Graph Summarization with Bounded Error

Saket Navlakha[*]
Dept. of Computer Science
University of Maryland
College Park, MD, USA-20742
saket@cs.umd.edu

Rajeev Rastogi[†]
Yahoo! Labs
Bangalore, India
rrastogi@yahoo-inc.com

Nisheeth Shrivastava
Bell Labs Research
Bangalore, India
nisheeths@alcatel-lucent.com

ABSTRACT

We propose a highly compact two-part representation of a given graph G consisting of a graph summary and a set of corrections. The graph summary is an *aggregate* graph in which each node corresponds to a set of nodes in G, and each edge represents the edges between all pair of nodes in the two sets. On the other hand, the corrections portion specifies the list of edge-corrections that should be applied to the summary to recreate G. Our representations allow for both lossless and lossy graph compression with bounds on the introduced error. Further, in combination with the MDL principle, they yield highly intuitive coarse-level summaries of the input graph G. We develop algorithms to construct highly compressed graph representations with small sizes and guaranteed accuracy, and validate our approach through an extensive set of experiments with multiple real-life graph data sets.

To the best of our knowledge, this is the first work to compute graph summaries using the MDL principle, and use the summaries (along with corrections) to compress graphs with bounded error.

Categories and Subject Descriptors

E.2 [**Data Storage Representations**]; H.3 [**Information Storage and Retrieval**]: Information Storage

General Terms

Algorithms

Keywords

Graph Compression, Minimum Description Length, Approximation

[*]This work was done when the author was visiting Bell Labs Research, Bangalore, India.

[†]This work was done when the author was with Bell Labs Research, Bangalore, India.

1. INTRODUCTION

Graphs are a fundamental abstraction that have been employed for centuries to model real-world systems and phenomena. Today, numerous large-scale systems and applications need to analyze and store massive amounts of data that involve interactions between various entities – this data is best represented as a graph; for instance, the link structure of the World Wide Web, group of friends in social networks, data exchange between IP addresses, market basket data, etc., can all be represented as massive graph structures. Below, we look at some of these application domains.

- *World Wide Web.* The Web has a natural graph structure with a node for each page and a directed edge for each hyperlink. This link structure of the Web has been exploited very successfully by search engines like Google [4] to improve search quality. Other contemporary research works mine the Web graph to find dense bipartite cliques, and through them Web communities [21] and link spam [12]. Recent estimates from search engines put the size of the Web graph at around 3 billion nodes and more than 50 billion arcs [3]. (Note that these are clearly lower bounds since the Web graph has been growing rapidly over the years as more of the Web gets discovered and indexed.) Thus, the Web graph can easily occupy many terabytes of storage.

- *Social Networking.* Popular social networking websites like Facebook, MySpace and LinkedIn cater to millions of users at a time, and maintain information about each user (nodes) and their friend-lists (edges). Mining the social network graph can provide valuable information on social relationships between users, the music, movies, etc. that they like, and user communities with common interests.

- *IP Network Monitoring.* IP routers export records containing source and destination IP addresses, number of bytes transmitted, duration, etc. for each IP communication flow. Recently, Iliofotou et. al. [14] proposed the idea of extracting *Traffic Dispersion Graphs* (TDGs) from network traces, where each node corresponds to an IP address and there is an edge between any two IP addresses who sent traffic to each other. Such graphs can be used to detect interesting or unusual communication patterns, security vulnerabilities, hosts that are infected by a virus or a worm, and malicious attacks against machines. These graphs, however, can be large – it has been reported in [7] that the AT&T IP backbone network alone generates 500 GB of IP flow data per day (about ten billion fifty-byte records).

- *Market Basket Data.* Market basket data contains information about products bought by millions of customers. This is essentially a bipartite graph with an edge between a customer and every product that he or she purchases. Mining this graph to find groups of customers with similar buying patterns can help with customer segmentation and targeted advertising.

A common theme in all of the above applications is the need to analyze large graphs with millions and even billions of nodes and edges. Visualizing such massive graphs is clearly 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.

In this paper, we propose *information-theoretic* techniques for computing compressed graph representations. Our graph representation R has two parts: the first is a graph *summary* S (much smaller than the input) that captures the important clusters and relationships in the input graph, while the second is a set of *corrections* C that helps to recreate the original graph, if necessary. Moreover, if the user is willing to tolerate a certain amount of error in the recreation process, we also show how to exploit this leeway to get further reduction in the size of the representation, and strike a trade-off between accuracy and memory.

Our graph representation has the following benefits:

- The summary S is itself a graph with substantially fewer nodes and links that can easily fit in memory. Thus, it is amenable to visualization and other graph analysis techniques (e.g., finding communities, customer segments); specifically, it provides insight into the high-level structure of the graph, and the dominant relationships among the various node clusters. Unlike clustering algorithms [17, 8] that group nodes based on their similarity or distances, our summary is computed using information-theoretic principles.

- Our representation allows for a high degree of compression to be achieved for general graphs. In addition, it supports a tunable ϵ parameter that can be used to achieve lossy compression, but with bounded errors. Essentially, the ϵ parameter allows us to trade accuracy for higher compression. Thus, with our highly space-efficient representations, approximate graphs can be stored in main memory and efficiently analyzed using graph algorithms. In contrast, most of the existing proposals [1, 30, 3] only support lossless compression for Web graphs.

1.1 A Generic Graph Representation

Given a graph $G = (V_G, E_G)$, our representation for it $R = (S, C)$ consists of a *graph summary* $S = (V_S, E_S)$ and a set of edge *corrections* C (see 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 superedge, represents the edges between all pair of nodes in A_u and A_v. The second part of the representation is a set of edges of the original graph G, which are annotated as either positive ($'+'$) or negative ($'-'$).

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. For instance, it is well known that in Web graphs, because of link copying between Web pages, there are clusters of pages with very similar adjacency lists [27, 26]. Similarly, communities in social networks and the Web frequently contain nodes that are densely inter-linked with one another [21]. Now, in such graphs, if two nodes have edges to the same set (or very similar set) of other nodes, then we can collapse them into a supernode and replace the two edges going to each common neighbor with a single superedge. Clearly, this will significantly reduce the total number of edges that need to be stored, and lead to much smaller space overheads. Generalizing further, if there is a complete bi-partite subgraph, then we can collapse the two bi-partite cores into two supernodes and simply replace all the edges with a superedge between the supernodes, thus reducing the memory requirement dramatically. Similarly, we can collapse a complete clique to a single supernode with a *self-edge*.

Next, let us look at the correction part, C. Note that we can reconstruct G by "expanding" the summary S, as follows: for each supernode $v \in V_S$, create the nodes in the set A_v, and for each superedge $(u, v) \in E_S$, 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 superedge (u, v), C contains entries of the form $' - (x, y)'$ for the edges that were not present in G, while if the same superedge was not added to S, C will contain entries of the form $' + (x, y)'$ for the edges that were actually present in G. We define the function $g(R)$ that maps a representation R to the equivalent graph G s.t. an edge (x, y) is present in G iff either (a) C contains an entry $' + (x, y)'$, or (b) S contains a superedge (u, v) s.t. $x \in A_u$ and $y \in A_v$ and C does not have an entry $' - (x, y)'$.

So while 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. Observe that recomputing the original graph (or a specific subgraph within it) 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. Before going into the details of our problem formulation, we first give a simple example of how the graph summaries plus corrections can be used to recover the original graph.

