Graph Summarization for Query Evaluation

Giridhar Manoharan

EECS, Washington State University

December 27, 2016

1 Introduction

Many real-world systems involving relationship between various entities have been widely represented by 'Graphs'. Indeed in today's world, many large-scale applications need to analyse and store huge amount of data which can be best represented by graphs. In other words,'Graphs' are the fundamental abstraction to model real-world systems. For example, link structures of the world wide web, group of friends in social networks, data exchange between IP addresses in a network and the pixels of an image can all be represented as massive graph structures. In addition, other data models like relational data, non-relational data like xml, json can also be easily represented in an understandable way using graphs. Pattern matching in graphs is increasingly being used in wide range of applications like recommendation systems and image recognition.

All the above applications require analysis of large graphs with millions and even billions of nodes and edges. As a result, it is almost impossible to understand the information encoded in large graphs by mere visual inspection. Visualizing such massive graphs is a major challenge due to the difficulty of getting everything to fit in a single screen. Furthermore, developing graph mining algorithms that can scale to such gigantic proportions is another non-trivial challenge, especially when the graph is too large to fit entirely in main memory. As the size of the real world data grows every day, queries like reachability, shortest path, and pattern matching on such large graphs are increasingly computational and time consuming. In order to overcome the above difficulties, we have explored the possibility of applying graph summarization techniques for query evaluation of large graphs in this project.

1.1 Background

Graphical expression is by far the best representation of data in a concise and comparable format. As already expressed, it is being followed for the representation of many real world systems. To understand graphical summarization, we present here the methods involved in the process. Let us consider a given graph $G = (V, E)$, where V represents the set of nodes and E represents the set of edges. We represent the summarized graph G_S as $R = (S, C)$, where $S = (V_S, E_S)$ represents graph summary and C represents a set of edge corrections (Figure 1). The graph summary is an aggregated graph structure in which each node $v \in V_S$, called a supernode, corresponds to a set A_v of nodes in G, and each edge $(u, v) \in E_S$, called a super edge, represents the edges between all pair of nodes in A_u and A_v. The set of edge corrections is a set of edges of the original graph G, which are annotated as either positive ($'+'$) or negative ($'-'$) depending upon the existence of the edge or not in the original graph respectively. The intuition behind the structure of the graph summary S is to exploit the similarity of the link structure present in the nodes of many practical graphs to realize space savings.

Consider for example, in Web graphs, because of link copying between Web pages, there are clusters of pages with very similar adjacency lists [8, 7]. Similarly, communities in social networks and the Web frequently contain nodes that are densely inter-linked with one another [6]. In such graphs, if two nodes have edges to the same set (or very similar set) of other nodes, then we can combine them into a supernode and replace the two edges going to each common neighbor with a single super edge. This will significantly reduce the total number of edges that need to be stored, and lead to much smaller space overheads. Thus, in general, if there is a complete bipartite subgraph, then we can combine the two bipartite cores into two supernodes and simply replace all

the edges with a superedge between the supernodes. This will result in significant reduction of the memory requirement. Similarly, we can combine a complete clique to a single supernode with a self-edge.

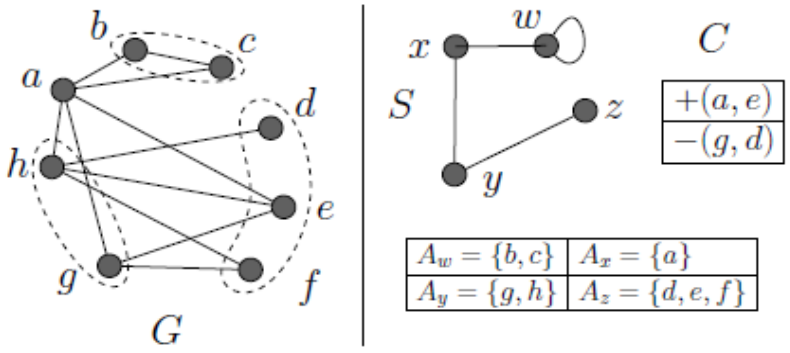

Figure 1: The two part graph representation. The LHS shows the original graph, while the RHS contains the graph summary (S), corrections (C), and the supernode mapping.

