GConnect: A Connectivity Index for Massive Disk-Resident Graphs

Charu Aggarwal
IBM T. J. Watson Research Center
Hawthorne, NY
charu@us.ibm.com

Yan Xie
University of Illinois at Chicago
Chicago, IL
yxie@cs.uic.edu

Philip S. Yu
University of Illinois at Chicago
Chicago, IL
psyu@cs.uic.edu

ABSTRACT

The problem of connectivity is an extremely important one in the context of massive graphs. In many large communication networks, social networks and other graphs, it is desirable to determine the minimum-cut between any pair of nodes. The problem is well solved in the classical literature, since it is related to the maximum-flow problem, which is efficiently solvable. However, large graphs may often be disk resident, and such graphs cannot be efficiently processed for connectivity queries. This is because the minimum-cut problem is typically solved with the use of a variety of combinatorial and flow-based techniques which require random access to the underlying edges in the graph.

In this paper, we propose to develop a connectivity index for massive-disk resident graphs. We will use an edge-sampling based approach to create compressed representations of the underlying graphs. Since these compressed representations can be held in main memory, they can be used to derive efficient approximations for the minimum-cut problem. These compressed representations are then organized into a disk-resident index structure. We present experimental results which show that the resulting approach provides between two and three orders of magnitude more efficient query processing than a disk-resident approach at the expense of a small amount of accuracy.

1. INTRODUCTION

The problem of managing and mining graph data has seen renewed interest in recent years because of the increased interest in a number of structural applications such as chemical data, biological data, XML data, and computer networks. A number of important data mining and management algorithms have recently been explored in the context of graph data [1, 15, 16, 17, 18, 19]. Detailed surveys on graph mining algorithms may be found in [4].

In many application domains such as the web and computer networks, the underlying graphs are so large that they cannot be stored in main memory, but may need to be stored onto disk [6, 7, 8, 11]. For example, a typical communication network may have millions of nodes, and the number of edges are several orders of magnitude greater. The web-graph [7, 8] is even larger, and may require specialized hardware to enable storage of the underlying graph structures. In such cases, it is necessary to design efficient and effective methods for indexing and retrieval. Many algorithms which are effective for the case of memory-resident data are no longer effective in this scenario. Some recent techniques have been designed [17, 18] for the case of disk resident graphs for query processing and indexing.

In this paper, we will study the minimum connectivity problem for massive disk-resident graphs. For a given pair of nodes s and t, it is desirable to determine the minimum number of edges required in order to disconnect the graph into two components such one of them contains s and the other contains t. This problem is identical to that of finding a minimum-cut between the pair s and t. This problem is particularly useful in diagnosing survivability in large communication networks. For example, the solution to the problem can be used to determine the number and identity of the minimum number of links which can be used to disconnect the communication network between a given pair of nodes. It can also be used to determine and isolate dense regions in large connected graphs.

The minimum-cut problem has been widely studied in the literature [2], and is well known to be the mathematical dual of the maximum flow problem. A variety of very efficient algorithms exist for solving the maximum flow problem in the memory-resident case. The best algorithms for the problem can solve it in (slightly worse than) $O(n \cdot m)$ time, where n is the number of nodes, and m is the number of edges. However, most of the existing techniques [2, 3] are designed with the implicit assumption that the underlying graphs are memory-resident. The problem is significantly more difficult for disk-resident graphs. This is because most of the existing algorithms for this problem use combinatorial or flow-based techniques which can access the edges in the underlying graph in arbitrary order. Each operation such as updating the flows or node labels will lead to a random access on disk. This leads to impractical running times for a problem which is easily solvable in the memory-resident case. Furthermore, a user may repeatedly query the graph over different source-sink pairs, and may therefore expect online response times. In general, our goal is to create *an index which can provide extremely efficient responses even for very large graphs.*

One possible solution to this problem is to pre-store the minimum-cut between every pair of vertices. However, this is not a practical solution to the problem. This is because for large graphs, the number of possible source-sink pairs may be too large to store the underlying cut values explicitly. For example, for a graph containing 10^6 nodes, the number of possible node pairs will be over 10^{11}. The detailed information for the corresponding cuts may require storage in the tera-byte order. This level of pre-storage may not be available on most platforms today. Furthermore, since the size of many graph data sets encountered in applications continue to increase over time, it is critical to have an approach which scales well with the size of the underlying data set.

In order to achieve this goal, we will design a *disk-based query index for connectivity queries*. We will use a randomized contraction approach in order to create compressed representations of the underlying graph. Repeated probabilistic contractions are used in order to create different compressions of the overall graph. These compressions are used to create an index which provides high-quality responses. The idea behind *repeated storage of probabilistically compressed data* is to ensure that each of the compressed graphs is small enough to handle effectively with the use of memory-resident algorithms. At the same time, the repetition in the storage is leveraged during query processing in order to provide robust responses. We will show that such an approach is extremely effective in creating an index for connectivity queries. At the same time, the approach retains its efficiency, because it can be decomposed into multiple problems of smaller size, rather than one graph of very large size. The savings in solving smaller combinatorial problems are far greater than the overhead of having to solve them multiple times.

This paper is organized as follows. In the next section, we will discuss the overall approach for creating the index for connectivity queries. We will discuss the details for constructing the index and then show how to use it for query resolution. In section 3, we will provide an analysis of the approach. The experimental results are discussed in section 4. Section 5 contains the conclusions and summary.

2. GCONNECT: THE CONNECTIVITY INDEX

In this section, we will discuss the method for creating the connectivity index. Before describing the algorithm for creating the index, we will introduce some notations and definitions. We assume that the connectivity queries are designed for a large graph with node set N and edge set A. For simplicity of discussion, we assume undirected graphs, though the approach discussed in this paper can be extended to the case of directed graphs. The number of nodes in the set N is n. We assume that the size of n is quite large, though it is typically dwarfed by the number of edges. Such large graphs can be stored only on disks. Typical maximum flow or minimum-cut algorithms cannot be efficiently applied to such cases, since the maximum-flow based methods [2] use combinatorial or flow-based techniques, which may require random access to edge information on disk. Such random access may cause so much performance deterioration, that it may become impractical to use these approaches effectively for disk-resident data. Furthermore, it is impossible to store pairwise information on minimum *s-t* cuts effectively, since the number of possible pairs may be too large to store effectively.

A natural solution to this problem is to replace a *single large problem*, with multiple-smaller approximations which can be efficiently solved. The solutions from the different approximations can be combined in order to create a single robust solution. We will use a probabilistic compression approach in order to create the connectivity index. In order to design the connectivity index, we will use some contraction techniques which are also used in [13] for the memory-resident case. While the contraction ideas discussed in [13] are useful for finding a *global* minimum-cut in a graph, they are not very useful for determining the minimum *s-t* cut between an *arbitrary pair* of vertices. Furthermore, we are interested in creating an *index* which allows responses to arbitrary source-sink pairs in *online response times.* In this paper, we will show how to adapt some of the broad ideas in this approach effectively in order to create an *efficient connectivity index.* We will also provide a theoretical analysis of the effectiveness of such an index. This also allows the index to provide an estimate of the accuracy of the cut-based approach. While the accuracy of the minimum connectivity estimation is different for different pairs of vertices, we will see that the estimation is particularly effective for the case of vertex pairs in which the value of the underlying minimum cuts is low. Such cuts are also the most critical in network connectivity applications, because they could lead to network disconnection by the removal of a few edges.