EXAMPLE 1. *Figure 1 shows a sample graph (left) and its representation (right). Note that the graph is compressed from size (number of edges)* 11 *to* 6 *(*4 *edges in graph summary and* 2 *edge corrections). The neighborhood of a node (say g) in the graph is reconstructed as follows. First, find the supernode (y) that contains g, then add edges from g to all the nodes in a supernode that is a neighbor of y. This gives the edges* $\{(g, a), (g, d), (g, e), (g, f)\}$. *Next, apply the corrections to the edge set, that is, delete all edges with a* $'-'$ *entry (edge* (g, d)*), and add edges with a* $'+'$ *entry (none in this example). This gives the set of edges in the neighbor-*

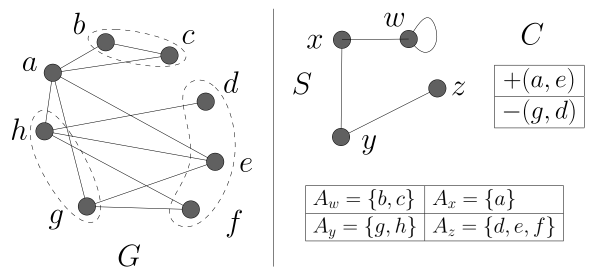

Figure 1: The two part graph representation. The LHS shows the original graph, while the RHS contains the graph summary (S), corrections (C), and the supernode mapping.

hood of g as $\{(g, a), (g, e), (g, f)\}$, which is the same as in the original graph. This can be repeated for all nodes in V_G to recover the original graph.

Our graph representations have similarities to the S-Node representations for Web graphs proposed in [26]. However, there are some differences. First, we allow supernodes to have self-edges which are not present in [26]; these, as we saw above, can be very effective for coalescing dense cliques into a single supernode. Instead, in [26], the graph structure within each supernode is compressed using a different reference encoding scheme. Second, [26] stores a (positive) superedge between supernodes u and v if there is even a single edge between nodes in A_u and A_v in the graph. In contrast, we store a superedge between supernodes u and v only if the nodes in A_u are *densely* connected to nodes in A_v (see MDL representation below). Thus, our representations are less cluttered, smaller in size, and more suitable for visual data mining. Finally, our representations allow for lossy compression and are computed using information-theoretic techniques. On the other hand, [26] uses URL-specific information to carry out lossless compression of Web graphs.

Note that our representations are equally applicable to directed as well as undirected graphs. However, for simplicity of exposition, we will only consider undirected graphs in the remainder of the paper.

1.2 MDL Representation

Rissanen's *Minimum Description Length* (MDL) principle [28] has its roots in information theory. It roughly states that the best theory to infer from a set of data is the one which minimizes the sum of (A) the size of the theory, and (B) the size of the data when encoded with the help of the theory.

In our setting, the data is the input graph G, the theory is the summary S, and the corrections C essentially represent the encoding of the data in terms of the theory. We define the cost of a representation $R = (S, C)$ to be the sum of the storage costs of its two components, that is, $cost(R) = |E_S| + |C|$. (We ignore the cost of storing the mappings A_v for supernodes $v \in V_S$ since this will generally be small compared to the storage costs of the edge sets E_S and C.) In the cost expression, the first term $|E_S|$ corresponds to (A) and the second term $|C|$ corresponds to (B). Thus, if $\hat{R} = (\hat{S}, \hat{C})$ denotes the minimum cost representation, then the MDL principle says that $\hat{S}$ is the "best possible" summary of the graph. In other words, $\hat{R}$, in addition to being the most compressed representation of graph G, also contains $\hat{S}$ which is the best graph summary. We will refer to the minimum cost representation $\hat{R}$ as the MDL representation.

Interestingly, the edge sets E_S and C, and as a consequence the cost of a representation R, are determined solely based on the supernodes contained in V_S. To understand how these are computed for a fixed V_S, consider any two supernodes u and v in V_S. We define Π_{uv} as the set of all the pairs (a, b), such that $a \in A_u$ and $b \in A_v$; this set represents all possible edges of G that may be present between the two supernodes. Furthermore, let $A_{uv} \subseteq \Pi_{uv}$ be the set of edges *actually* present in the original graph G ($A_{uv} = \Pi_{uv} \cap E_G$). Now, we have two ways of encoding the edges in A_{uv} using the summary and correction structures. The first way is to add the superedge (u, v) to S and the edges $\Pi_{uv} - A_{uv}$ as negative corrections to C, and the second is to simply add the edges in the set A_{uv} as positive corrections to C. The memory required for these two alternatives is $(1 + |\Pi_{uv} - A_{uv}|)$ and $|A_{uv}|$, respectively. We will simply choose the one with the smaller memory requirements for encoding the edges A_{uv}.

Based on the above discussion, the cost of representing the edge set A_{uv} between supernodes u and v in the representation is $c_{uv} = \min\{|\Pi_{uv}| - |A_{uv}| + 1, |A_{uv}|\}$. Further, the superedge (u, v) will be present in the graph summary S iff $A_{uv} > (|\Pi_{uv}| + 1)/2$; the positive and negative corrections are then chosen accordingly. Notice that given the set of supernodes V_S, the cost of the representation can be computed by looking at every supernode pair and making a simple choice as described above. However, finding the best set of supernodes for the MDL representation $\hat{R}$ is a much harder problem which we tackle in Section 3.

Problem Statement: Given a graph G, compute its MDL representation $\hat{R}$.

Because of the MDL principle and since our graph representation R includes a summary S, a by-product of computing the most compressed graph representation is that we get the best possible graph summary for free. Our approach is unique in this regard, since other compression schemes do not generate such graph summaries. Of course, the best graph summary is a subjective notion, and so it is difficult to capture precisely. But intuitively, it is something at a much higher level of abstraction that provides an insight into the coarse-level structure of the graph, the main groupings, and the important relationships between the groups.

Also, note that in our representation, each node in G belongs to *exactly* one supernode in S; hence nodes in V_S form disjoint subsets of nodes whose union covers the set V_G. There may be other approaches which can take advantage of overlapping subsets (supernodes) to get better compression (similar to the minimum clique cover or minimum complete bipartite subgraph cover problems [11]); however, we will only consider the disjoint case because we want S to be a graph that is easy to visualize. Observe that the disjoint case is conceptually similar to graph clustering, where the aim is to divide the nodes into (disjoint) clusters by putting similar nodes into the same cluster. However, clustering methods do not follow an information-theoretic approach like ours which seeks to find the MDL summary $\hat{S}$ with the minimum space requirements to represent the input graph.

1.3 Approximate Representations

Thus far, we have discussed representations that reproduce the original graph G exactly. However, for many of the applications described earlier, recreating the exact graph may not be necessary; rather, finding a reasonably accurate reconstruction may be good enough. To be more concrete, consider the following key graph operation required by all of the applications: *Given a node, find the set of all its neighbors in graph G.* Now, instead of the exact set of neighbors, if we were to return an approximate neighbor set that is reasonably close to the exact set, then that should be acceptable in most cases. For example, with bounded errors in the neighbor set, we should still be able to discern communities in Web graphs and social networks, and suspicious communication patterns in IP flow graphs. Similarly, we should be able to obtain good PageRank approximations for Web pages knowing only an approximate set of pages that each page points to. And finally, in our market basket data application, we should still be able to cluster customers with similar purchasing habits or compute most of the frequent itemsets even with partial knowledge of the items bought by each customer.

We now proceed to define an ϵ-approximate representation, denoted by R_ϵ, that can recreate the original graph within a user-specified bounded error ϵ ($0 \leq \epsilon \leq 1$). The structure of R_ϵ is identical to representation R discussed earlier; thus, it too consists of a (summary, corrections) pair (S_ϵ, C_ϵ). But unlike R, it provides the following weaker guarantee for the reconstructed graph $G_\epsilon = g(R_\epsilon)$: For every node $v \in G$, if N_v and N'_v denote the set of v's neighbors in G and G_ϵ, respectively, then

$$error(v) = |N'_v - N_v| + |N_v - N'_v| \leq \epsilon |N_v| \qquad (1)$$

where, $N'_v - N_v$ is the set difference operation. Here, the first term in the equation represents the nodes included in the approximate neighbor set N'_v but were not present in the original neighbor set N_v, while the second term represents vice-versa. In other words, for each node in G, the ϵ-representation R_ϵ retains at least $(1-\epsilon)$ fraction of the original neighbors correctly, while erring in (adding or deleting) at most ϵ fraction of neighbors. The motivation here is that since an ϵ-representation R_ϵ is permitted to contain some error, it will be more compact than the exact representation for the same graph. Thus, to get the highest compression ratio, we want to find an R_ϵ with the smallest cost.

Problem Statement: Given a graph G and $0 \leq \epsilon \leq 1$, compute the minimum cost ϵ-representation.