Now we will consider the correction part, C. By employing the corrections back to a summary graph S, we can reconstruct original graph G. This can be done according to the following: for each supernode $v \in V_S$, create the nodes in the set A_v, and for each super edge $(u, v) \in E_S$, add edges between all node pairs (x, y) s.t. $x \in A_u$ and $y \in A_v$. But it is possible that only a subset of these edges were actually present in G; to fix this, we keep the set of corrections C, which contains the list of edge-corrections that need to be applied to the graph constructed using S to recreate the original graph G. Specifically, for the super edge (u, v), C contains entries of the form $'-$(x,y)$'$ for the edges that were not present in G, while if the same super edge was not added to S, C will contain entries of the form $'+$(x,y)$'$ for the edges that were actually present in G.

Whereas S is a compact graph summary of G that highlights the structure and key patterns, the corrections C allow the user to reconstruct the entire graph. Re-computing the original graph from our representation can be performed very efficiently since reconstructing each node in G requires expanding just one supernode and reading the corresponding entries in C. Nodes in graphs usually represent real world objects and edges indicate relationships between objects.

1.2 Related Work

Graph compression problem is being studied in different number of fields and in research areas.
Graph Compression: Extensive literature exists in the field of Web graph compression. The aim of such compression is to optimize the space overhead of the link structure between billions of pages. Most of the work on Web-graph representations were focussed on lossless compression of web pages. Many studies [2, 3, 8, 9, 7] take advantage of well-established properties of the Web graph, e.g., pages largely pointing to other pages on the same host, and new pages adding links by copying links from an existing page. These Web pages with similar adjacency lists are encoded using a technique called reference encoding in which the adjacency list of one page is represented in terms of the adjacency list of the other. In [2], the reference encoding costs between pages are captured in an affinity graph, and a minimal spanning tree is then computed to determine the optimal reference encodings. Most of these papers, however, only focus on reducing the number of bits needed to encode a link, and none compute graph summaries since the compressed representation is not really a graph. Therefore, these methods do not provide any insight into the structure of the graph. An exception here is [7] which computes graph summaries by grouping Web pages based on a combination of their URL patterns and k-means clustering [4]. In contrast, our summaries are computed using the MDL principle, which has sound information-theoretic underpinnings.
Network Visualization: Traffic Dispersion Graphs (TDG) which are graph structures extracted from network traffic on a router, were proposed in the paper [5], which helps in detecting unknown applications on a network. This paper aims to extract relevant data at network speeds and represent them as a graph. In [10], neighbor-similarity based clustering techniques was used to classify hosts into different groups (having similar "roles", e.g., mail-servers), and to visualize these groups for

hosts on a network domain. This work focuses on performing role-classification, and not to achieve compression.

Apart from the above, graph summarization is used in the research domains such as Clustering, XML synopsis construction, Approximate query processing, etc.

2 Problem Statement

Given a large graph G, compute compact representation G_s to reduce the cost of disk space, memory space, and query time.

The goal is to find a minimum cost representation graph G_s of original graph G. Cost representation of G_s is the size of summary S (total number of super nodes and edges) and the size of corrections set C. The obtained compressed graph G_s should be a lossless representation of original graph G. From the compressed version of the graph, accuracy of correctly evaluated queries and query time are evaluated.

2.1 Input and Output

The input to the algorithm is the original graph G with nodes V_g and edges E_g. The output is the summarized graph G_s and a set of corrections C. Summarized graph G_s consists of supernodes V_s and superedges E_s. Each supernode in V_s corresponds to a set of nodes in G, and each superedge in E_s represents the edges between all pair of nodes in the two supernode sets. The corrections set specifies the list of edge-corrections that should be applied to the summary graph G_s to recreate G.

2.2 Challenges

The major challenge in the summarization of graph is that we need to find a lossless representation of the large graph .ie. the original graph should be recoverable without any loss from the compressed graph when the actions are reversed. Visualizing large graphs is a hard task and hence studying patterns in large graph is another challenge. Since large graphs can't fit in memory, processing them is time consuming. Coming up with a graph algorithm that can scale well is a difficult task. Greedy algorithm is a little slow, however with respect to the effectiveness of compression, it is far better than other algorithms like randomized and approximate algorithms.

3 Solution Approach

We require cost effective method for summarizing the graph for which we have to decide upon whether we need to merge two supernodes or not depending on whether there is a cost reduction in representation after merging.