The probabilistic compression approach creates *multiple compressed representations* of the underlying graph with the use of sampling. In order to create these representations, we need to sample edges from the underlying data. This is achieved by drawing on the concept of reservoir sampling. The compressed representations are used in order to create an index. The overall approach is to randomly sample the data in order to create *edge sampled and node sampled compressions from the underlying data.* We will first define edge-sampled compressions for creating compressed graphs:

DEFINITION 1. *Let G be a graph with node set N and edge set A. For any fraction in $f \in (0, 1)$, an edge-sampled compression $H(f)$ of a connected graph G is one in which we sample a fraction f of the edges of the graph. Let this sampled edge set be denoted by S. We contract each connected component induced by the edge set S into a single node. Then we use this contracted set of nodes in order to reconstruct the contracted graph $H(f)$ using the original edge set A, which is denoted by A' on the contracted node set. All self-loops (which will be created as a result of the contraction) from A' are removed.*

An example of graph coarsening process is illustrated in Figure 1. The set of edges sampled are $(1, 2)$, $(3, 4)$, $(5, 6)$ and $(5, 7)$ and are illustrated by bold lines. In this case the nodes within the dotted circles are compressed to supernodes. This is because the nodes within the dotted circles remain connected after sampling. We will see that edge-sampled compressions have the property that the resulting compressed graph is probabilistically biased towards retaining cuts of lower value from the original graph. This is essentially because edges in dense components are more likely to be sampled, and will eventually result in a contraction. This is also why contraction based techniques [13] work well for determining global minimum cuts. However, the techniques of [13] cannot be directly used for disk-resident cases or for determining minimum *s-t* cuts for a specific source or

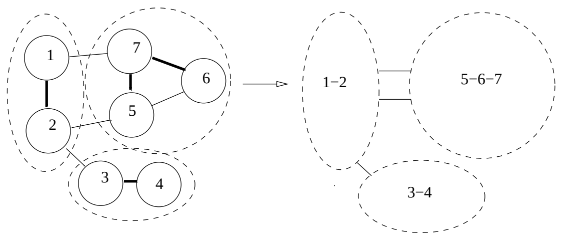

Figure 1: Illustration of Graph Coarsening

```
Algorithm GConnect(Graph: $G$,
    Edge-based Compressions: $k_e$,
    Node-based Compressions: $k_n$,
    Target Node Size: $m_t$);
begin
  Create $k_e$ edge-sampled compressions of graph $G$, each
    with target size $m_t$;
  Create $k_n$ node-sampled compressions of graph $G$, each
    with size $m_t$;
  Create inverted index on information for assignment
    of original nodes to node identifiers of
    compressed graphs;
end
```

Figure 2: Overall Approach for Constructing the Connectivity Index

sink node. Furthermore, the aim of this paper is to design an effective *index* for the disk-resident case. Next, we will define the concept of *node-sampled compressions.*

DEFINITION 2. *Let G be a graph with node set N and edge set A. For any integer m, we can define the node compressed graph $J(m)$ as follows. For each node, we map it to one of the integers from 1 through m by using an unbiased die. This is used in order to partition the nodes into m sets. Each of the m sets is then mapped into a contracted node. Then we use this contracted set of nodes in order to reconstruct the contracted graph $J(m)$ using the original edge set A, which is denoted by A' on the contracted node set. All self loops (which will be created as a result of the contraction) from A' are removed.*

The overall algorithm uses the parameters k_e and k_n in order to regulate the number of edge-compressed and node-compressed graphs. The number of edge-compressed graphs is denoted by k_e, and the number of node compressed graphs is denoted by k_n. In addition, we use a *target compression size m_t* which represents the target number of nodes in the compressed graphs. The optimal size of the compression is dependent upon the structure of the graph, and is therefore a hard problem. In general, we would like to pick a target compression size which is such that the resulting graph can be loaded in main memory. Note that the maximum size of the graph with m_t nodes is *at most* $O(m_t^2)$. Therefore, for a memory size of M, it is best to pick a target size which is at least equal to $m_t \geq \sqrt{M}$. For the case of the node compressed graphs, this target is met in a straightforward way by choosing a die with m_t sides. For the case of the node compressed graphs, this target is met in a straightforward way by choosing a die with m_t sides. In the case of edge-compressed graphs, this target is met by choosing the fraction f of the edges carefully. We will discuss more on this issue slightly later.

The algorithm proceeds in phases. In the first phase the algorithm constructs k_e edge-sampled graphs, each of which has a target size of m_t. Note that the definition of edge-sampled graphs is expressed as a function of the sampling fraction f rather than a target number of nodes. Therefore, the sampling fraction needs to be picked carefully in order to obtain the desired target number of nodes. We will discuss this issue slightly later. In the second phase, we create k_n node-sampled compressed graphs. In this case, we create m_t nodes in each node sampled graph. We note that each of the compressed graphs typically required much lower space as compared to the original graphs. This is because the compressed graph will contain many parallel edges between the compressed nodes. Such parallel edges can be compressed into a single edge with the appropriate weight. Furthermore, a large fraction of the edges in the compressed graph will be self-edges which can be eliminated. For example, in the case illustrated in Figure 1, the compressed graph on the right hand side has fewer edges than the original graph on the left hand side.

Once all the compressed graphs have been created, we store the mapping of the original nodes to the new compressed node identifiers in a way which facilitates easy query processing. In order to achieve this goal, we use an inverted mapping from the original node identifiers to the compressed node identifiers of the different compressed graphs. Note that since a total of $(k_n + k_e)$ compressions are created, we need to store the mapping information for $(k_n + k_e)$ different graph compressions. For the node i, let the corresponding node identifiers in the $h = (k_n + k_e)$ graphs be denoted $M(i, 1) \dots M(i, h)$. Thus, we denote the mapping of the ith node in the jth compression by $M(i, j)$. We note that i is an integer drawn from $\{1 \dots n\}$, j is an integer drawn from $\{1 \dots h\}$, and $M(i, j)$ is an integer drawn from $\{1 \dots m_t\}$.

The inverted index is defined as follows. For the ith node, we store the list of $h = (k_n + k_e)$ 2-tuples. Specifically, the list for the ith node contains the tuples $(1, M(i, 1)) \dots (h, M(i, h))$. Thus, the size of each list is $h = (k_n + k_e)$. The overall approach for constructing the index is illustrated in Figure 2.

Once the index has been created, it can be used very effectively for query processing. For any user-specified pair of nodes s and t as source-destination pairs, we access the inverted lists for s and t. These inverted lists provide us with the mapping to the node identifiers for the h different graphs. It remains to describe how the index is used for query processing. We will also describe how the compressed graphs are constructed by edge and node sampling. In a later section, will also provide an analysis as to why such an approach should be effective. Further, we need to design a way to implement this approach efficiently.

