

A Scalable Pattern Mining Approach to Web Graph Compression with Communities

Gregory Buehrer
The Ohio State University
390 Dreese Labs
Columbus, OH 43210
buehrer@cse.ohio-state.edu

Kumar Chellapilla
Microsoft Live Labs
One Microsoft Way
Redmond, WA 98052
kumarc@microsoft.com

ABSTRACT

A link server is a system designed to support efficient implementations of graph computations on the web graph. In this work, we present a compression scheme for the web graph specifically designed to accommodate community queries and other random access algorithms on link servers. We use a frequent pattern mining approach to extract meaningful connectivity formations. Our *Virtual Node Miner* achieves graph compression without sacrificing random access by generating virtual nodes from frequent itemsets in vertex adjacency lists. The mining phase guarantees scalability by bounding the pattern mining complexity to $O(E \log E)$. We facilitate global mining, relaxing the requirement for the graph to be sorted by URL, enabling discovery for both inter-domain as well as intra-domain patterns. As a consequence, the approach allows incremental graph updates. Further, it not only facilitates but can also expedite graph computations such as PageRank and local random walks by implementing them directly on the compressed graph. We demonstrate the effectiveness of the proposed approach on several publicly available large web graph data sets. Experimental results indicate that the proposed algorithm achieves a 10- to 15-fold compression on most real word web graph data sets.

Categories and Subject Descriptors

H.3.3 [Information Systems]: Information Search and Retrieval; H.2.8 [Database Management]: Database Applications—*Data Mining*

General Terms

Algorithms, Performance

Keywords

Webgraph Compression, Link Analysis, Log-linear mining

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WSDM'08, February 11–12, 2008, Palo Alto, California, USA.
Copyright 2008 ACM 978-1-59593-927-9/08/0002 ...\$5.00.

1. INTRODUCTION

Conservative estimates place the size of the web to be 11.5 billion pages¹, and more than 300 billion links. More aggressive estimates suggest a number closer to 30 billion pages². The scale is so large that many engineering techniques simply are not effective. Significant effort has been expended to develop information retrieval algorithms which scale to these proportions, tackling challenges such as deep page indexing, rapid user response times, link graph construction and querying, and efficient storage. The work we present here targets the latter two issues.

The connectivity of the web has been shown to contain semantic correlation amongst web entities [16]. For example, if many pages with similar hyperlink text point to the same page, it can be inferred that page contains information pertinent to the text. Algorithms such as HITS [22] and PageRank [7] formalized this notion. Gibson, Kleinberg and Raghavan [17] first suggested that the web was composed of fairly organized social structures. Naturally, the search community has developed dedicated systems to allow algorithm designers to query the web's structure, called *Connectivity* or *Link Servers*. Connected components in the structure, such as dense bipartite graphs, k-core sets or cliques can be inspected for interesting correlations. Search engine optimizers (SEOs) are known to exploit this fact by injecting additional, less-useful links to improve relevance rankings, a practice commonly referred to as *link spam*. Efficient community discovery via connectivity servers has been shown to improve detection of such spam [18].

We can generally classify link queries (or portions thereof) as either streaming or random access queries. Streaming queries touch most or every vertex in the graph in a predictable access pattern, while random access queries have unpredictable access patterns. For example, *PageRank* is typically implemented with a data-push model, where reads are streaming and writes are somewhat random. The cost of the random writes can be mitigated by collecting them locally and then transferring them in a single synchronization per iteration. However, many interesting questions are more efficiently answered by random access algorithms. Example queries include *How many hops from page X is page Y?*, *Is page Y a member of a close-knit community of pages?*, and *Do we think X could be link spam?* If the graph does not fit in main memory, frequent random seeks to disk can render the system useless, since hard disk access is five orders of magnitude slower than access to RAM. Therefore, signifi-

¹<http://www.cs.uiowa.edu/~asignori/web-size/>

²<http://www.pandia.com/sew/383-web-size.html>

cant research has focused on compressing the link structure of the web.

Existing strategies leverage the fact that if the vertices in the graph are ordered by their corresponding URLs, then consecutively ordered vertices have significant overlap in outlinks. In addition, it is common for the outlinks (in sorted order) to have small differences in destination Id (due to the ordering). This can be exploited by encoding the difference in successive Ids instead of the Id itself, a process called *gap coding* [25]. However, global ordering of URLs is fairly expensive, both because URLs average 80 bytes, and because the web graph is updated with new URLs by crawlers at least daily. The compression technique we illustrate does not depend on any ordering or labeling of the source data, and can easily be compressed in chunks.

The goal of knowledge discovery and data mining (KDD) is to extract useful information from data sources, often in the form of recurring patterns. While not completely ignored, frequent pattern mining has not been significantly leveraged to address web search challenges in the literature. The most likely cause is that pattern enumeration is quite expensive in terms of compute time. In this work, we leverage pattern mining to compress the web graph. We introduce an effective itemset mining algorithm that maintains scalability for large data via an $O(D \log D)$ complexity constraint. We believe it is the first itemset miner to have such a constraint. This restriction is significant, as the number of mining function calls is in the order of millions per graph.

It is our observation that rather than reference coding the graph, we can use community structure to compress it. It has the two-fold benefit of both providing high compression ratios, as well as affording constant-time access to dense bipartite subgraphs, which can then be used as seeds in the community discovery process for a given URL. To accomplish this, we cast the outlinks (or inlinks) of each vertex as a transaction, or itemset, and mine for frequent subsets. We find recurring patterns, and generate a new vertex in the graph called a *Virtual Node*. The virtual node has the outlinks of the target pattern. We then remove the links from the vertices where the pattern was found, and add the virtual node as an outlink. In many cases, we can represent thousands of edges with a single link to a virtual node. In addition, the pattern is a directed bipartite clique, which often has semantic meaning. The reason is that it is common for two competing companies to not link to each other, but third parties will often link to both, thus detecting inter-domain patterns [23]. Also, bipartite graphs within a domain often depict a web template pattern, such as navigation bars. For large, multi-domain hosts whose URLs do not share a common prefix, these template patterns can help to classify the page. Finally, community discovery algorithms can use virtual nodes as seeds to grow communities.

Specifically, the contribution of this work is a web graph compression mechanism with the following properties.

- The compression ratio is high, comparing favorably with other compression techniques in the literature.
- It potentially aids community discovery by providing connected bipartite graphs as seeds in $O(1)$ time.
- The compression supports random access to vertex lists without decompressing³.

³Enumerating list items then requires decompression

- It does not require sorted and adjacently labeled vertices, such as by URL, and thus supports incremental updates easily.
- It is highly scalable, capable of compressing a 3 billion edge graph in 2.5 hours on a single machine.
- It is flexible, clearly divided into two phases; algorithm designers are free to incorporate alternate algorithms easily.

2. BACKGROUND

We present relevant background information for the interested reader. Related research is presented in Section 6.1.

2.1 Connectivity Servers

There are several fundamental challenges to address when designing a link server. Two of the primary challenges are the size of the graph and the ability to support random walk queries.

In the graph representation, each page is represented as a vertex in the graph; hyperlinks are directed edges. Without compression, each link requires $\log(|V|)$ space, or 36 bits, thus the graph requires about 2 terabytes. For computations which access every edge, such as static rank, the graph can be stored on disk with a reasonable penalty to computation costs. For search queries, seeking on disk is cost-prohibitive, so it is desirable to store the 2TB in RAM. Most commodity servers house 4 to 16GB of RAM, suggesting a need of 128 to 500 machines. Compression can reduce the number of machines significantly.

