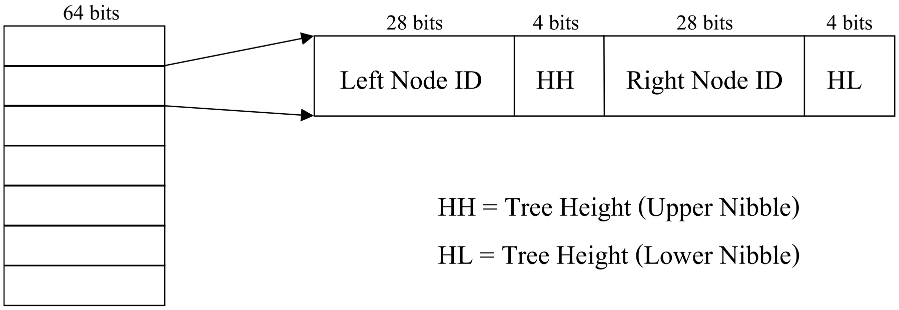
64 bits
28 bits
4 bits
28 bits
4 bits
Left Node ID
HH
Right Node ID
HL
HH = Tree Height (Upper Nibble)
HL = Tree Height (Lower Nibble)

Figure 3 TreeNode

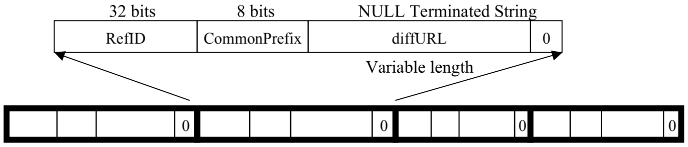

Figure 4 CompressedURL

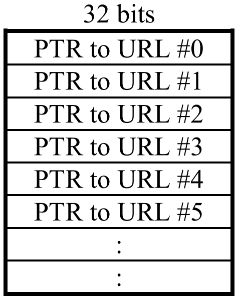

Figure 5 DataPtr

The height of each node is split into two nibble and store on the 64-bit structure on the first array. This can reduce memory space, and the maximum URLs is equal to 2^{28} entries, or around 268 million URLs. Like in Figure 6, **TreeNode** and **DataPtr** can be viewed as a single array, with the ability to discard the first array when needed. A node on this tree can be accessed directly using **DataPtr**. This means that the tree can be accessed by using an index (through the **DataPtr**) or by searching the URL (through **TreeNode**) liked a binary search tree.

The overhead for storing a single URL is equal to 18 bytes (8 bytes for a couple of left and right node ID in **TreeNode**, 6 bytes for the **CompressedURL** excluding and 4 bytes for the **DataPtr**). As , the overhead can be reduced to 10 **TreeNode** is discarded.

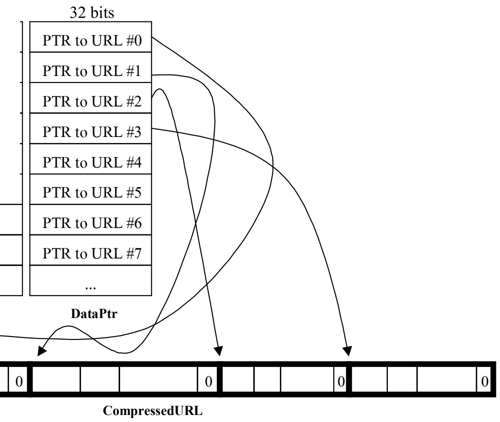

Figure 6 Data Structure of an AVL Tree for URLs Compression

4. Results

The prototype of this algorithm has been tested on a Pentium III 800 MHz machine, with 512 MBytes main memory. The test data set has been taken from NontriSearch[4] project. This data contains about 1.3 million unique URLs. The average size of URLs is about 55 bytes.

Size of a compressed URL is about 28 bytes by averaged, including all of the overhead, or about 20 bytes when the **TreeNode** is discarded. So the actual compressed data size, excluding all overhead is just only 10 bytes. This yields about 50% reduction in size for web spiders' purpose with an ability to store and retrieve the URLs on the fly. It yields about 64% size reduction for search engines, where only the ability to retrieve the URLs is required.

The access time is fast for both storing and retrieving. It takes about 92 microsecond by average to compress and store a URL (Min = 10 µs, Max = 400 µs as shown in Figure 7). Retrieving is much faster. It takes only 6 microsecond by averaged to retrieve a full URL from any given URL's ID. (Min = 2 µs, Max = 188 µs)

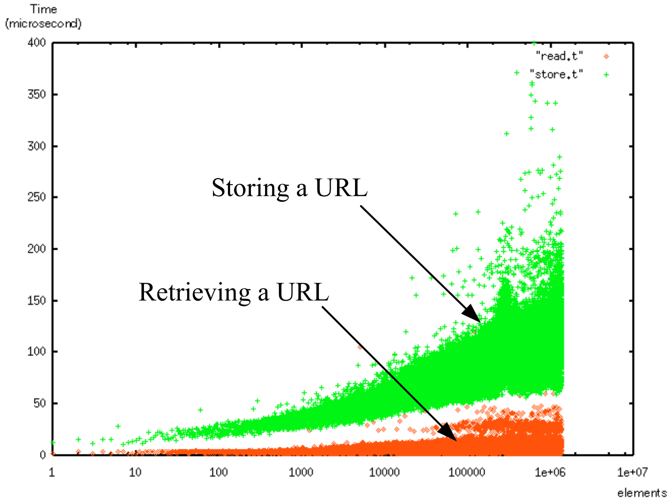

Figure 7 Access time

5. Conclusion

The AVL Tree can be used to compress URLs for web spiders or search engines. It can be constructed incrementally, which is suitable for web. Compressed URL can be efficiently searched using a general binary search algorithm, and can be accessed randomly with an index. The access time for both storing and retrieving is fast. The compression level is about 50%, including all overhead. When the tree structure is not needed, compression can be achieve up to 64%.

6. References

[1] M. Burner. *Crawling towards Eternity: Building an archive of the World Wide Web*, Web Techniques Magazine, May 1997.

[2] B. S. Michel, K. Nikoloudakis, P. Reiher, and L. Zhang. *URL Forwarding and Compression in Adaptive Web Caching* In Proc. of IEEE INFOCOMM 2000, volume 2, pages 670-678, Tel Aviv, Israel, March 2000.

[3] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian, *The Connectivity Server: fast access to linkage information on the Web*, In Proc. of the Seventh International World Wide Web Conference, Brisbane, Australia, April 14-18, 1998.

[4] S. Sanguanpong, and S. Warangrit. *NontriSearch: A Search Engine for Campus Networks*, National Computer Science and Engineering Conference (NCSEC98), Bangkok, December 1998.

[5] S. Sanguanpong, P. Piamsa-nga, S. Keretho, Y. Poovarawan and S. Warangrit**.** *Measuring and Analysis of the Thai World Wide Web.* Proceeding of the Asia Pacific Advance Network, pp225-230, Beijing, August 2000.

[6] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures in Pascal and C.* Addison-Wesley Publishing, 1991.