2.1 Creating the Indexed Representation from Compressed Graphs

In this section, we will discuss the process of creating the indexed representation from the compressed graphs. The actual process of creating the compressed graphs will be described slightly later. Let us assume that the total number of compressions is denoted by h. Then, for each of the h compressions, a given node i maps onto one of the m_t partitions. As discussed earlier the mapping of the ith node for

```
Algorithm QueryProcess(Inverted Representation: $I$,
 Source: $s$, Sink: $t$, Compressed Graphs: $H_1 \dots H_r$);
begin
    Use inverted representation to determine mapping from
      nodes $s$ and $t$ to corresponding nodes $M(s,r)$ and
      $M(t,r)$
    for each compressed graph with index $r \in \{1 \dots h\}$
     for which $M(s,r) \neq M(t,r)$ determine
       minimum-cut $v_r$ using efficient in-memory
       algorithms;
    report the minimum among all values of $v_r$
       and the corresponding cut;
end
```

Figure 3: Query Processing with the Connectivity Index

the jth compression is denoted by $M(i,j)$.

We create an inverted representation, in which each node i points to a list denoted by $(1, M(i,1)) \dots (h, M(i,h))$. Note that this inverted list provides the information necessary for determining the appropriate mappings to the sources and sink nodes in the compressed graphs for a given query. We also maintain a list of h pointers to the disk-resident data structures for the compressed graphs. The compressed graphs are represented in the form of adjacency lists. For each node, we maintain a list of their edges, along with the corresponding weight, which is also the number of parallel edges which consolidate into a single edge. We note that these adjacency lists are much smaller than those of the original graph. This is because many of the edges in the graph are removed during compression, and other edges were consolidated during the compression process. While the graphs are stored on disk during the index creation, it is assumed that each individual graph is small enough to be read into main memory for effective query processing. The additional space required by the inverted representation is $h \cdot n$, where n is the number of nodes and h is the number of compressions. While this can be quite large, we will see that only two of these lists are accessed for a given query. This ensures that a given query can be resolved very efficiently.

2.2 Query Processing from the Connectivity Index

In this section, we will discuss the techniques for query-processing with the use of the connectivity index. The process of querying with the index is relatively straightforward. For a given query-node pair (s,t), we first use the inverted representation in order to determine all the corresponding source-sink pairs in the compressed representation. These correspond to $(M(s,1), M(t,1)) \dots (M(s,h), M(t,h))$ in the compressed representation. For a given graph index j, we note that the pair $(M(s,j), M(t,j))$ represents the source-sink pair in the compressed representation. We note that we need to use only those pairs $(M(s,j), M(t,j))$ for which we have $M(s,j) \neq M(t,j)$. We note that for a given source-sink pair, we need to access only two inverted lists in order to determine the identity of the compressed graphs for which the source and sink do not map onto the same node.

For each of these pairs, we determine the minimum-cut in the compressed graph. The minimum of all these cuts is reported as the final solution. The overall algorithm is illustrated in Figure 3. A key observation is that each of these graphs is assumed to be small enough to be read into main-memory. Since the minimum-cut *s-t* problem is known to be efficiently solvable for memory-resident graphs [2], it follows that each of these problems can be solved efficiently. Even though the minimum-cut may need to be determined multiple times over the different compressed graphs, it is still much more efficient to do so, because the disk-to-memory scale up for each individual problem needs to be taken into account in addition to the reduction of the individual problem size. Most maximum-flow based techniques require augmenting path or other preflow-push techniques in order to determine the minimum-cut [2]. Such techniques do not access the edges in any particular order which can be efficiently pre-stored contiguously. Therefore, such methods are not very practical on the original disk-resident graph, since they may require random accesses to disk. On the other hand, since our approach is able to read the *entire compressed graph* in main memory at one time, this results in much more efficient algorithm, even though we need to perform the operation on multiple graphs. The overall approach for query-processing is illustrated in Figure 3.

```
Algorithm EdgeCompress(Target: $m_t$, Samples: $k_e$)
begin
   $s = n - m_t$;
   Create $k_e$ edge reservoirs with size $s$ each in
               one data pass;
   while at least one of the $k_e$ reservoirs
     induces a graph with more than $m_t$ components do
   begin
      Double the edges in the reservoirs which induce
       graphs with more than $m_t$ components in a
       database pass;
   end
   Remove edges one-by-one from all reservoirs for which
      the induced graphs have less than $m_t$
      components until the induced graph for each
      reservoir has exactly $m_t$ components;
   Compress each connected component for the induced
     graphs into a single node;
   Remove all self-loops from compressed graphs and
     consolidate parallel edges;
end
```

Figure 4: Creating the Compressed Graphs

2.3 Creating the Compressed Representation

In this section, we will discuss the process of creating the compressed graphs from the data. We will construct the edge-sampled graphs with the process of reservoir sampling. The key difficulty is that the edge sampled graph $H(f)$ is defined in terms of sampling fraction f, but we do not know what this sampling fraction should be in order for the compressed graph to have node cardinality which is approximately equal to the target m_t. We note that if we sample *too many* edges, this will typically result in some inefficiency in terms of storage, but we will see that the resulting reservoir can always be used to create a compressed graph of the appropriate size. Therefore, we will show how to use a logarithmic number of passes in order to achieve a reservoir of the appropriate target size. We note that the *minimum number of edges required* in order to create m_t components is $n - m_t$, and such a graph would be a perfect forest with no cycles. Therefore, we first create a reservoir of $n - m_t$ edges in the first pass, and check if the resulting graph has no more than m_t components. Typically, the number of components would be much greater than m_t because of the presence of cycles in the sampled edges. Therefore, we will perform another pass and double the number of edges present in the reservoir, by adding an equal number of edges to those *already present in the reservoir.* We again check if the number of components in the graph is no more than m_t. We repeat the process of doubling the number of edges until the number of components in the resulting graph is no more than m_t. We then randomly delete edges one by one from the reservoir, until the number of components is *exactly* m_t. The final set of edges in the reservoir defines the m_t connected components. We note that such an approach requires only a logarithmic number of passes over the data set in order to work effectively. We summarize as follows:

LEMMA 1. *For a graph with n nodes, the reservoir sampling approach requires $O(log(n))$ passes over the data in order to determine the compressed representation of the graph for a target node size of m_t.*

The results of the lemma are easy to prove, since the size of the reservoir doubles in each pass, and an upper bound on the reservoir size is the maximum number of edges, which is at most n^2. Note however, that we need k_e different reservoirs in order to construct k_e different compressed graphs. Building the reservoirs sequentially would blow up the number of passes by a factor of k_e. Therefore, we can build all the k_e reservoirs in parallel. In a given pass, we add to each of the reservoirs for which the number of components in the corresponding induced graph still exceeds m_t. Thus, the number of passes continues to be logarithmic, even when multiple compressions have to be handled at one time.

We note that the only purpose of the reservoir is to define the connected components in the graph. Once the connected components have been determined, we go back to the original graph, and perform the following steps:

- We compress each connected component in the original graph into a single node. We note that this will result in parallel loops and self-edges.

