Scalable Discovery of Best Clusters on Large Graphs

Kathy Macropol
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
kpm@cs.ucsb.edu

Ambuj Singh
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
ambuj@.cs.ucsb.edu

ABSTRACT

The identification of clusters, well-connected components in a graph, is useful in many applications from biological function prediction to social community detection. However, finding these clusters can be difficult as graph sizes increase. Most current graph clustering algorithms scale poorly in terms of time or memory. An important insight is that many clustering applications need only the subset of best clusters, and not all clusters in the entire graph. In this paper we propose a new technique, Top Graph Clusters (TopGC), which probabilistically searches large, edge weighted, directed graphs for their best clusters in linear time. The algorithm is inherently parallelizable, and is able to find variable size, overlapping clusters. To increase scalability, a parameter is introduced that controls memory use. When compared with three other state-of-the art clustering techniques, TopGC achieves running time speedups of up to 70% on large scale real world datasets. In addition, the clusters returned by TopGC are consistently found to be better both in calculated score and when compared on real world benchmarks.

1. INTRODUCTION

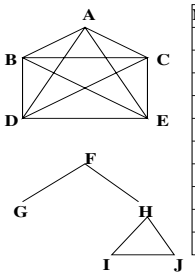
Large amounts of graph data are generated each day from applications such as social networks, communication graphs, biological networks, and more. These structures grow from the hundreds to the millions of nodes and are both useful and important to analyze [22, 23]. As graphs grow in size, it becomes difficult to use or inspect them without some form of summarization. Clustering the nodes of these networks is one technique shown to be of great practical importance [5, 23]. Not only are the small, dense subgraphs easier to visualize and analyze, but well-connected groupings of nodes within graphs have been found to correspond to many real world problems, like biological function prediction and social or web community detection [5, 22, 23, 30]. However, finding the clusters in a graph is difficult as graph sizes grow large. Unfortunately, most current graph clustering algorithms scale poorly in terms of time or memory with increasing graph size. Several recent graph clustering (and relatedly, graph partitioning) techniques have been introduced which improve scalability [10, 25]—however, tradeoffs in clustering quality are made, and these techniques are better suited for partitioning graphs into several large pieces rather than finding small densely connected subgraphs.

An important observation is that all these applications cluster the entire graph, regardless of whether every cluster from the entire graph is needed or useful. As an example, clusters connected weakly, if at all, would be useless for biological function prediction. Applications wishing to find significant social groupings would not need clusters returned of people loosely connected. In both these cases, only clusters that are most strongly connected are needed. An algorithm that finds all the clusters of an entire graph, only to rank and keep just a few, is extremely inefficient. Instead, it may be faster and still useful to obtain just the top clusters from a large network.

In this paper we propose a new algorithm, Top Graph Clusters (TopGC), which probabilistically finds the best well connected, clique-like clusters within large graphs. It is inherently parallelizable, and runs in linear time on the graph size. In addition, the memory consumption can be constrained through input parameters. The algorithm works on both directed and undirected edge weighted graphs, and can also find variable size clusters ranging from an input minimum cluster size to input maximum cluster size. The discovered clusters are allowed to overlap up to a given input percentage. The ability to find slightly overlapping clusters is important in many applications where a single node may be part of multiple clusters, for example in gene networks or social web graphs. This importance has been noted and analyzed [23, 28], however, very few current graph clustering algorithms allow for this feature.

The TopGC technique is inspired from Locality Sensitive Hashing (LSH) [12]. LSH is a probabilistic method for similarity search, and TopGC is therefore also probabilistic in nature. Well separated clusters with strong, clique-like connections between the nodes are more likely to be found and returned by the algorithm. The basic idea behind TopGC's algorithm comes in two parts. First, many previous clustering techniques and papers have noted that nodes with similar neighbor sets (neighborhoods) within a graph generally should cluster together. A perfect clique, for example, would contain nodes whose neighbor sets would match exactly between them (all nodes are connected to the same other nodes, if the neighborhood of a node includes itself). Nodes that do not cluster together would tend to have neighborhood sets that do not overlap. Fig 1 contains an example which illustrates this graphically. In this figure, five of the nodes form a perfect clique, and therefore have identical neighborhoods. The next five nodes do not cluster as well together, and this is reflected in their neighborhoods. In addition to the neighborhood, the edge weights among nodes also make a difference in the strength of a cluster. TopGC takes advantage of LSH's fast similarity search to quickly find only those

