Pre-compression algorithm for link structure of the web

Alireza Mahdian
Computer Engineering Department,
Sharif University of Technology,
Tehran, Iran
mahdian@ieee.org

Hamid Khalili
Computer Engineering Department,
Sharif University of Technology,
Tehran, Iran
khalili@ce.sharif.edu

Mohammad Ghodsi
Computer Engineering Department,
Sharif University of Technology,
Tehran, Iran
ghodsi@sharif.edu

Abstract

We introduce a new algorithm for compressing the link structure of the web graph by means of re-indexing the nodes in some web communities in order to decrease the differences between the numerical values of indices of these nodes. This is especially done for the nodes that participate in the adjacency lists of many nodes. Our algorithm will then partition these nodes into groups, each represented by a group-indicator node. We then remove the edges directed to nodes in one group and replace them with an edge to the group indicator. This algorithm preserves the overall characteristics of the graph and also increases similarity between link adjacency lists of nodes. So it can be used as a preprocessing algorithm to compress the web graph prior to compression by Huffman or other algorithms, as a result the final compression ratio is considerably improved. We will show in the paper that the time complexity of our compression algorithm is $O(n^2 \log n)$ for each community.

Keywords:

web graph, compression, Huffman, web community

1 INTRODUCTION

The World Wide Web can be thought of as a web graph with pages being the nodes and hyperlinks being the edges. This representation of the web graph has been very useful in developing recent internet related algorithms. But, with ever increasing rate in the growth of WWW, it is no longer possible to have the whole web graph in the main memory of a computer. With billions of pages and billions of links, it is essential to ask if we can make a better (meaning faster) use of the web graph. This is where compression algorithms come to light. Compressing the web graph with the aim of moving as much of the web graph into the main memory of the computer will decrease the running time of internet algorithms. This is done by removing the unnecessary and time consuming I/O interactions which are needed to have the required part of the web graph in the main memory.

In this paper we focus on problem of compressing the web graph.

- We will introduce a new algorithm for compressing the structure of the web graph by removing edges in a reversible manner.
- We demonstrate the effectiveness of our approach on a test bed of random graphs derived from the random graph model with embedded communities introduced in [8].
- We will also show that the compression ratio of Huffman algorithm on the web graph processed by our algorithm is significantly increased.

The structure of this paper is as follows: Section 2 provides a history of works on web graph compression. In Section 3 we will introduce our algorithm, followed by experimental results. We will conclude our work with some suggestions for future works.

2 PREVIOUS RESULTS

The pervious efforts on web graph compression can be divided into two general categories: Huffman based methods referred to as traditional methods, and new methods based on the nature of web graph. Huffman based methods will compress the web graph by giving smaller codes to nodes with higher in-degrees. The new methods, however, benefit from the features of web graph and result in a better compression ratio.

Web graph has two important characteristics. First, it is a sparse graph with dense sub-graphs in some parts. The dense sub-graphs, or *communities*, have been the subject of research in recent years. Although much effort has been focused on detecting the web communities, the running time of these algorithms is still quite high. The other characteristic of the web graph is the similarity of the link adjacency list of many nodes which is the result of mirror pages on the internet.

In [1] an algorithm is introduced for compressing the web graph which has three basic steps: finding nodes with partial similarities in their adjacency lists, selecting on of them as a reference node and replacing the other nodes by their differences from the reference node. This approach results in a greater compression ratio compared to Huffman based schemes. Our algorithm will benefit from this idea in a somewhat different aspect. Unlike [1] our algorithm will compress the web graph by removing some of the edges from the adjacency list of the nodes.

A more recent contribution to this subject makes use of clustering algorithms for re-indexing the nodes with the aim of representing the link adjacency list of each node by differences between the consecutive indices of out-degree nodes [2]. Our work has the same goal but with a different approach. Unlike [2] we will work on parts of the web graph which are dense. In particular, our algorithm will make use of communities as dense sub-graphs of the web and tries to re-index the participating nodes in the community with the intention of decreasing the differences between the indices of those nodes which participate in the adjacency list of so many nodes.