- We remove all self-loops. We also consolidate all parallel edges into a single edge with the appropriate weight.

The resulting graph is the edge-compressed graph. This graph is of significantly smaller size than the original graph. As in the previous case, the process of constructing the corresponding compressed graphs can be parallelized. The overall procedure for creating the edge-compressed graphs is illustrated in Figure 4.

Next, we construct the compression with the use of node sampling. In this case, each node is assigned to one of m_t partitions by using random assignment, where the probability of each partition being included is equal. Once the node partitions have been constructed, we use the same pair of steps as in the previous case in order to create the compressed graph.

A question arises as to why we need both node-based and edge-based compression. As we will see, the process of edge-based compression is biased towards preserving cuts with low value. Dense subgraphs are typically compressed into a single node in edge-sampled compressions. This leads to the possibility that the source and sink may map into the same node for edge-sampled compressions. In this case, it is not possible to determine the minimum-cut over the compressed graph in order to estimate the minimum connectivity over the original graph. In order to guard against this possibility, we also add a few node-sampled compressions. As we will see in the analysis section, the addition of a small number of node sampled compressions allows us to ensure with very high probability that the source and sink nodes map onto different nodes in at least one or more compressed graphs.

3. ANALYSIS OF ALGORITHMIC EFFECTIVENESS

In this section, we will provide an analysis of the algorithmic effectiveness of the approach. We first make the following simple observation about the correctness of the approach.

OBSERVATION 1. *The query processing technique for the GConnect algorithm always results in an approximation, which is an upper bound to the true minimum-cut value.*

The logic for this observation is as follows. For each cut in the compressed graph, a cut with equivalent value exists in the original graph. This ensures that the minimum-cut in the compressed graph is also represented in the original graph. On the other hand, the converse is not true. A cut in the original graph may not have a corresponding cut in the compressed graph. Therefore, the minimum-cut in the compressed graph is always an upper bound on the minimum-cut in the original graph. By using a larger number of compressed graph samples, this bound can be tightened. At a slightly later stage, we will investigate the nature of this bound with the use of different input parameters.

First, we will compute the probability that some cut can always be found with this sampling approach. We note that since the source and sink nodes may map onto the same node, a given compression may not yield a minimum-cut value in the compressed graph. If this is true across all compressions, then it will not be possible to provide an estimate of the minimum-cut value. Therefore, we define a (s, t)-valid compression for a source-sink pair s-t as follows:

DEFINITION 3. *A (s, t)-valid cut for a given compression and a source-sink pair s-t is defined as a compression in which s and t map onto different nodes in the compressed graph.*

We note that edge-sampled compressions typically tend to contract dense subgraphs into a single node. Note that when all edge-sampled compressions map the source s and sink t to the same node, it implies that the source and sink are densely connected. Such cuts are typically less important from the perspective of connectivity queries. This is because typical networking applications attempt to find communication bottlenecks with small minimum-cut value. Therefore, it is more important to determine those cases in which the connectivity between the source-sink pairs is as little as possible. However, even for such cases, we would like to be able to provide a reasonable estimate of the minimum-cut value. This is precisely the reason that node-sampled cuts are created. Since nodes are assigned randomly to the m_t different partitions, the probability that the source-sink pairs are assigned to different partitions is given by $1/m_t$. When the sampling is repeated over k_n different independent compressions, the probability is given by $(1/m_t)^{k_n}$. We note that when for modest values of k_n and m_t, such as $m_t = 100$, and $k_n = 10$, the probability of being able to obtain at least one (s,t)-valid compression is $1 - 10^{-20} \approx 1$. This probability is quite acceptable for most practical applications. In practice, a fraction $(m_t - 1)/m_t$ of the node-based compressions may turn out to be (s,t)-valid, and this leads to an even more robust estimate. We summarize the result below:

LEMMA 2. *Let k_n node-sampled graphs be created with node size of m_t. Then, for any source-sink pair (s,t), the probability of obtaining at least one (s,t)-valid compression is given by $1 - (1/m_t)^{k_n}$.*

We note that the entire purpose of the node-sampling approach is to allow a high probability of obtaining (s,t)-valid compressions, even when the source and sink are chosen from the same densely connected component. Even a modestly small value of k_n is sufficient to achieve this goal.

Next, we will study the effect of sampling on the *quality* of the underlying cuts. We would like to provide the *end-user* with some estimates on the quality of the underlying cuts for a given query. To this effect, we will use the edge-sampled cuts, since they are biased towards retaining cuts of low value. For each of the k_e edge sampled graph compressions, let $f(1, m_t) \dots f(k_e, m_t)$ be the fraction of edges (from the original graph) which are needed to be sampled in order to reduce the compressed graph to m_t nodes. Note that while $f(1, m_t) \dots f(k_e, m_t)$ cannot be *controlled* by the end-user (since they are dependent on a randomized sampling process), they can certainly be *known a-posteriori*, once the reservoir-based sampling process has been completed. This estimation is helpful in providing the user with the necessary data required in order to provide feedback about the quality of the cut. This data is stored after the creation of the compressed graphs. We make the following assertion:

LEMMA 3. *Let $V(s,t)$ be the minimum connectivity between the nodes s and t in the original graph G. Let us consider an (s,t)-valid compression for the jth graph, where $1 \leq j \leq k_e$. Then, the probability that the minimum-cut is unaffected by the compression is given by at least $(1 - f(j, m_t))^{V(s,t)}$.*

PROOF. We note that a minimum-cut will survive in the compressed graph, if none of the edges in it are sampled. The probability that a particular edge is sampled is given by $1 - f(j, m_t)$. Since the minimum-cut contains $V(s,t)$ edges, it follows that the probability that none of the edges in the cut are sampled in the reservoir is given by $(1 - f(j, m_t))^{V(s,t)}$. □

An important observation is that since $V(s,t)$ represents the number of edges in the minimum-cut, it typically contains far fewer edges than an average cut. This increases the relative probability of the survival of the minimum-cut. This is also the reason why such an approach for minimum connectivity indexing is likely to work in practice. Furthermore, we will see that typical values of the sampled fraction $f(j, m_t)$ are quite small. In such cases, the value of $(1 - f(j, m_t))^{V(s,t)}$ can be modestly large. For example, in important connectivity applications, cuts containing a small number of edges are especially important from an application point of view. For example, consider a vulnerable cut containing at most 5 edges, and $f(j, m_t) = 0.1$. In such cases, the probability that the cut survives is given by $0.9^5 \approx 0.59$. The use of multiple samples can reduce the probability of survival to arbitrarily large values. This is the reason for using k_e different edge-sampled compressions. For example, if we use $k_e = 10$, and each of these samples are (s,t)-valid with a similar value of $f(j, m_t)$, then the probability that the minimum-cut survives in none of the compressions is given by $(1 - 0.59)^{10} \approx 1.34 * 10^{-4}$. This probability is sufficiently small to assure us of the minimum-cut with high probability. While the value of $f(j, m_t)$ may be different across different values of j, the aim of this example was to show that the approach retains its effectiveness over small values of k_e. Next, we will quantify the effect of using multiple compressions on the probability of the survival of the minimum-cut value. In order to achieve this goal, we will introduce some additional notations.