The outlinks (and possibly the inlinks) of each vertex are stored serially in RAM. An additional structure is kept which stores V offsets, one for each vertex's first link. We point out that this structure requires approximately $O(V \log E)$ bits, or if compressed, $O(V \log \text{comp}(E))$ bits, where $\text{comp}(E)$ is the total cost for all the compressed links. Each halving of the bits per link value only reduces the cost of the offset structure by a small margin (for example, 1/32th). We will add to the cost of this index, because we add additional vertices to the graph. However, on average we add only about 20% more vertices, and does not change the size significantly (see Section 4.2). The majority of the web graph compression literature report bits-per-link values. We will do the same, to provide for comparisons. One can see that as we compress the edges, the roughly constant size of the offset structure becomes the bottleneck to lowering the total storage costs.

In addition, if reverse indexing of the actual URLs is desired, this will incur a fixed cost as well. Typically, the URL table is queried only at the conclusion of the computation, and only if page-level details are desired. Finally, it may be desirable to end all indexable data on a byte boundary, which will incur padding costs [6].

2.2 Frequent Itemset Mining

The solution provided uses an approximate frequent itemset mining approach to community discovery. In this section, a description of frequent itemset mining is provided. The problem was first formulated by Agrawal *et al.* [1] for association rule mining. Briefly, its description is as follows. Let $I = \{i_1, i_2, \dots, i_n\}$ be a set of n items, and let $D = \{T_1, T_2, \dots, T_m\}$ be a set of m transactions, where each

transaction T_i is a subset of I . An itemset $i \subseteq I$ of size k is known as a k -itemset. The *support* of i is $\sum_{j=1}^m [i \subseteq T_j]$, or informally speaking, the number of transactions in D that have i as a subset. The challenge is to find all $i \in D$ with *support* greater than a minimum value, *minsupp*.

All frequent pattern mining instantiations (itemsets, subtrees, subgraphs, etc.) have deterministic solutions which can be found if given sufficient compute time. However, it is clear that the number of subsets of a given set A is exponential in the cardinality of A , and in the worst case (*minsupp* = 1) these sets must be enumerated. Thus, the time required for full enumeration is exponential. Mining for other forms of itemsets, such as *closed itemsets*, *maximal*, or *non-redundant* itemsets produce potentially smaller results, but do not reduce the compute complexity.

3. VIRTUAL NODE MINER

In this section, we detail our solution, called *Virtual Node Miner*. The premise of our method is that the overlap in link structure present in the web graph can be summarized by adding a relatively small number of virtual nodes. These virtual nodes are a one-level indirection of intersecting bipartite graphs. As an example, consider the two graphs in Figure 1. The top of the figure displays a dense bipartite graph, where the source vertices (labeled S) all share a subset of destination links. By introducing an additional vertex in the graph (VN), 19 of the 30 shared links can be removed.

Algorithm 1 VNMiner

Input: A graph $G = (V, E)$
Output: A compressed graph G'

- 1: **for** $i = 0$ to NumberOfPasses **do**
- 2: $C = \text{Cluster } V$
- 3: **for all** $c \in C$ **do**
- 4: Patterns $P = \text{Find patterns in } c$
- 5: **for all** $p \in P$ **do**
- 6: Virtual Node $v = \text{elements of } p$
- 7: $G = G \cup v$
- 8: **for all** $v \in c$ **do**
- 9: **if** $p \subset \text{vertex.links}$ **then**
- 10: $\text{vertex.links} = \text{vertex.links} - p + v$
- 11: **end if**
- 12: **end for**
- 13: **end for**
- 14: **end for**
- 15: **end for**
- 16: Encode G

We cast the problem of finding common links for vertices as a frequent itemset mining challenge, where the outlinks (or inlinks) of each vertex is a transaction. The number of labels is then the number of vertices in the graph. The desired minimum support is then two, which presents a significant challenge. However, we do not require the full enumeration of frequent patterns, nor for the exact counts of each pattern. To circumvent the nearly impossible task of mining hundreds of millions of data points at once, we first cluster similar vertices in the graph. Next, we find patterns in the clusters, remove the patterns, and replace these patterns with virtual nodes. The pattern discovery process is restricted to $O(E \log E)$ time. This process can be repeated until the number of discovered patterns degrades. Each vertex (and thus each

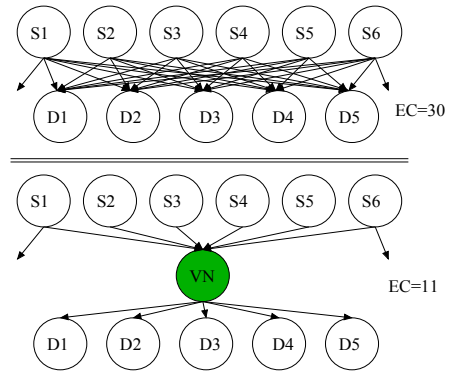


Figure 1: A bipartite graph compressed to a virtual node, removing 19/30 shared edges.

edge in the graph) is passed to the mining process at most once in each iteration. Finally, we can encode the remaining edges, using any available coding scheme.

Pseudo code outlining *Virtual Node Miner* is provided as Algorithm 1. Line 10 removes the outlinks associated with the current pattern from each vertex that contains the pattern, and adds a single outlink to the virtual node. We name the clustering process in line 2 the *Clustering Phase* and the mining process in lines 4 - 12 the *Mining Phase*. We now detail scalable solutions for these two phases.

3.1 Phase 1: Clustering

To maintain a scalable algorithm, we use probabilistic sampling on the graph. The goal is to group similar vertices together, where similarity is a function of the outlinks of the vertices in question, in $O(E \log E)$ time. For graph $G = (V, E)$ we use k min-wise independent hash functions [8] to obtain a fixed length sample for each vertex, obtaining a $k * V$ matrix. We then sort the rows of the matrix lexicographically. This sort is quite fast as it requires only $O(2V \log V)$ bits in memory at a time, which fits in RAM (or we make it by mining the graph in chunks). Next, we traverse across the matrix column-wise, grouping rows with the same value. When the total number of rows drops below a threshold, or we reach the edge of the hash matrix, we pass the vertex Ids associated with the rows to the mining process. For example, suppose in the first column there is a contiguous block of 200,000 rows with the same hash value. We then compare the second column hashes of these 200,000 rows. For each distinct hash value, we inspect the number of rows with that value. If cardinality is below the threshold, we call *MiningPhase()*; otherwise we inspect the third column hash values for the selected rows, and so on. The lexicographic sort surely biases the sampling left-wise in the matrix, but multiple iterations afford a reasonable shuffling. This probabilistic approach performs quite well in practice, grouping rows with high Jaccard coefficients.

Pseudo code for the clustering phase is provided as Algorithm 2. Lines 1 - 10 generate K minimum hashes for each vertex in the graph. Lines 12-20 scan the hash lists to generate clusters of vertices. The first column serves to roughly group the vertices, while line 17 prunes the list by comparing the current list of vertices with their next column hash values⁴. Line 19 calls the mining process.