We define edge cost between two super nodes as c_{xy}. Let E_{xy} be the actual set of edges between supernodes x and y in G, and let Π_{xy} be all pairs of edges between super node x and super node y, ($|\Pi_{xy}| = A_x * A_y$). Now, the cost without super edge between x and y is given by $|E_{xy}|$ which is the number of positive corrections that needs to be added to the corrections set. The cost with super edge between x and y is given by $1 + |\Pi_{xy} - E_{xy}|$ which is the cost of a super edge and the set of negative corrections that needs to be added to the corrections set. The minimum of these two quantities, $min|\Pi_{xy} - E_{xy}| + 1, |E_{xy}|$ is taken as the edge cost (c_{xy}). We define the cost c_v of supernode v to be the sum of the costs of all the super edges (v, x) to its neighbors $x \in N_v$. For any supernode $v \in V_s$, we define the neighbor set N_v to be the set of supernodes $u \in Vs$, s.t there exists an edge (a, b) in graph G for some node $a \in A_v$ and $b \in A_u$. Now, given a pair (u, v) of supernodes in V_s, the cost reduction $s(u, v)$ is defined as the ratio of the reduction in cost as a result of merging u and v into a new supernode w, and the combined cost of u and v before merge.

$$s(u, v) = \frac{c_u + c_v - c_w}{c_u + c_v} \tag{1}$$

The reason to pick the fractional instead of the absolute cost reduction is that the latter is inherently biased towards nodes with higher degrees, since it basically selects the node pair with the highest number of neighbors. These nodes can, however, have a large number of uncommon neighbors as

well, which implies that they should have a lower precedence than two lower degree nodes with an identical set of neighbors. The fractional cost reduction ensures that such cases do not occur by normalizing the cost reduction with the original cost.

Here we check if the merged cost is greater than the unmerged one. If $s(u, v)$ is positive then there is cost reduction after merging two nodes u and v. Notice that the maximum value that $s(u, v)$ can take is 0.5, when the neighbor sets of the two nodes are identical; on the other hand, its minima can actually be a very large negative value, but we are not interested in nodes pairs with a cost reduction value that is below zero.

3.1 The Greedy Algorithm

We have followed a Greedy algorithm for compression of large graphs. The pseudo code is given in Algorithm 1. The algorithm can be subdivided into three phases - Initialization, Iterative merging, and Output. The cost reduction given by $s(.)$ value of all pairs of nodes that are two hops apart is computed and those pairs having a positive cost reduction are inserted into a standard max heap structure H. The reason we only consider nodes that are 2 hops apart is based on the observation that any two nodes having no common neighbor cannot possibly give a cost reduction, and hence, the pairs with a positive cost reduction must be at most two hops apart (due to the presence of a 2-hop path through a common neighbor).

During the iterative merging phase, the pair (u, v) with the maximum $s(.)$ value is picked up from the heap. Then supernodes u and v are removed from V_s and merged into a new supernode w, and added to V_s. Since u and v are no longer in the graph, we remove all pairs in H containing either one of them, and then insert into H the pairs containing w with a positive cost reduction. The s(.) values of those pairs containing a supernode $x \in N_w$ will also change due to the merge. So, the costs of all the pairs containing $x \in N_w$ are recomputed and updated in the heap.

During the output phase, the summary edges and correction entries for all pairs (u, v) of neighbor supernodes in V_s are created. The super edge (u, v) will be present in the graph summary if $|A_{uv}| > (|\Pi_{uv}|+1)/2$, in which case we add correction entries $-(a, b)$ for all pairs $(a, b) \in \Pi_{uv} - A_{uv}$; otherwise, we add the corresponding $+(a, b)$ entries for all pairs $(a, b) \in A_{uv}$. This completes the construction of the representation $R = (S, C)$.

Figure 2 shows the steps taken by the greedy algorithm to construct the summary graph from original graph with corrections set.

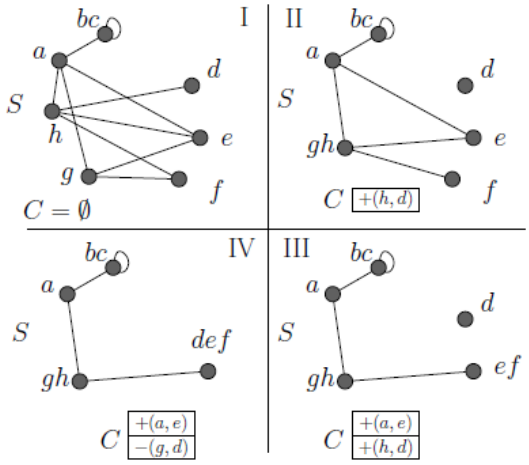

Figure 2: The steps(clockwise) for the Greedy algorithm. We show the summary S and the corrections C at the end of each step