3 OUR ALGORITHM

In this section we introduce our new algorithm that compresses the web graph by re-indexing the web pages within each community and then eliminating some of the edges in the link adjacency list of the nodes. Working on communities has two distinct advantages:

1. It makes possible to run the algorithm for each community on an independent processor. This means that the total running time of our algorithm can be highly reduced by running the re-indexing part, which has a high running time complexity, on parallel processors hence dividing this work among processors.
2. Since a very small fraction of the total nodes participate in a community we can bring the whole link structure of a community to the main memory and so the running time will be much faster due to the elimination of many I/O tasks.

Our algorithm has three basic steps:

1. Finding the communities with desirable density: in this step, we use [4] which is a recent algorithm for detecting the web communities and runs in $O(mm_c)$ where m is the number of edges in the web graph and m_c is the number of edges in a community c. Since community detection can be thought of as a multi-level task, we will continue the process of breaking each detected community into sub-communities until a decent level of density[1] is reached. The threshold for the value of density is based on experimental results and is a value around 1 percent.
2. Re-indexing each node in every community: This step is the main step of our algorithm. In this step we try to re-index every node of the community so that at the end of this step every node in the community points to (has out edges to) nodes with much less difference in their indices than it initially pointed to. We call this procedure The Re-indexing Algorithm and will further explain this procedure in the following section.
3. Eliminating some of the edges: in this step we will assign nodes based on their numerical index value into dynamic length groups (with 8, 16, 24, 32 members) and with the occurrence of several edges that point to nodes from the same group in the link adjacency list of a particular node, we will replace all of those edges with one edge that points to a particular node of that group (group indicator). In order to keep track of those eliminated edges we use an auxiliary data structure so that we can de-compress the web graph at later time. We will explain this, in more detail in section 3.2.

3.1 Re-indexing Algorithm

The re-indexing algorithm is our main contribution in this paper. This algorithm is designed on the idea of finding common edges between any two link adjacency lists of participating nodes in a community and trying to give new index values to those nodes (pointed to by the common edges) so that there is a high probability that the difference between the new indices of nodes in the list of common edges for any randomly selected nodes is less than what it was before.

[1]Density of a community has several definitions, but we use the ratio of average out-degree of nodes to the total number of nodes in a detected community as a measure of density for that community.

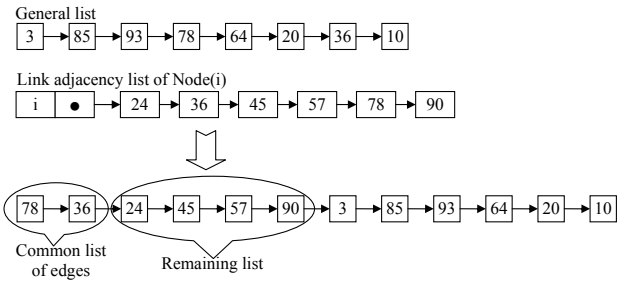

Figure 1: Effect of adding the link adjacency list of a new node to the general list

Our algorithm is a greedy algorithm which means we choose the best strategy at every step, so we do not necessarily come to the optimal result, but it can be proven that our result is necessarily better than the initial state.

We use an initially empty list for keeping the final order of the nodes. At the end of our algorithm the nodes in a community will be indexed based on their order in this list. We call this list the general list. It is obvious that this list should not have any repeated members. At each step we add the adjacency list of one node to the general list and this will be done for every node that participates in the community.

Considering the process of adding link adjacency list of Node(i) to the general list, one of the two situations will occur.