⁴In practice, this is implemented via recursion, such that

Algorithm 2 ClusteringPhase

Input: A graph $G = (V, E)$
Input: Number of hashes K
Output: A clustered graph G'

```
1: for all  $k \in K$  do
2:   for all  $v \in V$  do
3:     Hash  $minH = HASH\_MAX$ 
4:     for all  $e \in v$  do
5:        $h = hash(e, k)$ 
6:        $minH = Min(h, minH)$ 
7:     end for
8:     Add  $minH$  to matrix  $M$  for  $v$ 
9:   end for
10: end for
11: Sort  $M$  Lexicographically (or multi-shingle)
12: for all List  $N \in HashColumn(1)$  do
13:   Column  $col = 1$ ;
14:   List  $N = GetList(N, 0)$ 
15:   while  $|N| > threshold$  and  $col < K$  do
16:      $col ++$ 
17:     List  $N = GetList(N, col)$ 
18:   end while
19:   MiningPhase( $N$ )
20: end for
```

3.2 Phase 2: Pattern Mining

The goal of the mining phase is to locate common subsets of outlinks in the given vertices. Larger sets of higher frequency are of interest, as they may represent more relevant communities. In addition, they afford better compression. Specifically, a pattern P 's compression performance follows the following formula.

$$Compression(P) = (P.frequency - 1)(P.size - 1) - 1 \quad (1)$$

The frequency of the pattern is the number of vertex adjacency lists it was found in. We subtract one from this value because we must store the list once (as the virtual node's adjacency list). The size of the pattern is the number of links in it. We one from this value because for each vertex in which we remove the list, we must add a link to the virtual node. Finally, we subtract one more from the compression value to store the index of the virtual node in the index table (every vertex in the graph contains an offset in the index which provides the starting byte for its adjacency list). In this way, the compression value for a potential virtual node incorporates the total difference between the original graph and the compressed graph.

As discussed earlier, exact itemset mining is an exponential computation. Using a simple intersection of the provided links is too conservative, as many subpatterns are unnecessarily disregarded. Therefore, we instead focus on a greedy mining process which is again bounded by $O(E \log E)$. The mechanism is to construct a trie of the vertex's edge list Ids, and then to use long paths in the tree to construct patterns for virtual nodes.

The algorithm proceeds as follows. An initial scan is performed to retrieve a histogram of the outlink Ids, and then the lists are reordered such that the most frequent outlink Ids sort first. Then each outlink list is added to a prefix tree. Every member of the set from list N is eventually passed to the mining process.

Vertex Id	Outlink List
23	1,2,3,5,6,10,12,15
55	1,2,3,5
102	1,2,3,20
204	1,7,8,9
13	1,2,3,8
64	1,2,3,5,6,10,12,15
43	1,2,3,5,6,10,22,31
431	1,2,3,5,6,10,21,31,67

Table 1: Sample vertex list passed to the mining process.

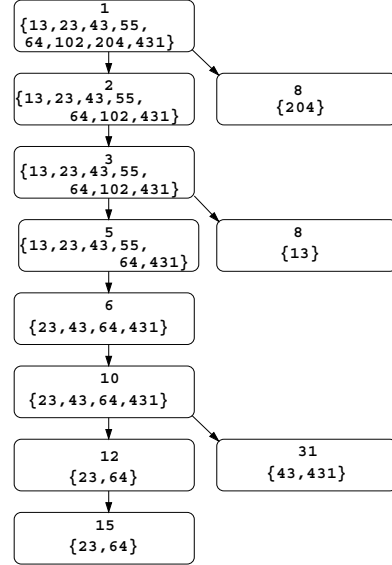


Figure 2: Example prefix tree representation based on the data in Table 1.

tree. Each node in the tree has a label and a sorted set of transactions which contained the prefix. Only items which occur at least twice are inserted into the tree. Then the tree is walked to record patterns of interest, specifically patterns which maximize Equation 1. Those patterns are then converted to virtual nodes and the vertex Ids in their lists are removed.

For example, consider the outlink list provided in Table 1. A total of eight nodes were provided. After removing singletons, the prefix tree is constructed, as shown in Figure 2.

A walk of the tree is performed, and at each leaf with a vertex list length greater than one, a *PotentialNode* object is generated. This object represents the potential for a virtual node. It contains a pointer to its associated prefix tree node, and its pattern length (depth in the tree). The prefix tree node is then colored as processed. From this node, a traversal to the root is performed. Whenever a parent node owns a longer vertex list, and it has not been colored, it is added to the list of potential virtual nodes. After the traversal completes, the potential list is sorted according to Equation 1. The list is then processed for virtual nodes. After each *PotentialNode* is processed, the associated vertices in its path up the tree are deleted. It may be the case that

Length	Frequency	Vertex List	Savings
6	4	43,431,23,64	14
3	7	23,55,102,13,64,43,431	11
8	2	23,64	6
7	2	43,431	5

Table 2: List of potential virtual nodes.

subsequent *PotentialNodes* are reduced in utility because the original ordering is greedy⁵. The actual utility is recomputed (in $O(1)$ time) before mining, and discarded if it is not fruitful. The *PotentialNodes* for the running example are presented in Table 2.

Algorithm 3 MiningPhase

Input: A graph $G = (V, E)$
Input: A set of vertices W
Output: A list of altered vertices W'

```

1: for all  $v \in W$  do
2:   for all  $e \in v$  do
3:      $Counts[e]++$ 
4:   end for
5: end for
6: Sort  $Counts$ 
7: for all  $v \in W$  do
8:   List  $L$ 
9:   for all  $e \in v$  do
10:    if  $Counter[e] > 1$  then
11:       $L += e$ 
12:    end if
13:  end for
14:  Sort  $L$  using  $Counts$ 
15:  Add  $L$  to  $T$ 
16: end for
17: Tree  $T$ 
18: List  $VN = null$ 
19: GenerateVirtualNodeList( $VN, T.Root$ )
20: Sort  $VN$ 
21: for all Pattern  $p \in VN$  do
22:   Virtual Node  $n = p$ 
23:    $G = G \cup n$ 
24:   for all  $v \in p.VertexList$  do
25:     if  $p \in v.links$  then
26:       Remove  $p$  from  $v$ 
27:       Remove  $v$  from  $p.parent$ 
28:     end if
29:   end for
30: end for

```

Pseudo code for the mining process is provided as Algorithm 3. Lines 1 - 5 generate a histogram of the edges for all the vertices passed, which is then sorted in decreasing order. Lines 7 - 16 examine each vertex, generate a transaction of its edges sorted based on the histogram, and add it to the prefix tree. The histogram ordering improves the overlap in the tree. Line 18 walks the tree to generate a list of potential virtual nodes. It is a function call to Algorithm 4, *GenerateVirtualNodeList*. Lines 19-29 generate virtual

⁵Subsequent sorting can be performed on the list but in practice we found this did not significantly change end to end compression values or execution time.

nodes and trim the edge lists of the original graph vertices. *GenerateVirtualNodeList* walks to the leaves of the tree (or to the last node in a path with a list greater than 1), and then walks back to the root. During the traversal back to the root, potential virtual nodes are generated and added to a list. The pseudo code is provided as Algorithm 4. Adding the tree node as a potential virtual node is accomplished by inserting the tree node Id into the list and then coloring the node (to avoid future processing). Further details of the mining process can be found elsewhere [10].