For a given source s and sink t, and index $j \in \{1 \dots k_e\}$, we define the binary bit $B(j,s,t)$ to be 1, if the jth edge-sampled graph is (s,t)-valid. Otherwise, we define $B(j,s,t)$ to be 0. We note that the value of $B(j,s,t)$ can be determined from the inverted representation for a source s and sink t. Then, the estimated value of the probability that the minimum-cut survives is as follows.

LEMMA 4. *Let $V(s,t)$ be the minimum connectivity between the nodes s and t in the original graph G. Then, the probability that the minimum-cut is obtained from the k_e different edge-sampled compressions is given by at least $1 - \Pi_{j \in \{1 \dots k_e\}, \{B(j,s,t)=1\}} (1 - (1 - f(j, m_t))^{V(s,t)})$.*

PROOF. These results can be proved directly from Lemma 3. We note that the minimum of the cut values is the true minimum-cut, if a minimum-cut survives in at least one of the k_e edge-sampled compressions in which s and t map onto different nodes. The complement of this event is one in which the cut survives in *none* of the edge-sampled compressions. We can obtain the value of this complement by multiplying together the probabilities for each of the indices j for which the value of $B(j,s,t)$ is 1. The individual probabilities in this product may be determined by using the results of Lemma 3. The result follows. □

We note that $V(s,t)$ is not known explicitly. However, an *over-estimate* on $V(s,t)$ can be obtained by using the least of the minimum *s*-*t* cut values across the different compressions. As long as we use an *over-estimate* on the value of $V(s,t)$, the results of Lemma 4 continue to hold true. We use this over-estimate in order to provide bounds on the quality of the results obtained by the index.

3.1 Further Optimizations

As discussed earlier, we use node-sampled compressions in order to reduce the probability that the source and sink pair map to the same partition in a given compression. This situation often arises in the case of edge-sampled compressions, but does not arise too often in node sampled compressions. The reason that this situation often arises in edge sampled compressions is that some of the dense subgraphs are very large, and may absorb too many of the nodes. In order to improve the results further, we put a constraint on the size of each edge-sampled component during the contraction process. In other words, during the process of contraction, we ignore edges which are incident on the components whose size has reached this maximum. We note that this option essentially incorporates some of the concepts of node sampled compression into the edge sampled compressions as well. Furthermore, the node-sampled compressions are used for estimation in query processing only for cases where a valid pair is not available from the edge-sampled compressions.

4. EXPERIMENTAL RESULTS

In this section, we will present the experimental results of the *GConnect* index. We will present the effectiveness and efficiency on a number of real data sets. We note that the minimum-cut method needs to be implemented as a subroutine for query-processing in our disk based index. For this purpose, we used the HIPR implementation available in [20]. This is essentially an efficient version of the push-relabel algorithm [3] for the *s-t* maximum flow problem. We also create a *disk-based implementation* of this algorithm, in which each access to a node or an edge was required to go back to the disk in order to determine the appropriate parameters. This also means that all operations such as retrieving an edge, updating the capacity information, or performing a relabel need to access the disk in an order which cannot be controlled a-priori. Because of this random access, the disk-based version is orders of magnitude slower than the memory-based version. Nevertheless, it is the only reasonable solution for very large graphs. In this paper, we will show that the repeated use of the memory-resident technique (which is required by our index structure) is much more efficient than even a single application of the disk-based algorithm.

The algorithm is tested on six real data sets. The first five are matrix data sets, which are downloadable from the University of Florida Sparse Matrix Collection web site [1]. The particular data sets used from that web site were graham1, ex3sta1, Andrews, gupta1, and cage13. The sixth data set is the well known DBLP data set[2].

The overall process comprises the main steps of index construction and query processing. We will test the effectiveness of query processing, and the efficiency of both. In the index construction step, we generate k_e edge-based contractions and k_n node-based contractions. The default value of k_e was set to 100, and the value of k_n is set to $0.2 \cdot k_e$ in our experimental setting. We will explicitly specify the places at which these parameters are set to different values. The query processing step is tested with 500 queries in order to illustrate the effectiveness and efficiency of our algorithm. We will provide detailed results in this section.

4.1 Accuracy Analysis

The objective of the *GConnect* algorithm is to efficiently determine the connectivity of a large graph by shrinking its size without significantly compromising accuracy. The effectiveness of the *GConnect* algorithm in terms of the compression factor and the accuracy is illustrated in Table 1. The table provides the size of the original data set, the compressed graph size, the compression ratio, and the accuracy in terms of the percentage of time that the minimum-cut is correctly determined. We note that the compressed graph size is the average behavior of a single graph, and the total size of all compressed graphs may possibly be larger than the original data. The raw sizes of the data sets are dominated by the number of edges rather than the number of nodes.

[1]http://www.cise.ufl.edu/research/sparse/matrices
[2]http://www.informatik.uni-trier.de/˜ley/db/

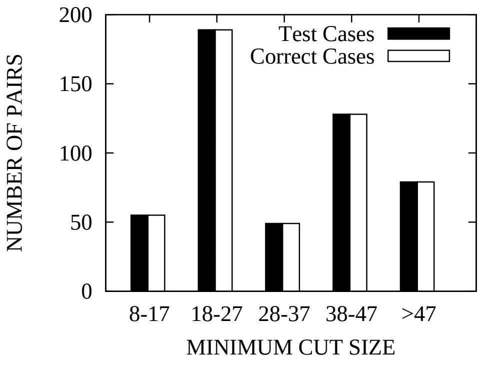

Figure 5: Accuracy Plot (Data Set *ex3sta1*)

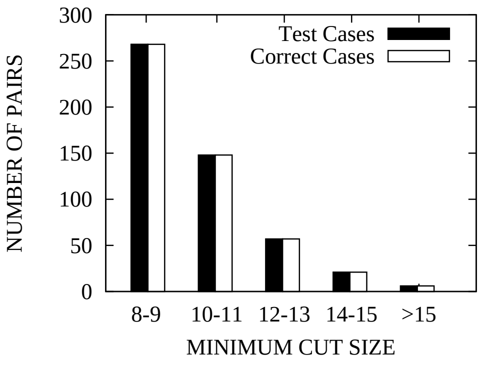

Figure 6: Accuracy Plot (Data Set *Andrews*)

In each case, the results are averaged over 500 source-sink pairs. In order to calculate the compression ratio, we computed the percentage *reduction* in the size of the graph size as a result of the compression process. It is evident from Table 1 that the reduction in the size of the compressed graphs is greater than 95% in almost all the cases. Thus, the compressed graph is at least an order of magnitude smaller than the original graph. We will see in a later section that this reduced size results in orders of magnitude improvement in the processing times of the *GConnect* algorithm. This is especially the case because the running times scale quadratically with the underlying graph size.