Observe that the MDL representation $\hat{R}$ is essentially a representation R_0 with error parameter $\epsilon = 0$ that has the minimum cost. Thus, one approach to compute a minimum cost R_ϵ is to first compute $\hat{R}$, and then delete edges from $\hat{C}$ or $\hat{S}$ that do not violate Equation (1) for any node $v \in G$. As ϵ increases, we can remove more edges from both the graph summary and corrections, and reduce the cost of the representation even further.

Example 2. *Consider the graph in Figure 1 and suppose $\epsilon = 1/3$. From the corrections C, if we remove the entry $+(a, e)$, then the approximate neighbor sets for a and e would be $N'_a = \{b, c, g, h\}$ and $N'_e = \{g, h\}$. Since the neighbor sets for a and e in the original graph G are $N_a = \{b, c, e, g, h\}$ and $N_e = \{a, g, h\}$, the approximate neighbor sets N'_a and N'_e satisfy Equation (1). So we can remove $+(a, e)$ from C and reduce its size without violating the error bounds.*

Notice that the edge corrections (that we remove) can be both positive and negative; hence the final neighbor set may miss some true neighbors of v and also contain some nodes that are not in N_v. Further, note that both edge corrections in $\hat{C}$ and superedges in $\hat{S}$ may be removed without violating the ϵ guarantee. In Section 4, we present a scheme for finding the set of edges to remove from $\hat{R}$ that gives the maximum cost reduction.

1.4 Our Contributions

The main contributions of our work are as follows.

- We present a representation for graphs as a (summary, corrections) pair. Our graph representations are highly compact, and allow for both lossless and lossy graph compression with bounds on the introduced error. In combination with the MDL principle, our representations yield highly intuitive coarse-level graph summaries.

- We develop two parameter-less algorithms, Greedy and Randomized, to compute the MDL representation with small size. Greedy repeatedly picks the best pair of nodes to merge in the entire graph and outputs a highly compressed representation. On the other hand, Randomized performs the best merge on a randomly selected node–it loses out on compression but is substantially faster in practice.

- We also devise two schemes (ApxMdl and ApxGreedy) to compute the minimum cost ϵ-representation R_ϵ. The first uses a matching algorithm to remove the maximum possible correction edges from the MDL representation while still satisfying the accuracy constraints. The second incorporates the ϵ error constraint into each step of Greedy used to compute the MDL representation.

- Through extensive experimental evaluation on real life graph data sets from various domains, such as WWW and RouteView, we show the effectiveness of our schemes in practice. Our results show that we get representations that are less than 30% and 40% of the original size, respectively; and with $\epsilon = .1$, the size further reduces by 10% of the size of the exact representation.

To the best of our knowledge, this is the first work to compute graph summaries using the MDL principle, and use the summaries (along with corrections) to compress graphs with bounded error.

2. RELATED WORK

The graph compression problem, in one form or another, has been studied in a number of diverse research areas.

Web Graph Compression. The most extensive literature exists in the field of Web graph compression, which aims to optimize the space overhead of the link structure between billions of pages. Much of the work has focused on lossless compression of Web pages so that the compact Web-graph representations can then be used to calculate measures such as PageRank [4] or authority vectors [19]. Several studies [1, 3, 27, 30, 26] 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 [1], 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 [26] which computes graph summaries by grouping Web pages based on a combination of their URL patterns and k-means clustering [9]. In contrast, our summaries are computed using the MDL principle, which has sound information-theoretic underpinnings.

In a very different setting, [20, 12] devise algorithms to extract large dense subgraphs from the Web graph, since these typically correspond to online communities or link spam. Thus, the objective in [20, 12] is very different from ours – we are primarily interested in finding a grouping of nodes that minimizes the space required to represent the graph.

Clustering. In the data mining community, graph clustering has been widely used as a tool for summarization and trend analysis [9]. The general theme of clustering is to group *similar* nodes in a cluster, while making sure that nodes in two separate clusters are not similar. In our setting, the similarity among (unlabeled) nodes can be defined using standard measures like the min-hop distance, the Jaccard Coefficient [9] on their neighbor sets, or linear matrix transformations [2]. Although clustering algorithms may give meaningful insights into the dominant patterns in the graph, they typically employ distance- or similarity-based metrics to compute the clusters containing similar nodes. In contrast, we use information-theoretic metrics to group nodes such that the graph representation is as compact as possible. Another problem with many of the widely-used clustering algorithms, such as METIS [17], Graclus [8], k-means and spectral clustering [24], is that they require the user to specify the number of partitions beforehand, which is typically hard to estimate and not required in our setting.

Like us, AutoPart [5] uses the MDL principle to compute disjoint node groups such that the number of bits required to encode the graph's adjacency matrix is minimized. However, [5] proposes a top-down scheme that iteratively splits node groups starting with a single node group. In our experiments, we found that AutoPart's top-down scheme gives very little compression, and performs much worse than our bottom-up approach based on greedily merging node groups. Further, Autopart only does lossless compression – so its performance relative to our Greedy scheme degrades even further when the compressed graph is permitted to have bounded error. In [22], the authors apply the MDL principle to summarize cells of interest in OLAP data by means of a covering with regions. However, their methods exploit the inherent data hierarchy and spatial properties of database tables, and thus cannot be generalized to compress general graph structures.

XML Synopsis Construction. Many recent papers [25, 18, 23] have proposed path-index structures for XML data graphs to estimate the selectivities of complex path expressions over XML documents. The basic idea is to group identically labeled element nodes in the data graph into coarser index-graph nodes based on the set of incoming label paths at each data element. These indices, while effective at capturing the path structure in XML graphs, are ill-suited for summarizing the structure of general graphs. This is because, unlike XML graphs where labels are frequently repeated across element nodes, nodes in general graphs have distinct labels. For example, in a Web graph, every node has a distinct URL; similarly, social network and IP network graph nodes correspond to different users and IP addresses, respectively. As a result, the path-index construction schemes of [25, 18, 23] will not coalesce any of the nodes, and will output a final graph summary that is identical to the original graph. Furthermore, although path indices are good for estimating path selectivities, they cannot be used to determine the (approximate) neighbors of an element node or recreate the original graph with bounded errors on neighbor sets like we do.

Approximate Query Processing. There is a vast body of work on maintaining synopses like samples [13], histograms [15], and wavelets [6] to provide approximate answers to relational queries. However, these have limited applicability to our graph scenario for the following reasons. First, many of the approximation techniques like sampling are more suitable for generating estimates for aggregate quantities (e.g., counts or averages) as opposed to set-valued answers (e.g., a node's neighbors). Second, many of the real-world graphs are typically sparse, which complicates the task of synopses construction. And finally, histograms and wavelets do not produce high-level graph summaries that can be visualized to find interesting patterns. We would also like to point out here that since graphs are traditionally represented as a two-column relation with one tuple per edge (that stores its two vertex endpoints), relational compression techniques like fascicles [16] will not work well for graphs.

Network Visualization. In a recent paper [14], graph structures called Traffic Dispersion Graphs (TDG), extracted from network traffic on a router, were proposed as a means to detect unknown applications on a network. However, the aim in [14] is solely to extract the relevant data at network speeds and display it as a graph, which is complementary to our work. In [31], the authors use neighbor-similarity based clustering techniques to classify hosts into groups (having similar "roles", e.g., mail-servers), and to visualize these groups for hosts on a network domain. However, their main focus is on performing this role-classification, and not to achieve compression.

3. COMPUTING MDL REPRESENTATIONS

In this section we present two algorithms for finding the MDL representation $\hat{R}$. The first algorithm, called Greedy, iteratively combines node pairs that give the maximum cost reduction into supernodes. The second algorithm, called Randomized, is a light-weight randomized scheme that, instead of merging the globally best node pair, randomly picks a node and merges it with the best node in its vicinity.