3.3 Complexity Analysis

The complexity of the proposed approach is of particular interest, due to the size of the problem at hand. For a graph $G = (V, E)$, the clustering phase is $O(kE \log E)$, where k is the chosen number of hashes. Each edge $e \in E$ is hashed, then for each column of data the hashes are sorted. Selecting subsets from the sorted data is efficient, requiring kV time, since at worst each hash is touched once. As k is a small constant, we remove it from the analysis.

The mining phase is also $O(E \log E)$. Since each vertex is passed once per iteration, each edge is also passed only once. The first step builds a histogram in $O(E)$ time. The histogram is then sorted. Each edge from each vertex is then added to the tree once, requiring at worst $O(\log E)$ time per edge, since the children of a tree node are a sorted set. Generating potential virtual nodes is an $O(E)$ operation, since the prefix tree can have at most E nodes, each node is visited twice, and the computation at a tree node is constant. Sorting the potential virtual nodes does not require more than $O(E \log E)$ time since its size is bounded by $O(E)$. Finally, each potential node is processed once. The processing step marks an edge in a graph vertex, removes a vertex Id from the tree, and adds a new virtual node to the graph. Each edge is marked only once, each vertex Id is removed from a list only once per edge, and there cannot be more than $O(E)$ virtual nodes generated. Therefore, the mining phase is bounded by $O(E \log E)$.

Algorithm 4 GenerateVirtualNodeList

Input: TreeNode $Root$
Input: List L of potential Virtual Nodes

```

1: for all Child  $c \in Root.children$  do
2:   if  $c.count > 1$  then
3:     while  $c.count > 1$  and  $c.child = NULL$  do
4:        $c = c.child$ 
5:     end while
6:     MarkTreeNode( $c, L$ );
7:   end if
8: end for

```

4. EMPIRICAL EVALUATION

We evaluate our algorithm empirically to gain an understanding of its performance. *Virtual Node Miner* is characterized according to the compression capabilities, the quality and quantity of the generated virtual nodes, and the time to run the process. The implementation is C++ using STL on a Windows PC running Windows Server 2003. The machine is a 2.6 GHz Dual Core AMD Opteron(tm) with 16GB of RAM. For this evaluation, only one core is used, as the implementation described in this work is serial. In all trials we use 8 hashes, as more did not improve results. Also, the

Data Set	Nodes	Edges	Edges/Node
WEBBASE2001	118,142,155	1,019,903,190	8.63
UK2002	18,520,487	298,113,762	16.09
IT2004	41,291,594	1,150,725,436	27.86
ARABIC2005	22,744,080	639,999,458	28.13
EU2005	862,664	19,235,140	22.29
SK2005	50,636,154	1,949,412,601	38.49
UK2005	39,459,925	936,364,282	23.72
UK2006	77,741,046	2,965,197,340	38.14

Table 3: Web data sets.

trials provided are for outlinks; we have investigated inlinks and observed slightly better compression values.

The data sets used are provided in Table 3. These data sets are made available to the public⁶ by the members of the *Laboratory for Web Algorithmics*⁷ at the *Univerita Degli Studi Di Milano*. Many of these web graphs were generated using *UbiCrawler* [4] by various labs in the search community. We provide compression values for all the data sets the table, however the focus is on the *UK2006* data set, as it is large and is the most recent.

4.1 Compression Evaluation

We evaluate the compression capabilities of the algorithm. In all cases we include the cost of the virtual nodes and their inlinks when reporting compression values. First we use only virtual nodes to remove edges from the graph. Figure 3 (left) illustrates the results for each of the eight data sets. The vertical axis is the ratio of total edges in the original graph divided by the edges remaining after compression (including the virtual node edges). Virtual nodes perform best on recent data sets which have a higher edge to node ratio. For example, the *UK2006* data set has more than 7 out of every 8 edges removed from the original graph for a compression of 7.1-fold. The compression decays with each pass, as fewer patterns are found. In all cases ten iterations (or passes) saturates compression.

We compared the lexicographically sorted clustering scheme with super-shingles [9] and found that the former bested the latter by 15% on average. Super-shingles are well suited for finding dense subgraphs. However, in our experiments, we found that their requirements for a match are too stringent for compression purposes. The break even point for compression to leverage similarity between two nodes is roughly two shared edges between them. These matches that are useful for compression have a much larger distance when compared with those produced using super-shingling.

In Figure 3 (right) we compress the remaining edges with Δ coding followed by Huffman coding. From the figure we can see that overall compression is quite high, up to 14.3-fold on the *UK2006* data set, and higher on others. In all cases, the maximum compression occurs within four iterations of the algorithm. This occurs because removing edges degrades the efficiency of gap coding. In Figure 4 this is illustrated for the *UK2006* and *SK2005* data sets. While virtual node compression continues to improve for several passes, gap coding techniques use more bits per edge. We implemented both Δ Huffman coding as well as ζ coding.

⁶Except for UK2006 which has yet to be posted, but is available at <http://www.yr-bcn.es/webspam/datasets/uk2006-links/>

⁷<http://law.dsi.unimi.it/>

Data Set	WebGraph	VNM	# VN
WEBBASE2001	3.07	3.01	13.3M
UK2002	2.22	1.95	4.11M
IT2004	1.99	1.67	4.48M
ARABIC2005	1.99	1.81	3.62M
EU2005	4.37	2.90	265K
SK2005	2.86	2.46	11.2M
UK2005	1.70	1.42	7.1M
UK2006	NA	1.95	17.1M

Table 4: Bits per edge values for *WebGraph* and *Virtual Node Miner (VNM)*, and the total number of virtual nodes generated (#VN).

An advantage of ζ codes is that they are flat (no need to search a tree for the code). While Huffman coding is optimal, it requires longer decoding times and also must store the decoding tree, which we truncate for efficiency. In these trials, the table was truncated.

A summary comparing *Virtual Node Miner* (labeled *VNM*) with the *WebGraph* framework is provided in Table 4. The proposed technique is quite competitive (we do not have maximum compression values for the *UK2006* data set at this time). Since these values are in bits/edge, they do not account for the cost of the offsets. We note that as the number of bits per edge decreases, offset overhead becomes significant. For example, the *IT2004* data set averages about 28 edges per node. If those edges are compressed using only 2 bits each (Table 4), then the representation requires 56 bits per node for the edges, and 26 bits per node for the offset into the edge array. Finally, the URLs are also stored, which at a 10-fold compression requires approximately 80 bits per node. Thus, link servers supporting random access and reverse URL lookup will find that *Virtual Node Miner* and *WebGraph* are comparable from a compression standpoint.

4.2 Virtual Node Evaluation

The number of virtual nodes generated to support maximum compression is reported in Table 4. On average, the number of virtual nodes added to the graph is about 20% of the original number of nodes, which does not introduce a significant additional overhead into the offset array. The *UK2006* graph required 22% of virtual nodes to reach maximum compression. These additional nodes add to the offset cost slightly, but reduce the number of total edges, as shown in the low bits/edge values. *WEBBASE2001* has a low edge to node ratio (8.63). As a result, does not compress well, and only generates 13.3M virtual nodes (11%).