In the last column of Table 1, we have also illustrated the accuracy of the minimum-cut with respect to the true value. This accuracy is expressed as the percentage of source-sink pairs for which a correct cut value is returned. In three of the data sets, we obtained the correct minimum-cut over all 500 cases. In the other data sets, there were errors in only a small percentage of the cases. While Table 1 provides a good overview of the accuracy behavior of the algorithm, it provides little understanding of how this accuracy is affected by the minimum-cut value and the behavior of the erroneous cases. The value of the minimum-cut plays a role in the accuracy level, since the probability of incorrectly contracting the true minimum-cut increases with the value of the minimum-cut.

In order to provide better insight, we first examine the distribution of the accuracy with different minimum-cut sizes. We binned the output values of the minimum-cut into dif-

	Original Size Nodes/Edges	Compressed Graph Size Nodes/Edges	Reduction (%)	Error (%)
graham1	9K / 0.21M	0.6K / 0.9K	99.58	5.8
ex3sta1	16K / 0.33M	3K / 12.7K	96.16	0.0
Andrews	60K / 0.37M	10K / 24.1K	93.15	0.0
dblp	164K / 0.76M	16K /19.8K	97.35	2.8
gupta1	31K / 1.06M	12K / 17.2K	98.39	0.0
cage13	445K / 3.52M	40K / 90.1K	97.44	1.4

Table 1: Size Reduction and Overall Effectiveness

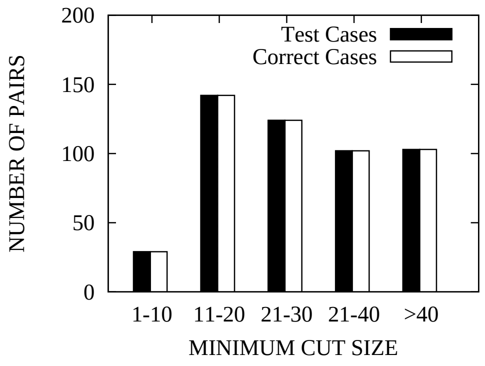

Figure 7: Accuracy Plot (Data Set *gupta1*)

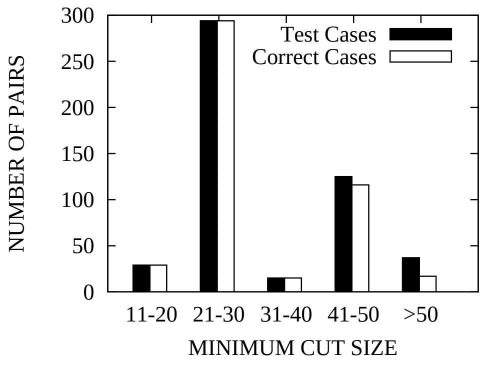

Figure 8: Accuracy Plot (Data Set *graham1*)

ferent ranges and show the accuracy over each range. The results for the six data sets are illustrated in Figures 5, 6, 7, 8, 9 and 10 respectively. The dark bar in each figure represents the number of cases tested, and the white bar represents the number of cases for which the correct minimum-cut value was returned. In the case of Figures 5, 6, and 7, the two bars are exactly the same, because all 500 source-sink pairs are solved correctly for these data sets. On the other hand, for the cases of Figures 8, 9 and 10, we can see that there are a few errors in some cases. Most of the errors were for cases where the minimum-cut value was large. This is because a larger value of the minimum-cut is more likely to cause an incorrect contraction. Even in the cases, where the minimum-cut was not correct, the relative errors which were obtained were relatively small. Another observation is that every single minimum cut returned by the index (throughout the experimental section) was from an edge-sampled compression rather than a node-sampled compression. Nevertheless, the use of node-sampled compressions is necessary in order to ensure that a theoretically valid cut is always available. Next, we will examine the error distribution of the cases in which an incorrect result was obtained by the algorithm.

We plot the distribution of the *relative error* behavior with the output value of the minimum-cut. The relative error is defined as the ratio of the absolute cut error to the correct minimum-cut value. The results are illustrated in Figures 11, 12 and 13. In this case, we present the results only for the three data sets in which some of the source-sink pairs yield incorrect results. Furthermore, since the number of incorrect cases is much smaller than the number of correct cases, we present only the results for the incorrect cases in order to enable clarity of presentation. The X-axis in each

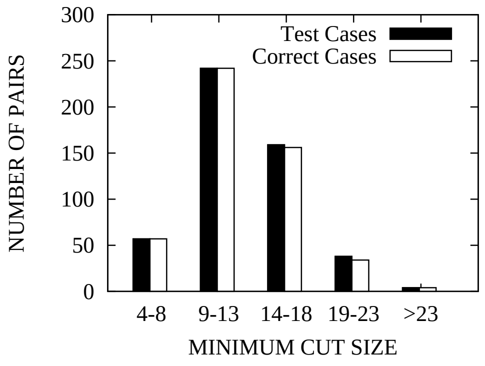

Figure 9: Accuracy Plot (Data Set *dblp*)

figure contains the range of relative error, while the Y-axis indicates the number of cases for which this range of relative error was achieved. In all cases, most of the erroneous cases had relative error which was less than 0.6. For two of the data sets (dblp and cage13), the majority of the cases had relative error which was less than 0.4. Thus, for the small number of source-sink pairs in which the correct cut was not obtained, the relative error was quite small and provided a good idea of the minimum-cut value.

4.2 Computational Efficiency

All experiments were performed on a Microsoft Windows XP machine with Intel Core2 Duo 2.5G CPU and 1.5GB main memory. The algorithm was implemented in C++. The existing minimum s-t cut algorithms require random access to disk for each operation such as performing a flow

Data Set	#Nodes(K)/#Edges(M)	Time (secs.)
graham1	9K/0.21M	22.96
ex3sta1	16K/0.33M	65.18
Andrews	60K/0.37M	173.13
dblp	164K/0.76M	691.52
gupta1	31K/1.06M	215.84
cage13	445K/3.52M	2189.67

Table 2: Index Construction Time

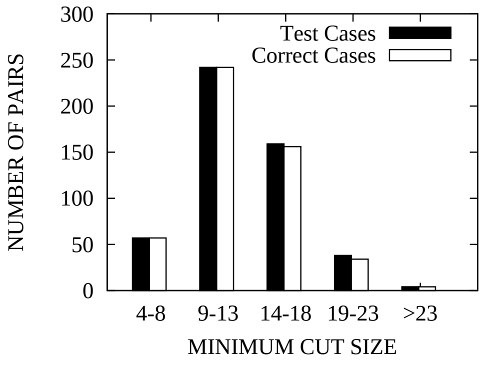

Figure 10: Accuracy Plot (Data Set *cage13*)

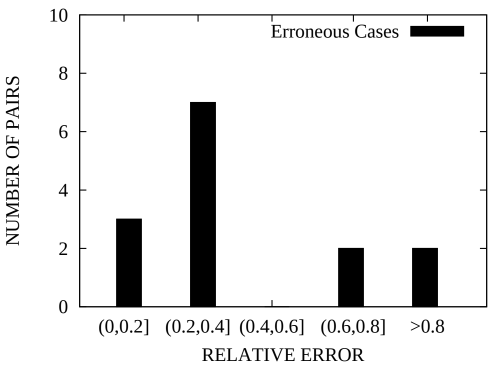

Figure 12: Relative Error Distribution (Data Set *dblp*)