3.2 Time Complexity Analysis

At every merge step in Greedy algorithm, for each neighbor x of the supernode w, costs of all pairs containing x are recomputed. The number of such pairs is roughly equal to the number of nodes that are at most 2-hops away from x (let us call this number 2Hop(x)). Now, summing for all the neighbors of w, this becomes roughly equal to 3Hop(w). Further, recomputing the cost requires

Algorithm 1 GreedySummarization(G)

/ ∗ *Initializationphase* ∗ /
$V_S = V_G; H = \phi$;
for `all pairs` $(u, v) \in V_S$ `that are 2 hops apart do` **do**
  if $s(u, v) > 0$ then insert $(u, v, s(u, v))$ into H;
end for
/ ∗ *Iterativemergingphase* ∗ /
while $H \neq \phi$ **do**
  Choose pair $(u, v) \in H$ with the largest $s(u, v)$ value;
  $w = u \cup v$; /* merge super nodes u and v*/
  $V_S = V_S - u, v \cup w$
  for `all` $x \in V_S$ `that are within 2 hops of u or v` **do**
    Delete (u, x) and (v, x) from H
    if $(s(w, x) > 0)$ then insert $(w, x, s(w, x))$ into H;
  end for
  for `all pairs` (x, y)`, such that` x `or` y `is in` N_w **do**
    Delete (x, y) from H;
    if $(s(x, y) > 0)$ then insert $(x, y, s(x, y))$ into H;
  end for
end while
/ ∗ *Outputphase* ∗ / $E_S = C = \phi$;
for all pairs (uv) such that $u, v \in V_S$ **do**
  if $(|A_{uv}| > (|\Pi_{uv}| + 1)/2)$ **then**
    Add (u, v) to E_S;
    Add $-(a, b)$ to C for all $(a, b) \in \Pi_{uv} - A_{uv}$;
  else
    Add $+(a, b)$ to C for all $(a, b) \in A_{uv}$;
  end if
end for
`return` representation $R = (S = (V_S, E_S), C)$;

iterating through all the edges of both the nodes (adding another hop), and updating each pair in H takes $O(\log |H|)$ time. If G contains n nodes, then the size of the heap $|H| = n * 2Hop(w)$. Hence, the total time complexity of each merge step is $O(4Hop(w) + 3Hop(w) * (\log n + \log 2Hop(w))$. Assuming an average degree of $d_{av} <= n$ for each node, this becomes $O(d_{av}^3(d_{av} + \log n + \log d_{av}))$.

4 Experimental Study

4.1 Data sets

We generated synthetic data set to test the compression made by our implementation and to run some experimental queries on the compressed data graph. We generated synthetic undirected graph data with number of nodes and edges given by the following for testing purposes,

- 100 nodes, 157 edges
- 250 nodes, 386 edges
- 500 nodes, 790 edges
- 750 nodes, 1172 edges
- 1000 nodes, 1576 edges
- 1250 nodes, 1975 edges
- 1500 nodes, 2376 edges
- 2000 nodes, 3151 edges

4.2 Platforms and Test Settings

The code is implemented in java programming language and the environment used eclipse IDE. Neo4j database is used to store the synthetic data graphs for running experiments. The experiments were run in single core Intel i5 (1.6 GHz) processor machine with 8 GB memory.

4.3 Results and Findings

4.3.1 Space savings

Cost of the original graph G is given by $|V_g| + |E_g|$, sum of the number of nodes and edges. Cost of compressed graph G_s is given by $|V_s| + |E_s| + |C|$, sum of the number of super nodes, super edges and size of the corrections set. Compression ratio is the ratio between the cost of representing the original graph G and the cost of representing the compressed graph G_s. Space savings is the reduction in size given by compressed graph G_s relative to the original graph G.

$$Space\ savings = 1 - \frac{|G_s|}{|G|} \tag{2}$$

Figure 3 shows the plot between original graph size (V_g, E_g) on the x-axis and space savings on the y-axis. The plot shows a good space savings of nearly 50-58% of the original graph. We can notice from the plot that space savings increases as the original graph size (V_g, E_g) is increased.