Figure 5 provides insight into the size and distribution of the virtual nodes generated. The left figure displays the mean number of virtual nodes referenced by an original node. After one pass, on average each original node has 0.7 outlinks to virtual nodes. After ten passes, each original node has 1.45 outlinks to virtual nodes. Thus, even after ten passes, the average number of dereferences to virtual node lists is only 1.45, requiring approximately 600 nanoseconds. To compare, *Webgraph*'s maximum compression requires 31,000 nanoseconds. The right figure displays the distribution of the number of virtual nodes for each original node as a function of the number of passes of the algorithm. The maximum number of virtual nodes for any given original node cannot exceed the number of passes. It can be seen that the worst case number of dereferences is a small

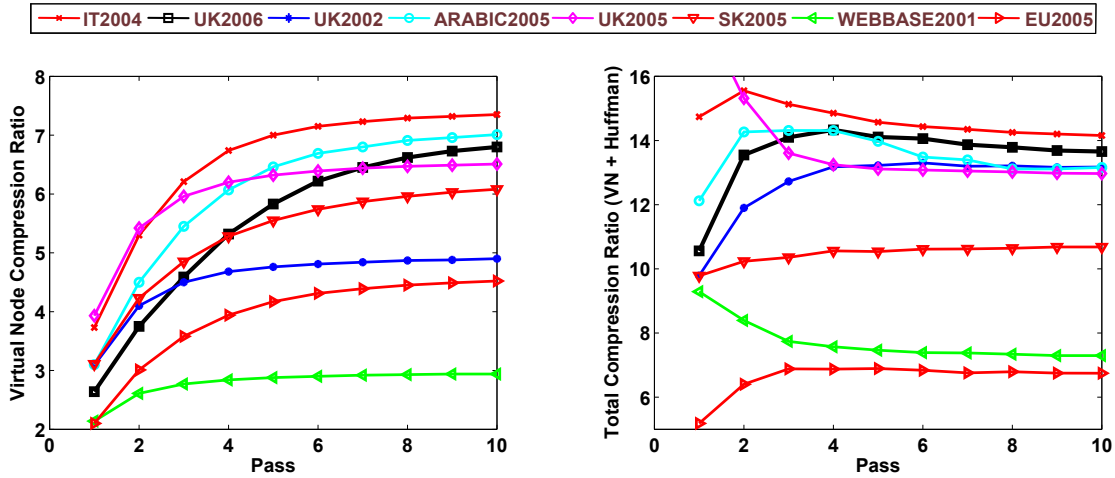


Figure 3: Compression afforded by virtual nodes (left), and total compression (right).

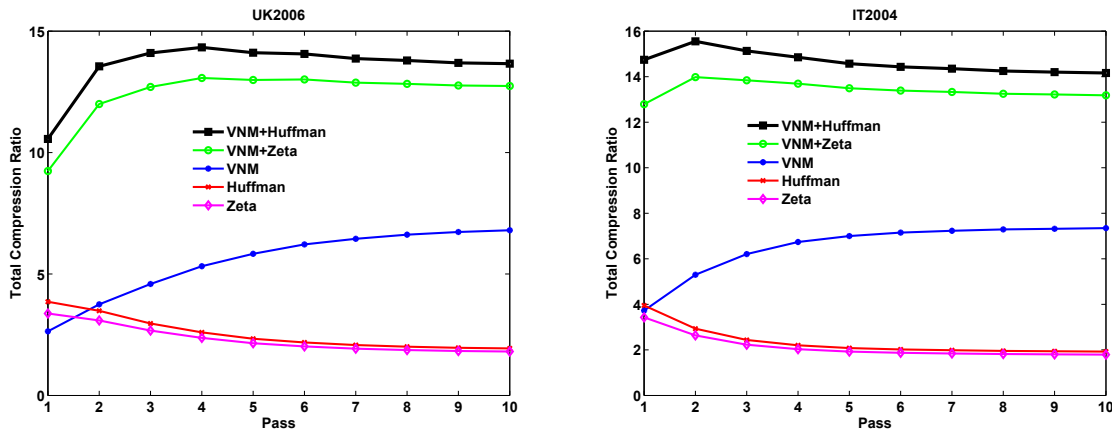


Figure 4: Compression factors for the *UK2006* data set (left), and the *IT2004* data set (right).

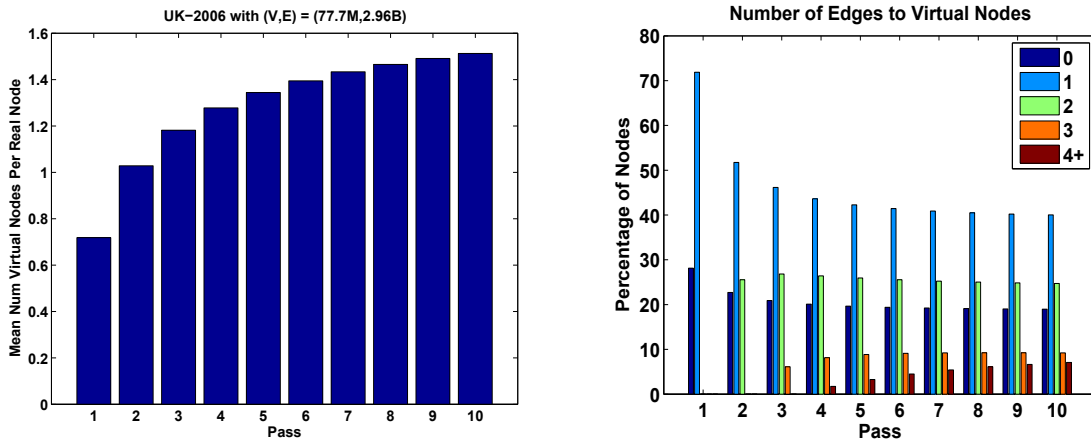


Figure 5: The number of virtual nodes per iteration (left) and the cumulative average number of virtual nodes (right).

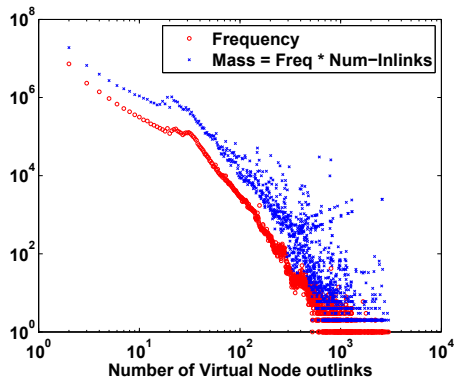
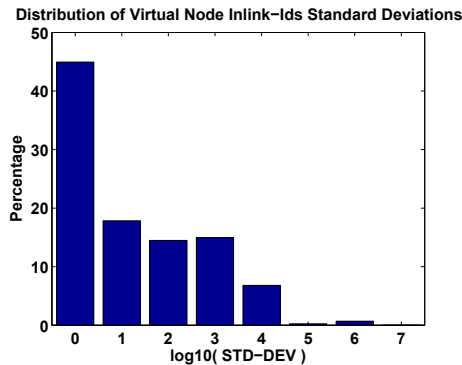


Figure 6: The standard deviation of the ranges for inlinks of virtual nodes (left), and size/mass values for the virtual nodes (right) for the *UK2006* data set.

percentage; at ten passes less than 7% have more than four virtual nodes in their outlink lists.

In Figure 6 (left), the standard deviation of the ranges of the inlinks of the virtual nodes is plotted. This plot provides a means to evaluate what percentage of the patterns captured by *Virtual Node Miner* would be captured by windowing schemes. Most windowing schemes use a window size less than ten [25]. At any window size less than ten, more than 50% of the patterns have ranges which would not be found. Figure 6 (right) plots the distribution of the mass of the virtual nodes. Mass is the number of inlinks using the pattern multiplied by the number of patterns with the associated length. The x-axis represents the length of the pattern. It can be seen that a large percentage of the compression mass occurs with long patterns.