Algorithm 1 GREEDY(G)

```
1: /* Initialization phase */
2: $V_S = V_G$;   $H = \emptyset$;
3: for all pairs $(u, v) \in V_S$ that are 2 hops apart do
4:     if $(s(u, v) > 0)$ then insert $(u, v, s(u, v))$ into $H$;
5: end for
6:
7: /* Iterative merging phase */
8: while $H \neq \emptyset$ do
9:     Choose pair $(u, v) \in H$ with the largest $s(u, v)$ value;
10:    $w = u \cup v$; /* merge supernodes $u$ and $v$ */
11:    $V_S = V_S - \{u, v\} \cup \{w\}$;
12:    for all $x \in V_S$ that are within 2 hops of $u$ or $v$ do
13:        Delete $(u, x)$ and $(v, x)$ from $H$;
14:        if $(s(w, x) > 0)$ then insert $(w, x, s(w, x))$ into $H$;
15:    end for
16:    for all pairs $(x, y)$, such that $x$ or $y$ is in $N_w$ do
17:        Delete $(x, y)$ from $H$;
18:        if $(s(x, y) > 0)$ then insert $(x, y, s(x, y))$ into $H$;
19:    end for
20: end while
21:
22: /* Output phase */
23: $E_S = C = \emptyset$;
24: for all pairs $(u, v)$ such that $u, v \in V_S$ do
25:    if $(|A_{uv}| > (|\Pi_{uv}| + 1)/2)$ then
26:        Add $(u, v)$ to $E_S$;
27:        Add $-(a, b)$ to $C$ for all $(a, b) \in \Pi_{uv} - A_{uv}$;
28:    else
29:        Add $+(a, b)$ to $C$ for all $(a, b) \in A_{uv}$;
30:    end if
31: end for
32: return representation $R = (S = (V_S, E_S), C)$;
```

3.1 The GREEDY Algorithm

We now present our first scheme, called GREEDY. To understand the intuition behind this approach, recall that in a graph there may be many pairs of nodes that can be merged to give a reduction in cost. Typically, any two nodes that share common neighbors can give a cost reduction, with more number of common neighbors usually implying a higher cost reduction. Based on this observation, we define the cost reduction $s(u, v)$ (see below) for any given pair of nodes (u, v). In GREEDY, we iteratively merge the pair (u, v) in the graph with the maximum value of $s(u, v)$ (the best pair).

As GREEDY progresses, we maintain a set of supernodes V_S, that constitute the supernodes in the graph summary S. For any supernode $v \in V_S$, we define the neighbor set N_v to be the set of supernodes $u \in V_S$, s.t. there exists an edge (a, b) in graph G for some node $a \in A_v$ and $b \in A_u$. Recall that the cost of the superedge (v, x) from supernode v to a neighbor $x \in N_v$ is $c_{vx} = \min\{|\Pi_{vx}| - |A_{vx}| + 1, |A_{vx}|\}$. We will define the cost c_v of supernode v to be the sum of the costs of all the superedges (v, x) to its neighbors $x \in N_v$. Now, given 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 the merge.

$$s(u, v) = (c_u + c_v - c_w)/(c_u + c_v) \qquad (2)$$

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 common 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. It is important to observe that such normalization does not make the cost of any pair switch from positive to negative (or zero), or vice-versa. In other words, if merging a pair gives some cost reduction, then $s(\cdot)$ will not prevent us from picking it eventually (but can only change the order in which it is considered). Notice that the maximum value that $s(u, v)$ can take is .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 of course not interested in node pairs with a cost reduction value that is below zero.

We are now ready to describe the GREEDY algorithm (Algorithm 1). The algorithm can be subdivided into three phases—Initialization, Iterative merging, and Output. In the Initialization phase, we compute all the node pairs that have a positive cost reduction; we examine all the nodes in V_S that are 2 hops apart, and compute their $s(\cdot)$ value. 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). To efficiently pick the node pair with the maximum $s(\cdot)$ value, we use a standard max-heap structure H to store all the pairs in the graph with $s(u, v)$ greater than 0. We add all the pairs computed in the initialization step to H, and use it later to determine the node pair with the maximum cost reduction in constant time.

During the Iterative merging phase, we first pick the pair (u, v) with the maximum $s(\cdot)$ value from the heap. We then remove supernodes u and v from V_S, merge them into a new supernode w, and add w 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. Observe that there may still be some more pairs (not containing u, v or w) whose $s(\cdot)$ values may have changed. Consider the supernode $x \in N_w$ which was previously a neighbor of u or v (or both). The cost of representing the edge (x, u) (and/or (x, v)) may change due to the merge of u and v. This, in turn, could change x's cost (c_x), and the cost reduction of any pair containing x. So we must recompute the costs of all the pairs containing $x \in N_w$ and update them in the heap (this may require both adding pairs to, and removing pairs from, the heap). The following example illustrates the steps of the GREEDY algorithm on a simple graph.

EXAMPLE 3. *Figure 2 shows the steps of the* GREEDY *algorithm on the graph shown in Figure 1. For the sake of simplicity, we refer to the supernode formed due to merging nodes x and y as the concatenated string "xy". In the first step, we merge the pair (b, c), which has the highest cost reduction of .5 (since both b and c have two edges incident on them, each has a cost of 2, and supernode bc also has a cost of 2 because of a self-edge and a superedge to a; hence*

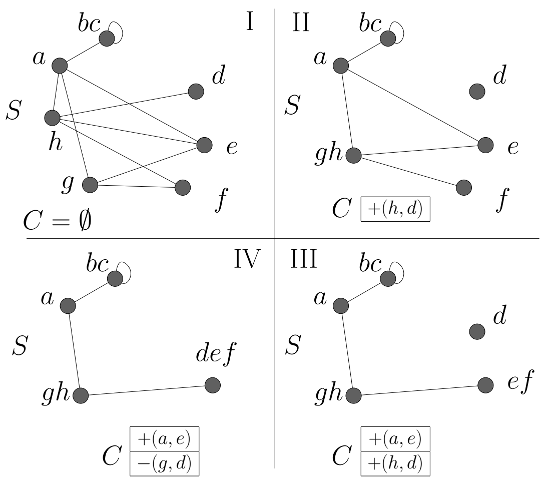

Figure 2: The steps (clockwise) for the Greedy algorithm. We show the summary S and the corrections C at the end of each step.

the cost reduction for (b, c) is .5). The other top contending pairs are (g, h) with an $s(\cdot)$ value of 3/7 and (e, f) with $s(e, f) = 2/5$. To see why $s(g, h) = 3/7$, lets derive the costs of nodes g and h before and after they are merged. Nodes g and h have 3 and 4 incident edges, respectively, and so $c_g = 3$ and $c_h = 4$. After g and h are merged to form supernode gh, we will have 3 superedges between supernode gh and nodes a, e and f, and one correction $+(h, d)$. Thus, $c_{gh} = 4$ and $s(g, h) = (c_g + c_h - c_{gh})/(c_g + c_h) = 3/7$.

In the next 3 steps, we merge the pairs (g, h) with cost reduction 3/7, (e, f) with cost reduction 1/3 (since $c_e = 2$, $c_f = 1$, and $c_{ef} = 2$ because of a superedge between ef and gh, and a correction $+(a, e)$), and (d, ef) with cost reduction 0. Note that the last merge does not decrease the cost, but only reduces the number of supernodes in the summary S resulting in a more compact visualization. After these merges, the cost reduction is negative for all pairs, and so Greedy *terminates.*

During the Output phase, we create the summary edges and correction entries for all the pairs (u, v) of neighbor supernodes in V_S. Recall that the superedge (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 $R = (S, C)$.

Time Complexity. During every merge step in Greedy, we look at each neighbor x of the supernode w, and recompute the costs of all the pairs containing x. The number of such pairs is roughly equal to the number of nodes that are at most 2-hops away from x (lets 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 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 $|H| = n \cdot 2\text{Hop}(w)$. Hence, the total time complexity of each merge step is O(4Hop(w) + 3Hop(w)$\cdot$($\log n + \log$ 2Hop(w))). Assuming an average degree of $d_{av} \leq n$ for each node, this becomes $O(d_{av}^3(d_{av} + \log n + \log d_{av}))$.