- There is no common nodes between the general list and the link adjacency list of Node(i): we simply add the link adjacency list of Node(i) at the start or end of the general list.
- There is a common list of nodes between the general list and the link adjacency list of Node(i): we move the common list with respect to ordering of its nodes in the general list to the start or end of general list and we add the remaining nodes (pointed to by out-edges) right after the common nodes at the start or end of the list. With this procedure, those nodes which are ordered based on the link adjacency list of previously added nodes will not be affected.

Since the nature of our algorithm is greedy and we choose the best option at each step, and since we deal with one node at each step, it is clear that the last nodes have more effect on the final result. So we sort the nodes in the increasing order of their out-degrees and add the nodes in this sorted list one at a time to the general list in order to come up with the best result. Figure 1 shows the effect of adding Node(i) to the general list and Table 1 presents the pseudo code for this algorithm.

3.2 Edge Eliminator Algorithm

We first define *group indicator* of a group of nodes as the first node member of that group. The basic idea in edge eliminator algorithm is to eliminate all of the edges in the link adjacency list of a node that point to nodes from the same group and instead adding an edge to the link adjacency list that points to the group indicator. It is obvious that with this procedure the overall similarity of the link adjacency list of nodes will increase.

Table 1: Pseudo code for Re-indexing algorithm

```
REINDEXING ALGORITHM()
    1   REINDEX-NODES(community(j))
    2   for each node(i) in community(j)
    3       do common ← FIND-COMMON(node(i).adjacency,list)
    4          list ← common + (node(i).adjacency - common)
                      + (list - common)
    5
    6   replace-map ← NUMBERING(list, community(j))
    7   Return replace-map
```

As was previously mentioned, we need to keep track of those edges that have been eliminated, in order to make the process of compression reversible. Since we want to make the best use of space for keeping these eliminated edges, we will assign nodes to a dynamic group (groups of 8, 16, 24, 32 members). The group number for each node consists of two parts. The first part is the main group number calculated by a simple formula: $\lceil \frac{n}{k} \rceil$ where n is the index of the node and k is the length of the largest group which should be a power of two (in our work we used 32). The second part is a number that assigns a node to a part of the main group. This number is calculated by the following formula:

$$\left(\left\lceil \frac{n_c \bmod k}{k'} \right\rceil - 1 \right)$$

where k' is the minimum length of a group (in our work we used 8). Now if a node has index value of 129 then it belongs to sub-group 0 within group 5.

The index value of the group indicator is calculated by $n_{i,1} = (i-1) \times k+1$ where i is the main group number. As the above formula implies, the group indicator of a particular group is actually the first member of that group.

We should keep in mind that the edges in the link adjacency list of each node are sorted in increasing order based on the index value of the nodes that they are pointing to. This will reduce the running time of the process of finding edges that point to node members belonging to the same main group in the adjacency list of a node. In the eliminating process of edges, on finding the first edge that points to a node from group i (meaning finding the lowest found index from group i) we look for all the edges that point to nodes from group i. Since the edges are in sorted order this process is really a straightforward procedure. We call the first node, Node(f) and the last node, Node(l). If both Node(f) and Node(l) belong to the same sub-group then we will use k' bits (in our work k' equals to 8) for representing the occurrence of each eliminated node within a bit pattern. If Node(f) and Node(l) belong to two different subgroups then we will use $((SGN(Node(l)) - SGN(Node(f)) + 1)k'$ bits for the bit pattern (where $SGN(Node(l))$ and $SGN(Node(f))$ are sub-group numbers for Node(l) and Node(f) respectively). Besides the bit patterns, we also need to keep the subgroup number as well as a group type so that we can exactly pinpoint those edges that have been eliminated. Group type indicates the length of the group. In our work we used two bits for the sub-group number (00 = subgroup 1, 01 = subgroup 2, ...) and also two bits for group type (00 = 8 members, 01 = 16 members, 10 = 24 members, 11 = 32 members). At each step of the elimination process we find and eliminate all the edges that belong to the main group i and we place an edge pointing to group indicator $n_{i,1}$ instead. We also keep track of the eliminated nodes in an auxiliary structure which holds the group type, subgroup number and also the bit patterns for eliminated edges. We will further compress the auxiliary data structure by running a Huffman-based algorithm on it.