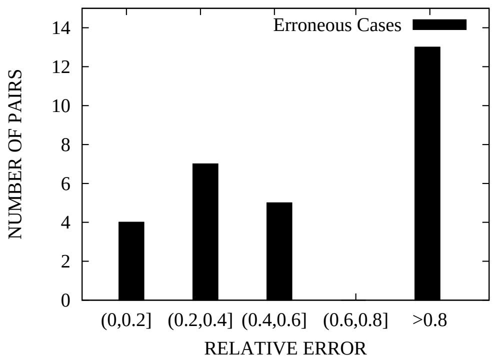

Figure 11: Relative Error Distribution (Data Set *graham1*)

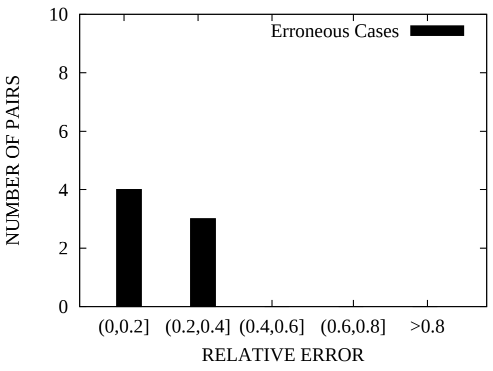

Figure 13: Relative Error Distribution (Data Set *cage13*)

augmentation on an edge or a relabel on a node. This makes the execution time orders of magnitude slower than memory resident algorithms. The aim of the repeated compression approach in our technique is to reduce the underlying graph size so that it can be efficiently solved with memory resident algorithms. We further note that the running times of maximum flow algorithms scale *superlinearly* with the graph size. Therefore, the computational time in processing a reduced graph is reduced by a much greater factor than the reduction in size. It is evident from Table 1 that the reduction in sizes of the different data sets are typically by a huge factor of nearly 20. Therefore, the CPU time reductions are by even greater factors. When the combine the savings from this reduction and the memory-based implementation, we will see that the query processing time of the *GConnect* algorithm is superior to the disk-based implementation *in spite of* the fact that the *GConnect* algorithm needs to run the method repeatedly in order to effectively approximate the minimum-cut.

We will examine the times required for index construction and query processing. In order to generate these results, we used the default value of 100 edge-sampled and 20 node-sampled compressions. The time required for index construction is illustrated in Table 2. The index construction time in Table 2, is dominated by two components. The first component is the creation of the connectivity index with the use of a sample-based compression approach. This step essentially determines the mapping of nodes to the different partitions. This mapping is then efficiently stored in the inverted representation. The second component processes this compressed graph, removes all self-loops, and consoli-

Data Set	*GConnect* Query Processing Time (seconds)	Disk-based Running Time (seconds) No Cache	Disk-based Running Time (seconds) 5% Cache
graham1	1.80	1,550	1,345
ex3sta1	7.59	2,300	2,164
Andrews	22.80	3,070	3,004
dblp	8.20	4,712	4,457
gupta1	12.66	5,245	5,076
cage13	78.37	59,339	58,710

Table 3: Running Time Comparison

dates all parallel edges into a single edge with an appropriate weight. It is evident from the results of Table 2, that the running times are quite modest in all cases. Since the index construction step is a pre-processing step, we have greater leeway in allowing for larger running times than the query processing step. It is evident from Table 2, that the construction phase required only a few minutes in most of the data sets.

Next, we will examine the query processing efficiency of the *GConnect* algorithm. We will compare the query processing times with a disk-based version of the algorithm. We also implemented a disk-based version with a cache which was 5% of the size of the original graph. The cache was used to speed up processing. The results are illustrated in Table 3. Each row in the table illustrates the running time for executing the minimum-cut algorithm on 50 different source-sink pairs. It is evident that the the *GConnect* algorithm is several orders of magnitude faster than (both versions of) the disk-based implementation, even though it needs to be run multiple times on several compressed graphs. This is quite reasonable because each compressed graph is almost two orders of magnitude smaller than uncompressed graph, and we do not need to perform any disk-based computations. The overall result is that the query-processing time is *between two and three orders of magnitude faster* than the disk-based version of the algorithm. It is also evident from the results of Table 3, that our approach provides the difference between an impractical disk-resident solution and an extremely efficient query processing index for the problem. We further note that the caching technique was not particularly helpful because of the random access behavior of the underlying edges. As a result, the advantage of the cache did not significantly outweigh the extra overhead of maintaining the cache.

4.3 Sensitivity Analysis

The experimental results presented so far show that the *GConnect* algorithm is extremely effective and efficient over the different data sets. In this section, we will also illustrate the robustness of the technique. Clearly, the effectiveness of the algorithm is sensitive to the number of compressions. Since edge-sampled compressions are more crucial to the effectiveness of our approach, we will test its effectiveness with increasing number of edge-sampled compressions. We will compare the different techniques over various ranges of minimum-cut values.

The results for the different data sets are illustrated in Figures 14, 15, 16, 17, 18 and 19. The X-axis corresponds to the minimum-cut size and the Y-axis corresponds to the accuracy. We have illustrated the accuracy for different number

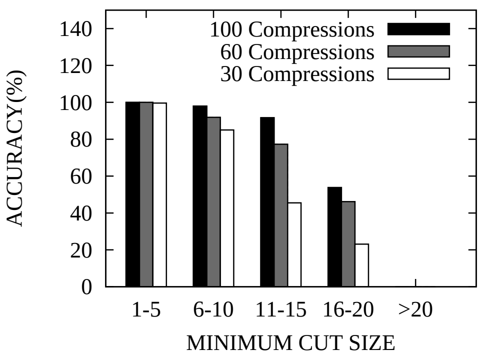

Figure 14: Sensitivity Plot (Data Set *dblp*)

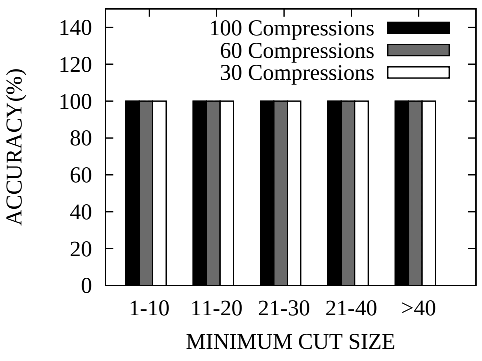

Figure 15: Sensitivity Plot (Data Set *gupta1*)

of compressions by bars of different colors. Clearly, the accuracy of the *GConnect* algorithm increases with increasing number of compressions. However, it is interesting to see that the *GConnect* algorithm was able to provide obtain modestly accurate results with as few as 30 compressions in some data sets. A particular example is the case of the gupta1 data set, which is illustrated in Figure 15. In this case, the accuracy is always 100% even when as few as 30 compressions are used. In general, the results are extremely robust across different numbers of compressions for the data sets illustrated in Figures 15, 17, and 18. In these cases, extremely robust results may be obtained by using as few as 100 edge-sampled compressions. These results indicate that the *GConnect* technique is an extremely robust method which is several orders of magnitude more efficient than currently available techniques, and can be used effectively in a