Optimizations. In our experiments, we found that, due to its large size, the time to update the heap structure dominates the running time. We can reduce this processing time by running the Greedy algorithm in rounds as follows. Suppose we fix a threshold τ and only process the node pairs whose cost reduction $s(\cdot) > \tau$. Thus, the heap H will only contain the pairs with cost reduction greater than the threshold, but we will still process these pairs in the same order as Greedy. Now, starting with the first round with a high value of τ (say .5), we run this thresholded Greedy procedure and keep reducing τ in successive rounds until in the final round τ is equal to zero. This reduces the size of the heap considerably, and allows us to avoid entire heap operations for most of the pairs with low cost reductions that are initially inserted into the heap but never merged anyways. Observe that at the start of every round, we also have to run the Initialization phase, which would process the entire graph; hence having too many rounds can also slow down the overall processing. In our experiments, we found that starting with $\tau = .25$ and reducing it by .05 in subsequent rounds gives good results. Note that such processing in rounds will produce strictly the same output as the (original) Greedy procedure since it merges node pairs in exactly the same order, but reduces the running time drastically due to faster heap operations.

3.2 The Randomized Algorithm

We will next describe our Randomized scheme which is a very light-weight randomized merging procedure. The motivation for this scheme is to trade off the cost of the computed summary for reduced computational complexity. Greedy has a high running time because it updates the cost reductions for all node pairs in the 3-hop neighborhood of the merged pair. These updates cannot be avoided in Greedy because it chooses the (globally) best pair to merge in each step, and any of the updated pairs could potentially be the best pair. To reduce computational complexity, in Randomized, instead of merging the globally best pair of nodes, we randomly select a node and merge it with the *best* node in its 2-hop neighborhood. This makes it much faster than Greedy and enables it to scale to very large input graphs. However, the representation R returned by Randomized may not be as compact as Greedy. Note that Randomized does not require any heap structure, which makes the merge operations considerably faster than Greedy.

The Randomized algorithm (Algorithm 2) iteratively merges nodes to form a set of supernodes V_S; these supernodes are divided into two categories, U (unfinished) and F (finished). The finished category tracks the nodes which do not give any cost reduction with any other node (that is, $s(\cdot)$ value is negative for all pairs containing them), while the unfinished category contains the remaining nodes that are considered for merging by the Randomized algorithm. Initially, all the nodes are in U. In each step, we choose a node u uniformly at random from U, and find the node v such that $s(u, v)$ is the largest among all pairs containing u. If merging these nodes gives a positive cost reduction, we merge them into a supernode w. We then remove u and v from V_S (and U), and add w to V_S (and U). However, if $s(u, v)$ is negative, we know that merging u with any other

Algorithm 2 RANDOMIZED(G)

```
1: $U = V_S = V_G$; $F = \emptyset$;
2: while $U \neq \emptyset$ do
3:     Pick a node $u$ randomly from $U$;
4:     Find the node $v$ with the largest value of $s(u, v)$ within 2 hops of $u$;
5:     if $(s(u, v) > 0)$ then
6:         $w = u \cup v$;
7:         $U = U - \{u, v\} \cup \{w\}$;
8:         $V_S = V_S - \{u, v\} \cup \{w\}$;
9:     else
10:        Remove $u$ from $U$ and put it in $F$;
11:    end if
12: end while
13: /* Output phase is same as GREEDY */
```

node will only increase the cost; hence, we should not consider it for merging anymore and so we move u to F. We repeat these steps until all the nodes are in F. Finally, the graph summary and corrections are constructed from V_S, similar to the GREEDY algorithm.

Time Complexity. In each merge step, we compute the cost reductions for all the 2Hop(v) pairs containing v; this requires a total of O(3Hop(v)) time. Again, assuming an average degree of d_{av}, this becomes $O(d_{av}^3)$. This reduced time complexity makes RANDOMIZED much faster than GREEDY in practice.

4. COMPUTING ϵ-REPRESENTATION

In this section, we will present two algorithms to compute a low-cost ϵ-representation R_ϵ. The first algorithm, called APXMDL, takes the (exact) MDL representation (computed in Section 3), and deletes correction and summary edges while still satisfying the approximation guarantee. In the second algorithm, called APXGREEDY, we build the ϵ-representation directly from the original graph, keeping in mind the ϵ constraint at each step.

4.1 The APXMDL Algorithm

Let us consider the representation $R = (S, C)$ constructed by one of the algorithms in Section 3. In APXMDL (Algorithm 3), starting with R, we compute an ϵ-representation R_ϵ of reduced size by throwing away edges from C and S that do not violate the approximation guarantees of Equation (1) for any node $v \in G$. The algorithm can be divided into two steps, (1) to remove unwanted edge-corrections from C, and (2) to remove edges from S. We first describe step (1) (Algorithm 3, lines 1-3). However, to give the intuition behind our approach, we first look at a similar problem on the original graph.

Given a graph G, suppose we construct a new approximate graph G' with $V_{G'} = V_G$ and $E_{G'} \subseteq E_G$, such that (a) the neighbor set N'_v for every node v in G' satisfies Equation (1), and (b) G' has the minimum size (number of edges). Then, one possible strategy is to find the set $M \subseteq E_G$ of maximum size such that removing M from the graph does not violate the approximation guarantee for any node. Further, if we denote the degree of a node v in G as n_v, then we know that at most $\lfloor \epsilon n_v \rfloor$ edges incident on v can be present in M. We observe that this problem is

Algorithm 3 APXMDL($G, R = (S, C)$)

```
1: Construct a graph $H$, with $V_H = V_G$ and $E_H = C$;
2: Compute the maximum $b$-matching $M$ for $H$ with $b_v = \lfloor \epsilon n_v \rfloor$;
3: $S_\epsilon = S$; $C_\epsilon = C - M$;
4: for all superedges $(u, v) \in S_\epsilon$ (in order of increasing $|\Pi_{uv}|$ value) having no entry in $C_\epsilon$ do
5:     if removing $(u, v)$ does not violate $\epsilon$ guarantee for any node in $A_u \cup A_v$ then
6:         Remove the edge $(u, v)$ from $S_\epsilon$;
7:     end if
8: end for
9: return $R_\epsilon = (S_\epsilon, C_\epsilon)$;
```

same as the *maximum b-matching problem*, defined as follows: Given a vector $b = \{b_1, b_2, ..., b_{|V_G|}\}$, find the largest set $M \subseteq E_G$ (called a b-matching) s.t. the number of edges in M incident on the node v is at most b_v. It is easy to see that with $b_v = \lfloor \epsilon n_v \rfloor$, constructing the graph G' is equivalent to finding the maximum b-matching M in G. (When all $b_v = 1$ ($\epsilon n_v = 1$), then this reduces to the standard *maximum matching problem.*) The b-matching problem can be solved in $O(m \cdot \min\{m \log n, n^2\})$ time using Gabow's algorithm [10] – here n in the number of nodes and m is the number of edges in G.

In our setting, we will remove the corrections in C by converting it into an instance of the b-matching problem. We construct a new graph H with $V_H = V_G$ and with the set of edges present in C. Specifically, for any (positive or negative) edge correction $(a, b) \in C$, we add an edge between nodes a and b in H. Now, we set $b_v = \lfloor \epsilon n_v \rfloor$ ($n_v = |N_v|$ is the number of neighbors of v in G). The b-matching M in this new graph H corresponds to the maximum number of edge corrections in C that can be removed without violating the approximation guarantees. This is because each edge correction contributes an error of 1 to the neighbor sets of its two endpoints. We remove all the corrections in M from C to get C_ϵ. Note that the graph H is typically much smaller than the original graph, since it has much fewer edges (strictly less than $|E_G|$).

In our experiments, we found that the corrections C typically constitute a major portion (70-80%) of the representation R. Thus, reducing the corrections from C to C_ϵ already gets us substantial savings in space. We now describe step (2) of APXMDL (Algorithm 3, lines 4-8), which is to reduce the size of the graph summary S. Interestingly, unlike for the corrections, finding the best superedges to remove from S turns out to be very hard. The problem is that removing a superedge corresponds to a bulk removal of all the edges incident on the nodes contained in the corresponding supernodes. Furthermore, since each node has a different constraint, based on its original degree and the number of corrections already removed, deleting the superedge may violate the approximation guarantee for a subset of nodes in the supernodes. Due to these reasons, removal of edges in S does not map cleanly to an instance of the b-matching problem.