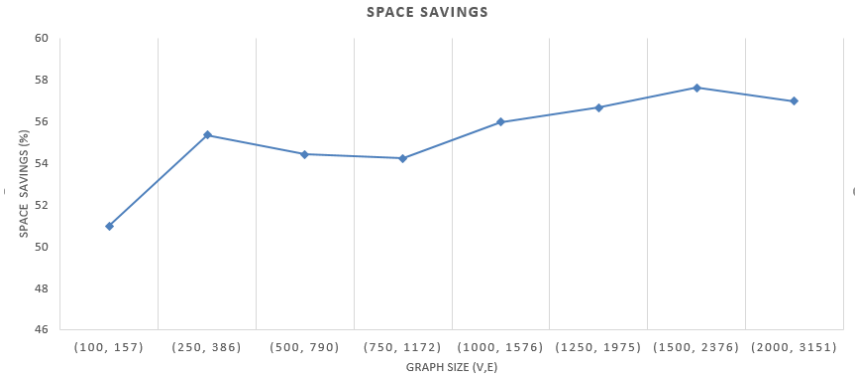

Figure 3: Percentage of space saved w.r.t size of the original graph

4.3.2 Running time

Since the algorithm is greedy and time complexity is of the order $O(d^3(d + \log(n * d))$ where d is the average degree of nodes and n is number of nodes, we can expect a slow running time. Figure 4 shows a plot between the graph size in terms of nodes on the x-axis and running time in seconds on the y-axis. The plot shows that there is an exponential increase in algorithm running (compression) time as the size of the graph is increased.

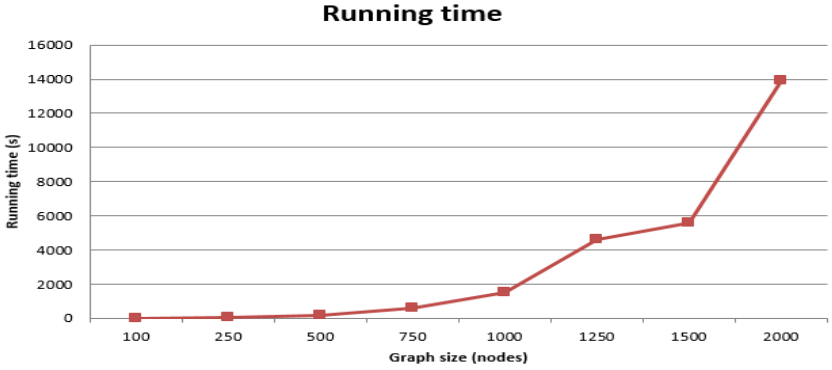

Figure 4: Running time of the Greedy algorithm w.r.t size of the original graph

4.3.3 Accuracy of correctly evaluated queries

A number of queries N is sampled and each query Q is of the form $Q(u, v, \theta)$. The output of the query is $True$ if node u can reach node v within theta hops and $False$ otherwise. A query Q is correctly evaluated if the result of running the query in both G and G_s is same. We sampled $N = 100$ queries in each synthetic data graph and conducted experiments with two θ values - 6 hops and 12 hops. Accuracy of the compressed graph is defined as the ratio of number of correctly evaluated queries to the total number of queries N. Figure 5 shows the plot between original graph size in nodes and accuracy of correctly evaluated queries in G_s. From the plot we can see that accuracy of correctly evaluated queries decreases with increase in graph size. This is because as the graph size increases, the probability of the distance between two sampled nodes greater than the threshold in original graph G is higher, and after compression the distance between the two nodes can get lesser than the threshold in compressed graph G_s. As a result accuracy decreases. From the plot, we can also notice that accuracies with $\theta = 6$ are lower than accuracies with $\theta = 12$. This is because in the original graph the number of pairs of nodes with distance within 12 hops will be greater than those pairs of nodes with distance 6 hops between them. As the number of queries with such pairs having distance 12 hops will be higher (since random selection of nodes), the query result will be true in both G and G_s. Hence, accuracy increases as θ is increased.