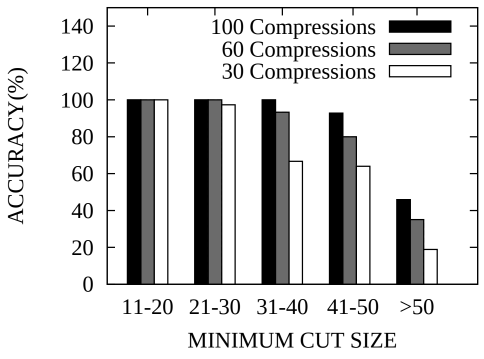

Figure 16: Sensitivity Plot (Data Set *graham1*)

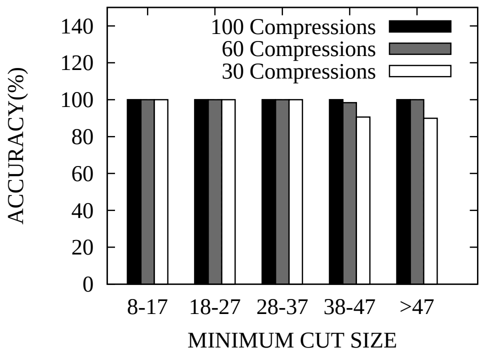

Figure 17: Sensitivity Plot (Data Set *ex3sta1*)

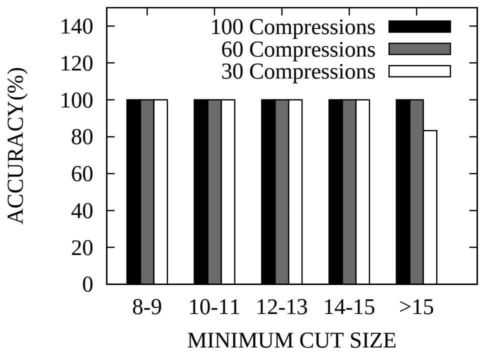

Figure 18: Sensitivity Plot (Data Set *Andrews*)

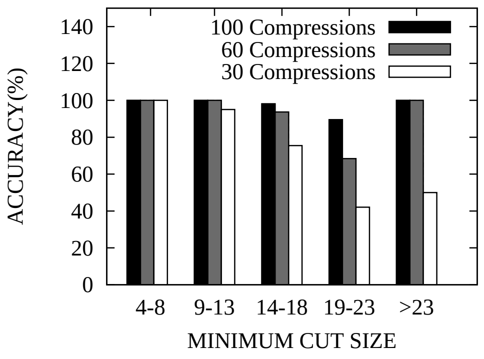

Figure 19: Sensitivity Plot (Data Set *cage13*)

wide variety of practical scenarios.

5. CONCLUSIONS AND SUMMARY

In this paper, we presented a connectivity index for massive disk-resident graphs. The problem of connectivity queries is extremely challenging in the disk-resident case, because the query-processing techniques may need to access the edges on disk in random order. As a result, queries which are normally quite efficient for the memory-resident case are extremely slow over such graphs. The goal of the approach is to create a compressed representation of the underlying graphs, and use it as an index for connectivity query processing. Even though such an approach requires multiple applications of a minimum-cut algorithm over the compressed graph, the ability to do so over a memory-resident graph is an enormous advantage for the technique. The results show that the approach is an extremely effective technique which improves the performance by orders of magnitude for the disk-resident case. At the same time, the approach provides extremely high accuracy in terms of quality.

This paper discusses the particularly challenging case of disk-resident graphs. An even more challenging case is that of graph streams which evolve over time. Such graphs can arise in the context of a variety of network-based applications. In future work, we will discuss the extension of this approach to the case of graph streams. We expect that such applications will require a more innovative approach to graph compression, since it is no longer possible to perform multiple passes over the data set.

Acknowledgements

Research of the first author was sponsored in part by the US Army Research laboratory and the UK ministry of Defense under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies of the US Government, the US Army Research Laboratory, the UK Ministry of Defense, or the UK Government.

6. REFERENCES

[1] C. Aggarwal, N. Ta, J. Feng, J. Wang, M. J. Zaki. XProj: A Framework for Projected Structural Clustering of XML Documents, *KDD Conference*, pages 46-55, 2007.

[2] R. Ahuja, J. Orlin, T. Magnanti, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1992.

[3] B. Cherassky, A. Goldberg, On Implementing the Push-Relabel Method for the Maximum Flow Problem. *Algorithmica*, 19(4): 390-410, 1997.

[4] D. Cook, L. Holder, Mining Graph Data, *John Wiley & Sons Inc*, 2007.

[5] M. Faloutsos, P. Faloutsos, C. Faloutsos, On Power Law Relationships of the Internet Topology. *SIGCOMM Conference*, pages 251-262, 1999.

[6] D. Gibson, R. Kumar, A. Tomkins, Discovering Large Dense Subgraphs in Massive Graphs, *VLDB*

Conference, pages 721-732, 2005.

[7] R. Kumar, P Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, E. Upfal. The Web as a Graph. *ACM PODS Conference*, pages 1-10, 2000.

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

[9] S. Navlakha, R. Rastogi, N. Shrivastava, Graph summarization with bounded error. *SIGMOD Conference*, pages 419-432, 2008.

[10] N. Polyzotis, M. Garofalakis. Xsketch synopses for XML data graphs. *ACM TODS Journal*, 31(3):1014-1063, 2006.

[11] S. Raghavan, H. Garcia-Molina. Representing web graphs. *ICDE Conference*, pages 405-416, 2003.

[12] T. Suel, J Yuan. Compressing the graph structure of the web, *Data Compression Conference*, pages 213-222, 2001.

[13] A. A. Tsay, W. S. Lovejoy, David R. Karger, Random Sampling in Cut, Flow, and Network Design Problems, *Mathematics of Operations Research*, 24(2):383-413, 1999.

[14] H. Wang, H. He, J. Yang, J. Xu-Yu, P. Yu. Dual Labeling: Answering Graph Reachability Queries in Constant Time. *ICDE Conference*, pages 75, 2006.

[15] X. Yan, J. Han. CloseGraph: Mining Closed Frequent Graph Patterns, *ACM KDD Conference*, pages 286-295, 2003.

[16] X. Yan, H. Cheng, J. Han, and P. S. Yu, Mining Significant Graph Patterns by Scalable Leap Search, *SIGMOD Conference*, pages 433-444, 2008.

[17] X. Yan, P. S. Yu, and J. Han, Graph Indexing Based on Discriminative Frequent Structure Analysis, *ACM Transactions on Database Systems (TODS)*, pages 960-993, December 2005.

[18] X. Yan, P. S. Yu, and J Han, Substructure Similarity Search in Graph Databases, *SIGMOD Conference*, pages 766-777, 2005.

[19] M. J. Zaki, C. C. Aggarwal. XRules: An Effective Structural Classifier for XML Data, *KDD Conference*, pages 316-325, 2003.

[20] http://avglab.com/andrew/soft.html