To remove edges from summary S, we instead apply a simple greedy approach. For each superedge $(u, v) \in E_S$, we can find out whether removing it will violate the ϵ-constraint of any node in $A_u \cup A_v$ (it cannot obviously violate constraints on any other node). If deleting edges in Π_{uv} do not violate any node neighborhood constraints, then the superedge can be removed from S. We make a pass through all the superedges in E_S in order of increasing $|\Pi_{uv}|$, and keep removing edges whose removal do not violate any constraint. Our rationale for considering superedges in increasing order of $|\Pi_{uv}|$ is that this is a good estimate of the total error that will be introduced when superedge (u, v) is removed, and so we pick the superedge that will introduce the least extra error at every step. Note that when we decide to remove a superedge from S there cannot be any corrections for that superedge in C_ϵ. This is because if there was such a correction (a, b), then the error at node a or b must have already reached its maximum value ($\lfloor \epsilon n_a \rfloor$ or $\lfloor \epsilon n_b \rfloor$), otherwise we would have removed (a, b) while processing the corrections. Using this observation, we can further reduce the computation time by only checking superedges for removal that have no corrections in C_ϵ.

Time Complexity. In ApxMdl (Algorithm 3), we first process the corrections and then the summary. Due to [10], we get that the time complexity of the first step is $O(|C|^2 \cdot \log |V_G|)$. Further, the second step requires $O(|E_S| \cdot |V_G|)$ time in the worst case to check for every superedge whether deleting it will violate the error constraint of any node in the two supernodes it is incident on.

4.2 The ApxGreedy Algorithm

In the previous section, we described how to compute the ϵ-representation from a given (exact) MDL representation. The main advantage of our ApxMdl scheme is that during a majority of the processing (to compute the MDL representation), the algorithm is oblivious of ϵ; thus, it provides a flexible and light-weight method to compute approximate representations for different ϵ values. However, for exactly the same reason (being oblivious to ϵ), ApxMdl fails to take the maximum advantage of the leeway provided by the approximation constraints.

In our next scheme, called ApxGreedy, we compute the approximate representation starting with the original graph itself, while keeping in mind the approximation guarantees. The steps of ApxGreedy are exactly the same as Greedy, the only difference is in the way we compute the node costs c_v. Basically, we exploit the knowledge of ϵ to compute the (new) cost c'_v of approximately representing a node v, where we only count the minimum number of edges in S and C that are required to satisfy v's ϵ-constraint. Let us first look at an example of how this cost is different from what we computed in the previous section.

Example 4. *Consider again the graph in Figure 1. For $\epsilon = 1/3$, the new cost for the node h is $c'_h = 3$ (instead of $c_h = 4$), because we can remove up to 1 incident edge on h and still satisfy the ϵ-constraint. Furthermore, for supernode y, the new cost will be $c'_y = 2$ (down from $c_y = 3$) since the correction $-(g, d)$ in C can be deleted, as it does not violate ϵ-constraint of g or d. On the other hand, any edge deletion from f or w will violate the ϵ-constraint of these nodes (in the case of supernode w, the constraint of the nodes b and c contained in it); hence, the costs remain 2 for both of them, the same as before.*

In general, the cost c'_v of approximately representing a supernode v is the minimum total cost of representing (a subset of) its graph edges while satisfying the approximation guarantee of all nodes in A_v. The cost reduction $s'(u, v)$ when two nodes u and v are merged into a new supernode w is then given as before by $(c'_u + c'_v - c'_w)/(c'_u + c'_v)$; thus $s'(u, v)$ is the difference in the cost of approximately representing nodes u and v when separate, and when merged together. In our ApxGreedy procedure, we run exactly the same steps as the Greedy algorithm, but using this new cost reduction value $s'(\cdot)$ instead of $s(\cdot)$ to select the next node pair to merge.

Now, lets look at how to compute the cost c'_v for a supernode v. The complication here is that deleting a correction edge (a, b) affects the ϵ-constraints of both nodes a and b; thus, when deleting correction edges that are incident on nodes in A_v, we also need to consider the impact it has on the error constraints of v's neighbors. However, to ensure that c'_v can be computed efficiently, we will ignore the ϵ-constraints of v's neighbors when deciding which correction edges incident on nodes in A_v can be deleted. While this will result in a slight underestimate for cost c'_v, it will help to speed up its computation considerably.

Recall that the cost c_v is the cost of exactly representing all the graph edges incident on nodes in A_v. Our strategy for computing c'_v for a supernode v is to deduct from its exact cost c_v the number of correction edges incident on nodes A_v that can be deleted without violating their ϵ-constraints. So if e_a represents the number of correction edges that can be deleted for node $a \in A_v$, then $c'_v = c_v - \sum_{a \in A_v} e_a$. Clearly, $e_a \leq \epsilon n_a$ since deleting more than ϵn_a correction edges will cause node a's constraint to be violated. Thus, e_a is the minimum of ϵn_a and the total number of correction edges involving node a.

It is important to observe that we do not actually remove the unwanted edges during ApxGreedy, since to remove them we have to also consider constraints on the other (neighboring) nodes. When ApxGreedy finishes, we have the set of supernodes V_S which we use to compute an exact MDL representation R similar to Greedy. We then run the ApxMdl algorithm on R to get the final approximate representation. The key difference here is that ApxGreedy takes into account the ϵ-constraints when computing the exact representation R that is fed to ApxMdl.

5. EXPERIMENTS

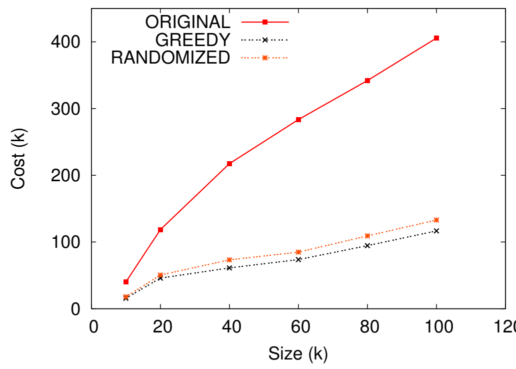

Figure 3: Comparison of costs of representations computed by Greedy and Randomized on the CNR dataset.

We now present the experimental evaluation of our tech-

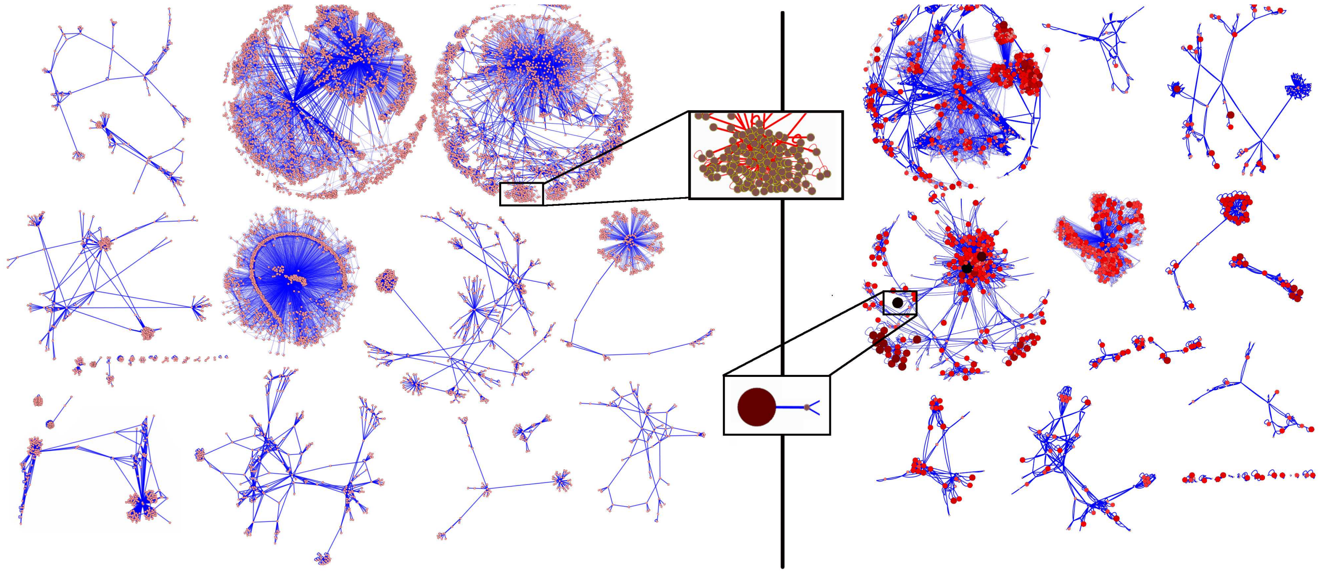

Figure 6: Visual comparison of CNR-$10k$ graph (left) and its summary (right) S computed by GREEDY. The size of a supernode in S is proportional to the number of nodes included in it. The dataset contains a bipartite subgraph (shown in middle) with a large set of nodes connecting to a single other node, which is hard to identify in the original graph; however, these nodes are condensed into a single (large) supernode in the summary that distinctly stands out.