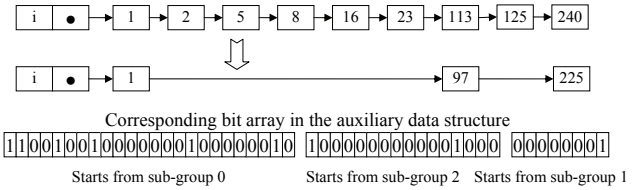

Figure 2: Demonstration of how elimination of edges is performed.

At the decompression process, for each edge in the link adjacency list, which would definitely point to a group indicator since each node belongs to a unique group, we find the corresponding group. Also we extract the corresponding group number, group type and bit array from the auxiliary data structure. Next we replace the node indicator with edges that point to nodes corresponding to the extracted bit array. Figure 2 visualizes the process of edge elimination.

4 COMPUTATIONAL COMPLEXITY AND SPACE REQUIREMENTS

Our algorithm has three different stages and each stage has its own computational complexity. Since each step is performed in series, the total computational complexity of our algorithm is equal to sum of the computational complexities at each stage.

At the first stage (which we refer to as the community detection stage) we use one of the most efficient algorithms until today. This algorithm which is introduced in [4] has a computational complexity of $O(mm_c)$ where m is the number of edges in the web graph and m_c is the number of edges in the detected community.

At the re-indexing stage, three different tasks are performed.

- Sorting the participating nodes in a community based on the order of their out-degree: using one of the fast sorting algorithms like quick sort for this stage will result in a computational complexity of $O(n_c \log n_c)$ magnitude (n_c is the number of nodes in the detected community).
- Sorting link adjacency list of each node based on the index value of pointed node: using a fast algorithm, the computational complexity of this task is equal to $O(\sum_{i=1}^{n_c} m_{c,i} \log m_{c,i})$ (where $m_{c,i}$ is the out-degree for node i from community c). The upper bound for the above series is $O(n_c^2 \log n_c)$ based on the following conclusions:

$$\begin{aligned} m_{c,max} &= \max_{i=1}^{n_c}\{m_{c,i}\} \\ \sum_{i=1}^{n_c} m_{c,i} \log m_{c,i} &\leq \sum_{i=1}^{n_c} m_{c,max} \log m_{c,max} \\ &= \\ n_c \times m_{c,max} \log m_{c,max} &\leq n_c^2 \log n_c \end{aligned}$$

- Adding a new link adjacency list to the general list: since this task is required to find the common set between the general list and the current link adjacency list and because unlike the adjacency list, general list is not sorted in any order; the computational complexity of this task is $O(n_c^2 \log n_c)$ which is derived from the following inequalities:

$$\sum_{i=1}^{n_c} l_{c,i} \log m_{c,i} \leq n_c \sum_{i=1}^{n_c} \log m_{c,i} = n_c^2 \log n_c$$

(Where $l_{c,i}$ is the general list length before the addition of link adjacency list of the ith node to the general list)

From what we mentioned, we can conclude that the running time of the re-indexing stage is: $O(mm_c + n_c^2 \log n_c)$. The edge elimination stage is basically a straightforward process of moving through each link adjacency list and eliminating all the edges from the same group and inserting an edge pointing to the group indicator instead. This task has a time complexity of $O(m)$ magnitude. The time consuming task in this stage is the Huffman coding of the auxiliary data structure which is of $O(m \log m)$ magnitude.

Addition of the time complexity of each stage, results in the total time complexity of our algorithm which is: $O(\sum_{c1}^{w}(mm_c + n_c^2 \log n_c) + m \log m)$ Where w is the total number of detected communities.

The maximum amount of main memory space required for our algorithm is proportional to the largest number of nodes participating in a detected community.