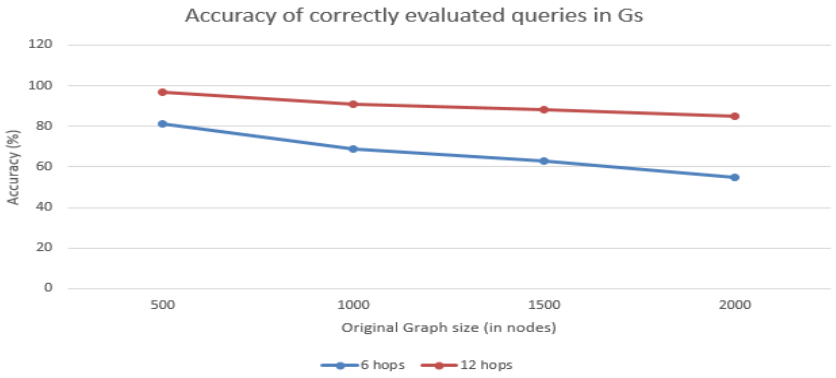

Figure 5: Accuracy of correctly evaluated queries w.r.t original graph size

4.4 Reduction in time for bounded Search

A query is run by arbitrarily choosing a node x in G and the total time taken to reach all the nodes (let's call this set n) that are 12 hops away from the arbitrary node is noted. Now the same query is experimented on the compressed graph G_s. The total time taken to reach all the nodes in set n in the compressed graph is noted. Figure 6 shows the plot between the time taken for bounded search in G and G_s with respect to graph sizes. From the plot we can notice that as the graph size is increased, the bounded search query time on compressed graph decreases marginally. So there is no significant difference between the query time execution in G and G_s.

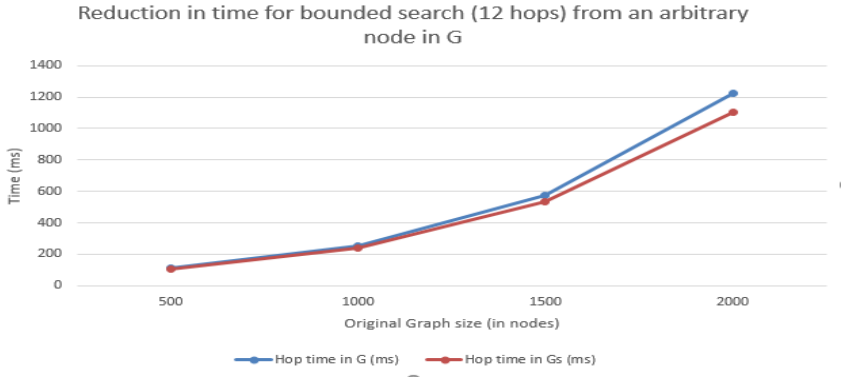

Figure 6: Reduction in time for bounded search w.r.t original graph size

4.5 Summary of Experimental Result

Graph summarization results in considerable saving of memory space. Nearly 50-58% reduction in space than the original graph could be achieved. Further, it is also found that space savings increases as the original graph size (V_g, E_g) is increased. Running time is found to be minimum for the graphs with few number of nodes and an exponential increase in running time is noticed as the size of the graph is increased. Accuracy of correctly evaluated queries depends on both original graph size and number of hops. Gradual decrease in accuracy of correctly evaluated queries is noticed with increase in graph size. Similarly, it is noticed that higher the number of hops, higher will be the accuracy. Marginal reduction in the bounded search query time is noticed for compressed graphs. No significant difference is observed between the query time execution in G and G_s.

5 Conclusions and Future Work

Experimental study using greedy algorithm has shown that considerable memory space can be saved due to graph summarization. However, since Greedy algorithm is iterative, it results in high running time cost. Moreover, no significant variation is noticed with respect to improvement of query time between original and compressed graphs. It is felt that the compression strategy employed does not reduce query time significantly. So, in order to improve the shortfalls of query execution time, other compression strategies like compression based on frequent query patterns may be explored for faster query execution. Randomized algorithms may work faster, but doesn't give optimized compression. Approximate algorithms can be used if some loss is affordable in compressed representation of graph.

References

[1] Saket Navlakha, Rajeev Rastogi and Nisheeth Shrivastava - Graph Summarization with Bounded Error. SIGMOD 2008.

[2] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In Data Compression Conference, pages 203–212, 2001.

[3] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In WWW, pages 595–602, 2004.

[4] R. C. Dubes and A. K. Jain. Algorithms for Clustering Data. Prentice Hall, 1988.

[5] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese. Network monitoring using traffic dispersion graphs (tdgs). In IMC, pages 315–320, 2007.

[6] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling the web for emerging cyber-communities. In WWW, pages 1481–1493, 1999.

[7] S. Raghavan and H. Garcia-Molina. Representing web graphs. In ICDE, pages 405–416, 2003.

[8] K. H. Randall, R. Stata, J. L. Wiener, and R. Wickremesinghe. The link database: Fast access to graphs of the web. DCC, pages 122–131, 2002.

[9] T. Suel and J. Yuan. Compressing the graph structure of the web. In Data Compression Conference, pages 213–222, 2001.

[10] G. Tan, M. Poletto, J. V. Guttag, and M. F. Kaashoek. Role classification of hosts within enterprise networks based on connection patterns. In USENIX Annual Technical Conference, General Track, pages 15–28, 2003.