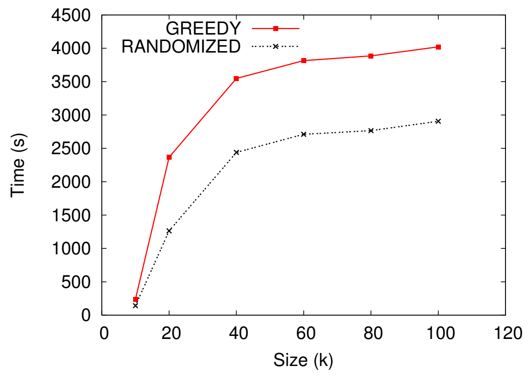

Figure 4: Comparison of running times of GREEDY and RANDOMIZED on the CNR dataset.

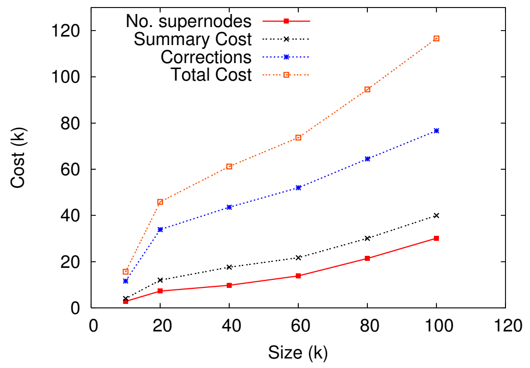

Figure 5: Breakup of the cost of representation of the CNR dataset.

niques. The goal of these experiments is to evaluate our techniques in the following three objectives. First (Section 5.1), to study the compression quality and anatomy of the representation R and also evaluate the effectiveness of S as a compact summary highlighting important trends. Second, we compare our algorithms with existing graph compression techniques (Section 5.2). And finally (Section 5.3), we discuss the performance of the approximate representation.

We first briefly explain the experimental set-up. We have implemented the proposed algorithms for finding both the exact (GREEDY and RANDOMIZED) and approximate (APXMDL) MDL representations. We ran the GREEDY algorithm in rounds, starting with $\tau = .25$ and reducing it by .05 in subsequent rounds. The results for RANDOMIZED are averaged over 10 seeds for the random number generator (although we saw very little variation in both cost and running times for different seeds). All the experiments involving running times were run on a Linux server with Intel Core-2 Duo processor and 2 GB RAM. We compare our techniques against the following existing algorithms.

- **Reference Encoding (REF) [3]:** This has been a very successful and popular technique for web-graph compression. It basically consists of two logical parts, the first of which reduces the size of the neighbor lists for each node, and the second generates compressed representations of these lists using complex bit-level encodings. For a fair comparison, we disabled all the bit-level encodings while running this algorithm. It should be noted that the same encodings can be used to represent our graph and correction summaries, however, that comparison is not considered as it is not the main focus of this study.

- **Graclus (GRAC) [8]:** This is a graph clustering algorithm that divides the nodes of a given weighted graph into clusters such that the sum of weights of the *inter-cluster* edges is minimized. We ran it on a new derived graph having the same set of nodes as G, while edges exist between any two nodes with a non-zero cost reduction and the weight on the edge equal to the cost reduction $s(\cdot)$. Although GRAC does not explicitly focus on reducing the representation cost, by setting weights same as cost reductions we ensure that it also tries to minimize the cost. We compute the cost of representation by creating supernodes from the clusters generated by GRAC, and then summing costs of the superedges between these supernodes. GRAC also requires the number of clusters (k) as an input, which we vary in the range $\pm10\%$ of the number of supernodes returned by GREEDY; the results shown are for the value of k that gave the maximum compression.

- **Sampling (SAMP):** We used a simple edge sampling scheme to compare against the approximate MDL representation. In this scheme, we chose a fixed number M (equal to the cost of the approximate representation) of edges uniformly at random from the input graph. The (sub-) graph induced by these M edges is taken as an approximation to the original graph, which is compared against the approximate representation computed by APXMDL.

In our experiments, we used the following datasets to evaluate the compression ratio and running times of various approaches.

- **CNR dataset**[1]**:** This web-graph dataset was extracted from a crawl of the CNR domain. We replaced each directed edge by an undirected edge. To view the variation of running time and compression ratio, we also ran experiments with subgraphs of this dataset. Specifically, the dataset CNR-x is the subgraph induced by the node indices $[0, x)$ (e.g. CNR-5k has node indices from 0 to 4999 along with all their edges to each other). The largest dataset, CNR-100k has 100k nodes and 405, 586 edges.

- **RouteView**[2]**:** This is a graph that represents the autonomous system topology of the Internet. Here each node is an autonomous system, and two nodes are connected by an edge if there is at least one physical link between them. This dataset is collected by the University of Oregon Route Views Project, and consists of about 10, 000 nodes and 21, 000 edges.

- **WordNet**[3]**:** WordNet is a large lexical database of English words often used in natural language processing applications. We extract a graph from the data where nodes correspond to English words, and an edge (u, v) exists if u is a hypernym, entailment, meronym, or attribute of v, or if u is similar or causal to v (or vice-versa). This graph has 76, 853 nodes and 121, 307 edges.

- **Facebook**[4]**:** This dataset was extracted in 2005 from a crawl of the Cornell University community of the Facebook social networking website, and consists of 14, 562 nodes and 601, 735 edges. Here, nodes are profiles of students at Cornell, and an edge exists between two students who are friends.

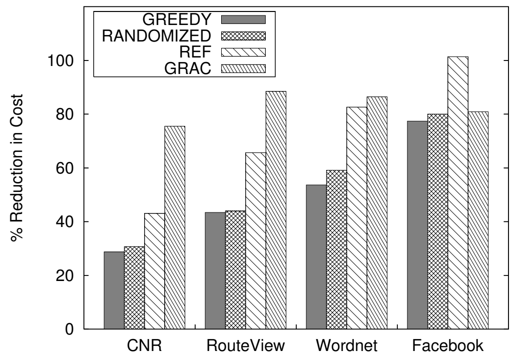

Figure 7: Comparison of graph compression algorithms. % Compression is defined as the ratio of the summary cost and the original cost (lesser % implies better compression). Clearly, GREEDY beats all other algorithms.

5.1 Analysis of MDL Representations

We first study the quality of compression of the two schemes. In Figure 3, we plot the cost of representation produced by GREEDY and RANDOMIZED, with varying size of the CNR graph. Our results show that GREEDY gives the best compression, consistently computing representations with cost roughly 10% lower than RANDOMIZED. Next, in Figure 4, we compare the running times of these schemes for different graph sizes. Observe that as predicted, RANDOMIZED is much faster than GREEDY, finishing in about half the time required for the latter on the same graph. This gives a clear trade-off between the two: the user should use GREEDY to get the best compression, while RANDOMIZED if he wants to compress the graph quickly, with comparable compression.