Table 2: Characteristics of generated communities

Name	Total number of pages	Average degree
Companies	100000	486
	50000	487
	10000	380
	1000	167
Newspapers	100000	285
	50000	310
	10000	306
	1000	132
Universities	100000	1525
	50000	1447
	10000	1135
	1000	349
Scientists	100000	14
	50000	14
	10000	15
	1000	3
Web out links	100000	16
	50000	16
	10000	16
	1000	4

5 EXPERIMENTAL RESULTS

We present the results from a preliminary prototype running on a generated random graph model with embedded communities. We emphasize that these experiments are meant as a preliminary proof of concept. The random graph which we have used is introduced

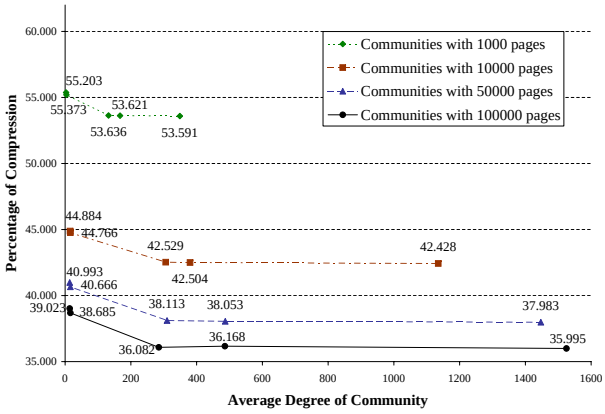

Figure 3: Effect of Huffman compression on community graph.

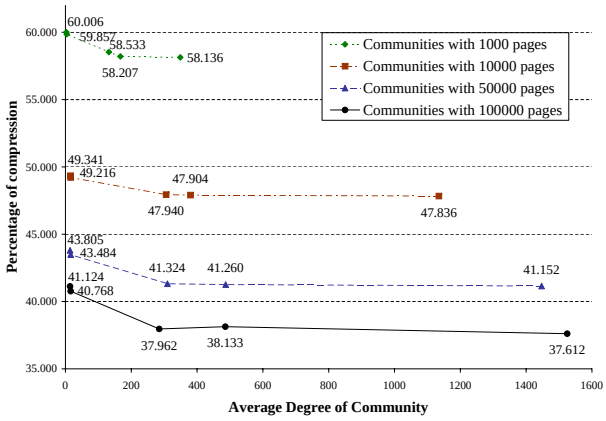

Figure 4: Effect of Huffman compression on pre-compressed community graph.

in [8]. Since the community detection algorithms are not our contributions and we only use them as a tool, we decided to directly run our algorithm for several different communities based on [16] instead of running the algorithm for a web graph with embedded communities. This will eliminate the time consuming process of community detection and will not have any effect on our demonstration. Table 2 shows the characteristics of each generated community.

Since we intended to demonstrate that our algorithm does not deteriorate compression ratio of Huffman-based schemes, we decided to show that the Huffman coding will have at least the same effect on the result of our work and on the uncompressed graph. In one experiment we compressed the community graphs with both our algorithm and Huffman coding and in a second experiment we compressed the community graph with only the Huffman coding. As shown in figure 3 and 4, the curves of Huffman coding compression ratios have the same shape for both experiments which means that no matter if we compress the community graph with our algorithm or not, the final compression ratio will be at least the same. This confirms our claim that our algorithm does not deteriorate compression ratio of Huffman based schemes (see figure 3 and 4). An important variable that has a large impact on the performance of our algorithm is the density of the detected community. Figure 5 shows this fact that the ratio of compression has a direct relation with the density of the detected community. This