4.3 Execution Time Evaluation

Experimental evaluation clearly demonstrates that the algorithm is scalable. As shown in Figure 7 (left), the execution time for the *UK2006* graph (which has about 3B edges) requires less than 2.5 hours. Note that the times in the vertical axis in both figures are cumulative. In this trial, 89 million separate function calls were made to the mining phase. All other data sets in this study required less compute time. For data sets which exceed main memory, the algorithm is run in batches. We have found that *Virtual Node Miner* can be run effectively on machines with 2GB of RAM. If desired, the number of passes could also be trimmed, since the compression degrades with each pass. In Figure 7 right, the graph is partitioned into blocks of 125 million edges. *Virtual Node Miner* scales linearly as the size of the graph increases, from 27 minutes on 250M edges to 110 minutes on 1.5B edges. The first pass requires more time than the others primarily because the data set is read into memory. For example, Pass 1 required 47 minutes for 1.5B edges, whereas pass 2 only required 11 minutes.

4.4 Community Seed Semantic Evaluation

Finally, we examine several of the typical community seed patterns discovered represented by virtual nodes. For this experiment, we output the members of communities whose vertex Ids flagged several simple heuristics when compressing the *UK2006* data set. We measured the size of the pattern, the number of vertices linking into the pattern, the

standard deviation of the Ids for those inlinking vertices, among other properties. Since the public web graph data is sorted by URL, virtual nodes with high standard deviations in vertex Ids represent communities whose members do not reside in the same host domain. We then generated histograms by bucketing Ids into 100 bins and visually inspected the graphs. Figure 8 depicts four example communities. Table 5 lists the starting vertex Id, ending vertex Id, and a sample URL for each cluster in each community. Each community is separated by vertex Id ranges, where applicable. The size of each community is then the sum of the sizes of its ranges. For example, community 16 consisted of 10,182 web pages.

The first image, community 11, consists of just one range of contiguous vertices, from 11,393,132 to 11,394,190 totaling 1,058 pages. It was flagged by our pattern filter because it is a rather large (over 1,000 outlinks) pattern and had many inlinks (again, over 1,000). Upon inspection it has properties consistent with link farms. This site is for "loan69". Often times the pages linking into a farm push PageRank towards advertising pages.

The second image is community 16, consisting of three distinct groups. It appears to be an online cellular equipment sales company using multiple domains – *mobilefun.co.uk* and *mobilepark.co.uk* – using common web templating. It was flagged because the three groups all have significant size and the total range is large. We found this pattern common as well. In fact, they also use *mobileguru.co.uk* which happened to sort closely to *mobilefun*.

The third image is community 31. This pattern is also from *mobilefun.co.uk* but includes a prefix *ringtones*. The site is a search engine for downloadable cellular content. The site appears to use a common technique to improve crawler interaction. Dynamic web pages are simulated in server scripts to appear to be static pages within a directory structure. For example, the local page */family-guy-Mp3-Ringtones.htm* is actually the results of a query to the search input form, which when sent to the server uses the querystring */search.php?term=family+guy+mp3&s=0*. Crawlers often downgrade querystring terms in URLs, which would have resulted in fewer instances of the *search.php* page in the index.

The fourth image is community 40. It consists of four distinct ranges. The number of vertices in each range is [651

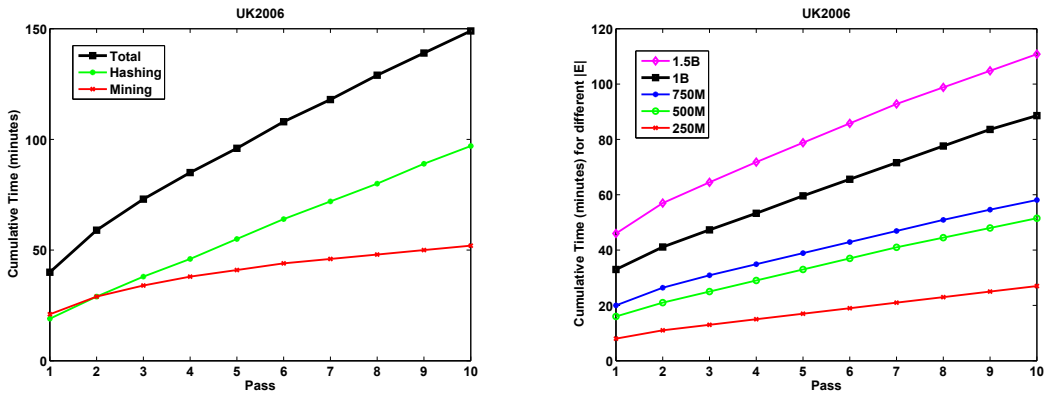


Figure 7: Execution time for each phase (left), and execution time as a function of graph size(right) for processing the *UK2006* graph.

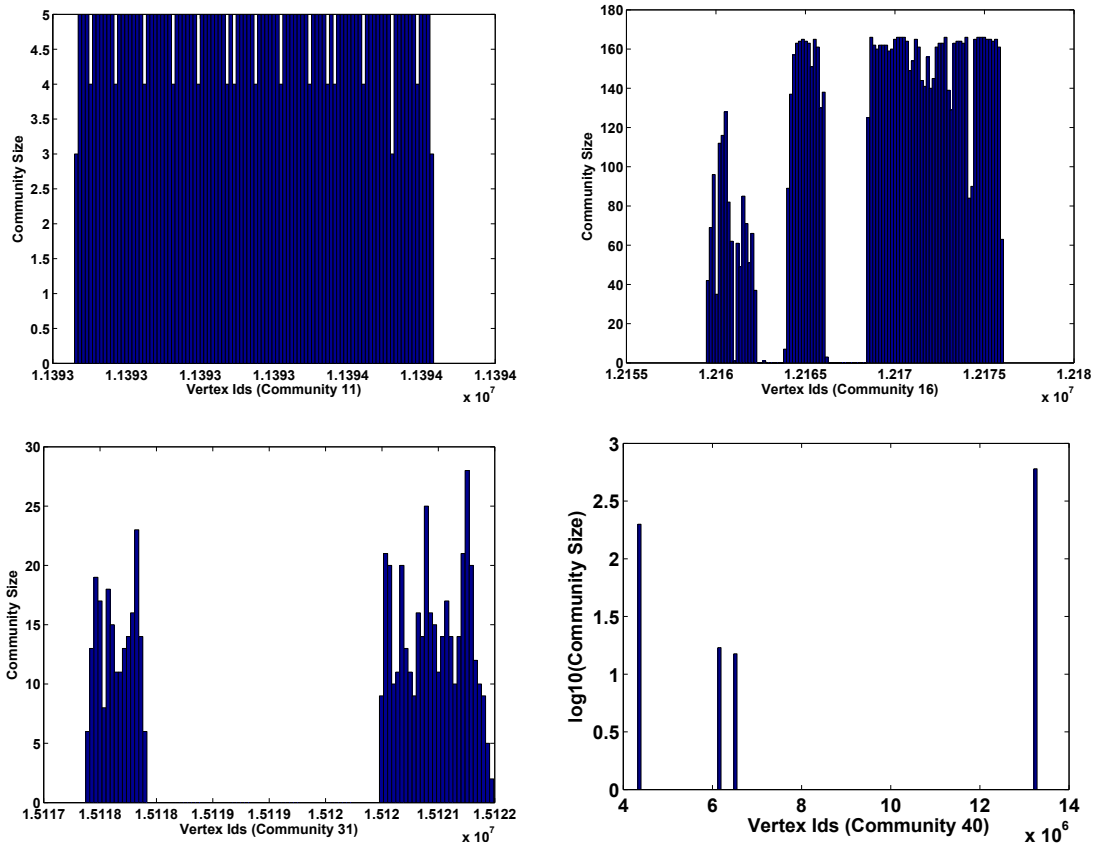


Figure 8: A sample of community seeds discovered during the compression process (from top left to bottom right, community 11, 16, 31 and 40).