In our next experiment (Figure 5), we show the breakup of the cost of the representation. The representation has three kinds of entries, namely supernodes, superedges, and corrections. We plot the number of these entries as we vary the size of the dataset. Notice that the size of corrections is the dominant factor in the cost of the representation; only about 20% of the representation cost is due to the summary.

The small sizes of superedges and supernodes (about 10% of the original graph) indicates the usefulness of the summary for visualization and trend analysis. Figure 6 shows a visualization (constructed using the Cytoscape tool [29]) of the original input and the corresponding summary S for the CNR-10k dataset. Apart from being much smaller in size and hence less cluttered, there are many interesting patters that stand out. One such example is shown in the middle zoom-in boxes, where a large bipartite subgraph is extracted in the summary. In the original graph, this subgraph is barely visible as a cluster of nodes surrounding (and connected to) one node in the center; while in the summary it is extracted as a large supernode (displayed in the middle) connected to just one other node via a superedge.

5.2 Comparison with Graph Compression

We now compare our techniques against Reference Encoding (REF) and Graclus (GRAC). We present the cost reduc-

[1]Laboratory for Web Algorithmics (http://law.dsi.unimi.it/).
[2]http://www.routeviews.org/
[3]http://vlado.fmf.uni-lj.si/pub/networks/data/
[4]http://www.facebook.com/

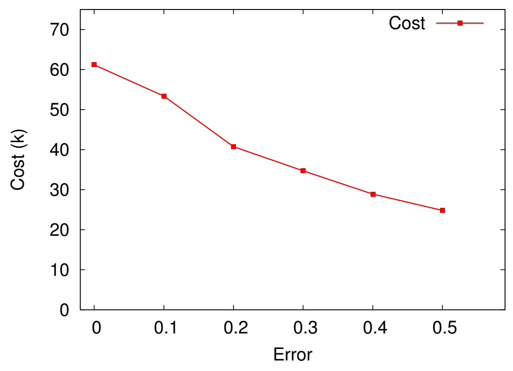

Figure 8: Evaluation of approximate representation computed by ApxMdl for CNR-$40k$. Cost reduces almost linearly as we increase the value of ϵ.

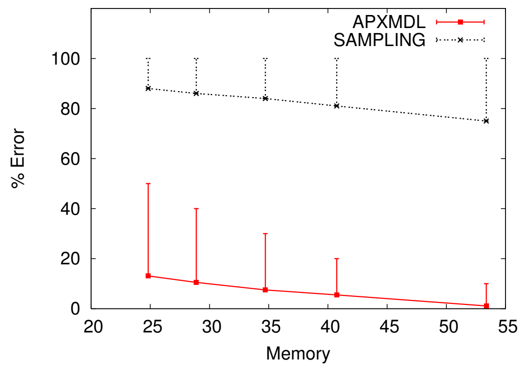

Figure 9: Comparison of R_ϵ and SAMP.

tion of various schemes in Figure 7, where y-axis shows the cost of the compressed representation as the percentage of the original cost (lower is better). Clearly, Greedy obtains the highest compression among all the schemes, especially for the RouteView and CNR datasets (here CNR refers to the CNR-$100k$ dataset), where its compression ratio is more than twice that of REF or GRAC. Randomized also performs better than REF and GRAC on all datasets.

Notice that on the Facebook dataset, no scheme gets better than 80% compression. We believe the reason is that it is not possible to compress this dataset much further using graph compression techniques. This is because although users form communities in social networks, their variety of individual tastes makes their friend-lists (neighbor set) sufficiently different from those of other users, allowing for lesser commonality than, for example, web-graphs. Among other schemes, REF mostly gets good compression on the *compressible* graphs such as CNR and RouteView, but performs even worse than Graclus on the Facebook dataset. We believe the reason for this is that the techniques of REF are tailored for finding nodes with similar neighbor lists among nodes with neighboring indices (matching urls), which are of course not present in Facebook.

5.3 Approximate MDL Representations

We now discuss the effectiveness of the approximate representations in reducing the cost. We ran the ApxMdl algorithm on a fixed dataset (CNR-$40k$), and varied the value of ϵ in the range $[0, .5]$. Figure 8 shows the cost of the approximate representation for different values of ϵ. Note that with $\epsilon = 0$, the cost is same as the exact MDL representation. However as ϵ is increased, the cost reduces almost linearly, down to almost 50% of the exact MDL when $\epsilon = .5$. Unfortunately, due to time constraints we were not able to implement the ApxGreedy algorithm. All results shown here are for ApxMdl .

The next study details the comparison of ApxMdl with the random sampling scheme (SAMP). In Figure 9, we show the % errors, defined as the ratio of $error(v)$ (Equation 1) and $|N_v|$, in neighbor sets of the nodes of the reconstructed graph, for R_ϵ and SAMP. The x-axis shows the costs of the approximate representation, while y axis plots the average (shown as the line) and maximum (shown as the error-bars) % errors. For a fair comparison, we fixed the sample size in SAMP to be same as the cost of R_ϵ. As expected, SAMP does not guarantee any bound on the maximum error, which is 100% in almost every case. Moreover, even the average error of ApxMdl is considerably lower than that of SAMP.

6. CONCLUSIONS AND FUTURE WORKS

In this paper, we have presented a highly compact two-part representation $R(S, C)$ of the input graph G based on the MDL principle. In this representation, S is an aggregated graph structure that gives a high level *graph summary* of G, and C is a set of edge corrections using which one can recreate G. We have shown how to compute representations that allow for both lossless and lossy reconstruction of graphs with bounds on the introduced error. We have also presented algorithms to compute these representations with the minimum cost, and shown their effectiveness in compressing the input graph through experimental evaluation on multiple real life graph datasets.

As for future work, we noticed that in many real life graphs, the edges and nodes also contain various attributes. For example, nodes in webgraphs have urls, edges (transactions) in market basket data have weights (monetary value), packets in IP traffic have multiple attributes such as port-numbers and type of traffic, etc. We would like to investigate if our two-part representation can be extended to compress graphs containing node and edge attributes.

We have presented two heuristics in this paper, called Greedy and Randomized, which perform very well in practice, effectively beating every other algorithm that we compared with. Proving either the optimality of these algorithms, or conversely a hardness result on computing the minimum cost representation, are definitely other interesting areas of future research.

7. REFERENCES

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

[2] V. D. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. V. Dooren. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Rev.*, 46(4):647–666, 2004.

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

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, pages 107–117, 1998.

[5] D. Chakrabarti. Autopart: Parameter-free graph partitioning and outlier detection. In *PKDD*, pages 112–124, 2004.

[6] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *VLDB*, pages 111–122, 2000.

[7] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, pages 647–651, 2003.

[8] I. Dhillon, Y. Guan, and B. Kulis. A fast kernel-based multilevel algorithm for graph clustering. In *KDD*, pages 629–634, 2005.

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

[10] H. N. Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In *STOC*, pages 448–456, 1983.

[11] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[12] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *VLDB*, pages 721–732, 2005.

[13] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[14] 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.

[15] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, pages 174–185, 1999.

[16] H. V. Jagadish, J. Madar, and R. T. Ng. Semantic compression and pattern extraction with fascicles. In *VLDB*, pages 186–198, 1999.

[17] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[18] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, 2002.

[19] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. In *Journal of the ACM.*, volume 46, pages 604–632, 1999.

[20] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large-scale knowledge bases from the web. In *VLDB*, pages 639–650, 1999.

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

[22] L. V. S. Lakshmanan, R. T. Ng, C. X. Wang, X. Zhou, and T. Johnson. The generalized mdl approach for summarization. In *VLDB*, pages 766–777, 2002.

[23] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[24] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *NIPS*, pages 849–856, 2001.

[25] N. Polyzotis and M. Garofalakis. Xsketch synopses for XML data graphs. *TODS*, 31(3):1014–1063, 2006.

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

[27] 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.

[28] J. Rissanen. Modelling by the shortest data description. *Automatica*, 14:465–471, 1978.

[29] P. Shannon, A. Markiel, O. Ozier, N. Baliga, J. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Res.*, 13(11):2498–504, 2003.

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

[31] 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.