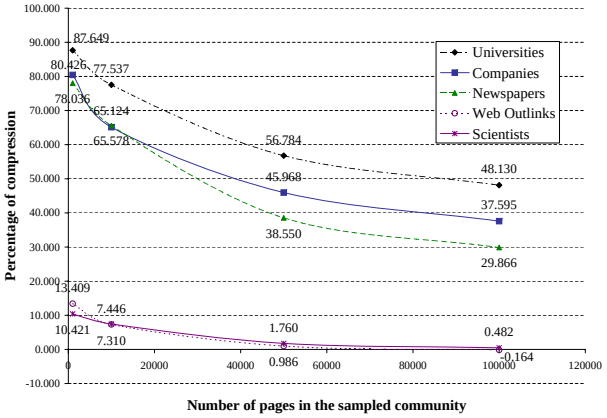

Figure 5: Comparison of compression ratio for different communities.

means as the density of detected community decreases, i.e. the community becomes sparser, the ratio of compression will also decrease (see figure 5).

6 CONCLUSIONS

Our algorithm has several distinct advantages:

- It can be run on parallel processors and this will reduce the total running time considerably.
- The space requirement of our algorithm is so little that it can be run on an ordinary desktop.
- Our algorithm will increase the similarity of link adjacency lists enormously, so algorithms like the one introduced in [1] which are based on finding similar nodes will benefit very much from the result of our algorithm.
- As stated before, Huffman algorithms can further compress the output of our algorithm, resulting in a higher total compression ratio.

We are currently working on the problem of re-compressing the result of our algorithm based on the algorithm introduced in [1]. Future efforts might lead to the elimination of the time consuming stage of community detection and running the Re-indexing stage directly on the Web Graph itself. We also appreciate any efforts to execute our algorithm on real data samples from the web graph which we hope will further confirm our results.

7 ACKNOWLEDGEMENT

The authors would like to thank Ehsan Nourbakhsh for his help in preparation of this paper.

References

[1] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In Proc. of IEEE Data Compression Conference (DCC), Mar. 2001.

[2] D. Blandford and G. Blelloch. Index compression through document reordering. In Proc. of IEEE Data Compression Conference (DCC), Jan. 2002.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to algorithms. MIT Press and McGraw-Hill Book Company.

[4] G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In Proc. 6th Int. Conf. on Knowledge Discovery and Data Mining, pages 150-160, 2000.

[5] G. W. Flake, S. Lawrence, C. Lee Giles, and F. M. Coetzee. Self-organization and identification of Web communities. IEEE Computer, 35(3):66-71, 2002.

[6] L. R. Ford Jr. and D. R. Fulkerson. Maximum flow through a network. Canadian J. Math., 8:399-404, 1958.

[7] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In Proc. 9th ACM Conference on Hypertext and Hypermedia, 1998.

[8] E. J. Glover, T. Oates and V. B. Tawde. Generating web graphs with embedded communities. ACM WWW Conference, May. 2004.

[9] M. Kitsuregawa, P. K. Reddy. An approach to relate the web communities through bipartite graphs. IEEE Second International Conference on Web Information Systems Engineering (WISE). Dec. 2001.

[10] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. In Proc. of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 1998.

[11] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan and A. S. Tomkins. The web as a graph: measurements, models, and methods, In Proc. Intl. Conf. on Combinatorics and Computing, 1999.

[12] J. M. Kleinberg, S. Lawrence. The structure of the web. Science Mag (VOL 294), Nov. 2001.

[13] R. Kumar, P. Raghavan, S. Rajagopalan, D Sivakumar, A. Tomkins and E. Upfal. Stochastic models for the web graph. In Proc. of the 41st IEEE Symp. on Foundations of Computer Science. Nov. 2000.

[14] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. WWW8 / Computer Networks, 31(11-16):1481-1493, 1999.

[15] L. Laura, S. Leonardi, G. Caldarelli, P. De Los Rios. A Multi-layer model for the web graph. Apr. 2002.

[16] D. M. Pennock, G. W. Flake, S. Lawrence, C. L. Giles, and E. J. Glover. Winners don't take all: Characterizing the competition for links on the Web. Proc. of the National Academy of Sciences, 2002.

[17] S. Raghavan, H. Garcia Molina. Representing Web Graphs. 2003.