273 4,662 11,463] respectively. This pattern is clearly the result of a sorting artifact, where the four ranges are actually four sub domains linking to common files, such as header and navigation links. These domains are listed in Table 5. We found this template pattern to be quite common. The total distance between the vertices in the pattern is about 9 million in a graph of about 77 million vertices (note that both axes are log scale). This type of pattern could not be easily found using reference windows, as most practical solutions of this form limit the window size to less than ten vertices. It has been suggested to sort domains in reverse order, to close the gap in vertex Ids for sub-domains. For communities 31 and 40 this looks to be effective, however for communities similar to 16 this would aid the process of discovery.

5. DISCUSSION

Virtual Node Miner has been shown to be efficient, compressing web graphs many-fold in a short amount of time. Also, the fraction of the graph held in main memory is configurable, so it can support a variety of machine configurations. The algorithm executes in two distinct phases, both of which are amenable to parallelization. More than half of the execution time is spent computing hashes, which are data level independent. The sorting at the end of the first phase is a reduce operation and needs synchronization. The second phase mines nodes in groups. These mining function calls are independent. Each call has a disjoint set of graph nodes that can be processed in parallel. Virtual nodes found by parallel mining operations are concatenated at the end of the second phase of each pass. Although the cost for each mining call will vary with the number of edges passed, in iteration of compression there are over 2 million such calls, and we expect a simple work queue to load balance well.

When examining compression ratios and execution times, we can conclude that the proposed technique is competitive with the state of the art, namely the Webgraph Framework[6]. We now discuss several advantages.

Virtual node mining based compression does not require a specific coding mechanism for edges. Competing coding schemes can be applied, such as ζ codes [5] or Huffman codes. The input data need not be sorted by URL. In fact, the proposed algorithm works equally with either sorted or unsorted data. This affords incremental graph updates, which occur at least daily for most web crawl systems. In contrast, gap-based coding schemes using reference windows are highly sensitive to the URL ordering [5, 25].

The virtual node based compression scheme natively supports popular random walk based computations such as PageRank [7], spectral graph partitioning [3], clustering [11, 27], and community finding algorithms [28]. These algorithms can be implemented on the compressed graph without requiring decompression. We briefly present a description of PageRank on the compressed graph. In compressed PageRank, each iteration is split into a sequence of sub-iterations. The push operations in each sub-iteration are of two types: push operations towards virtual nodes and push operations towards original (non-virtual) nodes. In the first sub-iteration, original nodes push probabilities to both original and virtual nodes. In subsequent sub-iterations only virtual nodes push partial sums (accumulated over previous sub-iterations) to original and downstream virtual nodes. The number of sub-iterations is equal to the longest sequence of virtual nodes

and never exceeds the number of passes used during graph compression. From Figure 5 (left) we note that each node on average has less than 1.45 virtual nodes. Thus, the total number of push operations after the first one or two sub-iterations is very small. Note that the recursive references to virtual nodes produce a directed acyclic graph (DAG) of virtual nodes. For streaming efficiency, one can envision storing the virtual nodes in a topologically sorted order.

It is also worth noting that the proposed compression scheme not only supports computations in compressed form, but also has potential for speeding up computations. It is easy to see that if the number of edges undergoes a five-fold reduction during compression, then the total number of push operations also undergoes a five-fold reduction. The five-fold reduction in push operations can be expected to produce up to a five-fold speedup. The compression scheme stores tightly connected communities as shallow hierarchies of virtual nodes. Using this representation one can efficiently answer community queries such as: *Does page X belong to a community? Who are the other members in the community page X belongs to? Do pages X and Y belong to the same community?* etc. While answering these queries we can expect to drop nepotistic/intra-domain links [14].

We consider the mining process presented in this work to be approximate because it produces support values for itemsets which may be lower than the actual support in the data set. Also, the same itemset may be found multiple times. We do not attempt to reduce these collisions – instead we compress the virtual nodes as well. The penalty is then one additional outlink, and one additional dereference.

Finally, we believe that the approach is generally applicable, and easily implementable. Alternate clustering methods can be easily inserted, as well as alternate pattern mining kernels. For example, the algorithm designer is free to use more than $O(E \log E)$ time in the mining process if tolerable.

6. RELATED WORK

This section discusses the relevant literature.

6.1 Web Graph Compression

Several existing works are targeted at web graph compression. Randall *et al.* link developed a coding scheme for their LINK database. They leverage the similarity of adjacent URLs and the distribution of the gap between outlinks for a given node.

Boldi and Vigna [5, 6] developed ζ codes, a family of simple flat codes that are targeted at gaps. ζ codes are well suited for compressing power-law distributed data with small exponents (usually in [1.1 1.3]). They achieve high edge compression and scale well to large graphs. They also demonstrate that using ζ codes one can get close to the compression achieved using Huffman codes without paying for the coding table overhead. They have also published several useful benchmark data sets that we use in this paper. On these datasets, in comparison with Huffman codes, ζ codes are shown to have no more than 5% loss in the number of bits used per edge. The typical loss is in the range of 2-3%. Both works point out the need for the graph to reside in RAM. They also require the URLs to be sorted, and are sensitive to the labels assigned to the graph vertices. As a result, they do not intrinsically support incremental graph updates. Since gap-compression has a small window of the

Seed	Start Id	End Id	Size	URL
11	11,393,132	11,394,190	1,058	http://loans69.co.uk/site-map/index_c_426.html
16	12,159,546	12,162,237	1,164	http://mobilefun.co.uk/products/6789.htm
16	12,163,934	12,166,105	1,957	http://mobilefun.co.uk/sale/Memory.htm
16	12,168,466	12,175,979	7,061	http://mobileguru.co.uk/Mobile_Technology_resource.html
31	15,117,891	15,118,397	204	http://ringtones.mobilefun.co.uk/family-guy-Mp3-Ringtones.htm
31	15,120,491	15,154,304	474	http://www.mobilefun.co.uk/sale/Samsung-S500i.htm?sortby=nameasc
40	4,368,219	4,368,870	651	http://comment.independent.co.uk
40	6,167,376	6,167,649	273	http://education.independent.co.uk
40	6,506,773	6,511,435	4,662	http://enjoyment.independent.co.uk
40	13,236,350	13,247,813	11,463	http://news.independent.co.uk

Table 5: Properties of the ranges from the community seeds shown in Figure 8.

past few nodes, these compression schemes do not incorporate community discovery into the compression of the web graph to support native community queries with a reduced memory footprint.

6.2 Min-wise Hashing

Several works use min-wise hashing to process data efficiently. Minimum hashing was first proposed by Cohen [12] to estimate the size of transitive closure and reachability sets. The technique was generalized to k-way minimum hashing by Broder and Charikar [8]. The algorithm afforded by Cohen *et al.* [13] leverages the k-way hashing technique to discover association rules with high confidence. The authors seek to find interesting implications ($A \rightarrow B$) with very low support, where A and B are both singletons. This restriction allows for column-wise comparisons for rule detection, but only discovers itemsets of length two. In addition to these smaller patterns, we seek to find long patterns in the data set. Interestingly, because they are seeking to find two similar items, their matrix is hashed orthogonally to ours. Finally, the authors note that increasing their algorithm to more than 3 columns (and hence patterns of length 4 or more) would suffer exponential overheads. However, their techniques clearly exhibit the general scalability of minimum k-way hashing. Indyk and Motwani [19] first used hashing to find nearest neighbors in high dimensional data. Goinis, Indyk and Motwani [21] subsequently improved the idea. They use hashes as signatures to localize data for answering queries in databases. They illustrate the benefits of the technique over sphere/rectangle trees when the number of dimensions is high.

6.3 Community Discovery

There has been much work recently on community discovery in the web graph, and its usefulness. Flake *et al.* [15] define a community on the web as a set of sites that have more links (in either direction) to members of the community than to non-members. They presented an efficient algorithm using a max-flow / min-cut approach. The web has also been shown to self-organize such that highly related web communities can be efficiently identified based solely on connectivity [16]. This observation has been critical in contributing to the success of link-only (or content agnostic) algorithms for finding communities.

Gibson, Kleinberg and Raghavan [17] first presented and provided evidence for structure in underlying web communities. They leverage the HITS [22] algorithm to discover communities influencing web topics. Kumar *et al.* [23] present

an alternate mechanism to discover bipartite graphs. They discover small graphs, and then grow them using HITS [22]. The authors point out that bipartite graph patterns capture communities which may not be found when enumerating cliques. It is common for two competing companies to not link to each other, but third parties will often link to both. Gibson, Kumar and Tomkins [18] introduced the strategy of sorting multiple fingerprints generated from min hashes to cluster large graphs for community discovery. Using this technique they detect large link spam farms, among other web graph communities, and illustrate the technique’s effectiveness and scalability. We find only a particular type of community structure (bipartite graphs); these may be grown to other types, a direction of future research. Most dense subgraph discovery methods [14] are very stringent in their requirements of matches. However, for compression a pattern of length two with a frequency of three is a net gain. This observation presents the need for scalable approximate data mining, as presented in this work. Also, we have seen no existing work which has attempted to conjoin graph compression with the community discovery process to improve community query response times.

6.4 Itemset Mining

There is existing research on itemset mining for streaming data. In general, this research attempt to limit main memory consumption while enumerating the full result set for a given support and error. Manku and Motwani [24] provide a single pass algorithm for mining itemsets using a bucketing strategy. The recent window of buckets is designed to reside in main memory. The algorithm outperforms a fast implementation of *Apriori* [2] on a database of size 69MB at a support of 0.1%. Toivenen [26] proposed a two pass algorithm for mining itemsets in very large databases. In the first pass, the method samples the database, computes the patterns, and establishes the negative border (the minimal set of items needed to separate the frequent sets from infrequent sets in the search lattice). In the second pass, both the frequent sets and the negative border are verified. A failure of the negative border requires an additional pass. FPGrowth [20] is an exact itemset mining algorithm which also uses a prefix tree in frequency descending order. It uses two passes of the original data set to enumerate the full set of frequent patterns at a user-provided minimum support. All the algorithms listed above scale exponentially with decreasing support and increasing unique labels. We seek to find patterns which occur at extremely low support (even in only two nodes) in databases with billions of unique labels.

7. ACKNOWLEDGEMENTS

We would like to thank the reviewers for their thorough input and many constructive suggestions.

8. CONCLUSION

This work proposes a scalable approximate data mining mechanism to compress the web graph for link servers, which incorporates the formation of global patterns. We feel these patterns can be used as seeds in the community discovery process. The algorithm exhibits high compression ratios, and scales to large data sets. The end representation averages only slightly more than one dereference per vertex for random walks. It also has several properties not present in existing compression technologies: i) it requires no particular ordering or labeling of the input data, ii) it can find global patterns in the input data, and iii) it supports any of the several available coding schemes. In addition, this work introduces a process for mining itemsets in log-linear time. Directions for future work include a parallel formulation, and a mechanism to grow larger communities from the discovered seed patterns.

9. REFERENCES

- [1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 1993.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Very Large Data Bases*, 1994.
- [3] R. Andersen, F. Chung, , and K. Lang. Local graph partitioning using pagerank vectors. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486. IEEE Press, 2006.
- [4] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. In *Software: Practice & Experience*, number 8, pages 711–726, 2004.
- [5] P. Boldi and S. Vigna. The webgraph framework ii: Codes for the world-wide web. In *Technical Report 294-03*. Universit di Milano, Dipartimento di Scienze dell'Informazione, 2003.
- [6] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International World Wide Web Conference (WWW)*, pages 595–601, Manhattan, USA, 2004. ACM Press.
- [7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th World Wide Web Conference*, 1998.
- [8] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. volume 60, pages 630–659, 2000.
- [9] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. volume 29, pages 1157–1166, 1997.
- [10] G. Buehrer, K. Chellapilla, and S. Parthasarathy. Itemset mining in log-linear time. In *OSU-CISRC-11/07-TR76*, 2007.
- [11] F. R. K. Chung. Spectral graph theory. In *American Mathematical Society*, Providence, RI, 1997.
- [12] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. In *Journal of Computer and System Science*, volume 55, pages 441–453, 1997.
- [13] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding interesting associations without support pruning. In *IEEE Transactions on Knowledge and Data Engineering*, volume 13, 2001.
- [14] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *Proceedings of the International World Wide Web Conference (WWW)*, 2007.
- [15] G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 150–160, New York, NY, 2000. ACM Press.
- [16] G. W. Flake, S. Lawrence, C. L. Giles, and F. Coetzee. Self organization of the web and identification of communities. In *IEEE Computer*, volume 35, pages 66–71, 2002.
- [17] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *HYPertext*, pages 224–235, 1998.
- [18] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of 31st International Conference on Very Large Data Bases*, 2005.
- [19] A. Goinis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.
- [20] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2000.
- [21] P. Indyk and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. In *30th Annual Symposium on Theory of Computing*, pages 604–613, 1998.
- [22] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. In *J. ACM*, volume 48, 1999.
- [23] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. In *Computer Networks*, pages 1481–1493. Elsevier Science, 1999.
- [24] G. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.
- [25] K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener. The link database: Fast access to graphs of the web. In *Proceedings of the Data Compression Conference*. IEEE Press, 2002.
- [26] H. Toivonen. Sampling large database for association rules. In *Proceedings of 22th International Conference on Very Large Data Bases*, pages 134–145, 1996.
- [27] A. Vetta. On clusterings: Good, bad and spectral. In *J. ACM*, volume 51, page 497–515, 2004.
- [28] S. White and P. Smyth. A spectral clustering approach to finding communities in graphs. In *SIAM Data Mining Conference*, 2005.