Proceedings of the VLDB Endowment, Vol. 3, No. 1



Node	Neighborhood
A	{A,B,C,D,E}
B	{A,B,C,D,E}
C	{A,B,C,D,E}
D	{A,B,C,D,E}
E	{A,B,C,D,E}
F	{F,G,H}
G	{F,G}
H	{F,H,I,J}
I	{H,I,J}
J	{H,I,J}

Figure 1: Relationship between neighborhood and clustering. Here, nodes *A,B,C,D,* and *E* form a perfect clique, and contain matching neighborhoods. Nodes *F,G,H,I* and *J* do not cluster as well, and this is reflected in their neighborhoods.

nodes that have similar neighborhoods. Further, TopGC modifies the algorithm so that only those sets of nodes connected by high edge weights are found. It is shown later that the probability of a cluster being found by TopGC is related to the score of the cluster.

Second, to reduce the memory consumption needed to cluster a large graph, TopGC takes further advantage of the fact that only the best clusters in the graph need to be found. The algorithm further modifies the LSH process, splitting it into two phases: a pruning phase and a final clustering phase. The pruning phase picks only those nodes most likely to be part of the best clusters, maximizing a score function shown to be related to the final clustering score. These selected nodes are then clustered in the final clustering phase.

Both phases of the algorithm are parallelizable into an arbitrary number of separate threads. This indicates that parallelization frameworks such as MapReduce [9] can work efficiently with it, which would allow for still further scalability of the TopGC algorithm.

The main contributions of this paper include the proposal of a new algorithm, TopGC, which is able to quickly and effectively find strong, overlapping clusters within large, directed, edge weighted graphs. The algorithm is analyzed and shown to probabilistically return clusters with higher scores. When implemented and compared with three other state-of-the-art clustering techniques, TopGC is found to achieve running time speedups of up to 70% on large scale real world datasets. In addition, the memory savings achieved from TopGC's pruning algorithm allows it to cluster several massive graphs that other techniques can not process due to memory constraints. The clusters returned by TopGC are found to score higher both in calculated score and when compared on real world benchmarks. TopGC performs better than the popular clustering tool MCL on biological datasets.

The rest of this paper is divided as follows. First, in Section 2 several related works in the area of graph clustering are reviewed. Next, the scoring criteria for judging the strength of a cluster are discussed in Section 3. After that, the basic TopGC algorithm is introduced in Section 4, and the probability of it discovering a cluster is shown to relate to the cluster's score. Finally, the results obtained from implementing TopGC and comparing its performance on both generated and real world networks against three current clustering techniques are analyzed and discussed in Section 5.

2. RELATED WORK

Multiple previous graph clusterings algorithms have been proposed, using techniques from areas such as spectral clustering [26] and random walks [22, 25, 29]. Some have been found to return very strong clusters which match well to real world standards and clusterings. MCL in particular has seen much use in the area of biological networks, returning highly relevant clusters in several areas [3, 7]. MCL is a clustering technique based on the simulation of random walk flows. By alternating between applying two operators (expansion and inflation) upon the graph matrix of transition probabilities, a clustering related to the random walk distance between nodes can be obtained. The difficulty with both MCL and the majority of other graph clustering techniques is their inability to scale as graph sizes increase.

Given the complexity of the graph clustering problem on even small graphs, much less on the massive graphs available today, there have been several prior approaches proposed to tackle the scalability bottleneck. One approach is to prune the search space by analyzing and returning only the *local* clusterings around a given seed node. This allows the algorithm to focus only on a limited subset of the graph to achieve time and space savings. Examples of such local graph clustering algorithms include the Local Spectral Algorithm [2] and Nibble [27].

Graph partitioning, which decomposes a graph with a predetermined number of graph cuts, is a separate but related problem to the graph clustering problem we examine here. Graph partitioning can be viewed as clustering of a graph into a predefined number of clusters. Metis [1] is a multi-level graph partitioning program that achieves scalability by coarsening and performing clustering on a reduced size graph, then uncoarsening to obtain the final partitions. This algorithm, however, restricts its partitions to be nearly equal in size, in addition to the number of partitions being specified in advance. Graclus [10] is another variant on multi-level graph partitioning. By maximizing a weighted k-means objective shown to be equivalent to spectral clustering, Graclus reduces time complexity by avoiding eigenvector computations. In addition, Graclus allows for its partitions to vary in size, though the number must still be specified in advance. An obvious problem with such partitioning approaches is that the number of clusters in a graph is frequently not known in advance, especially for large, complex networks. Furthermore, these algorithms often are most efficient when the number of graph cuts is small, which can lead to cluster sizes too large for practical analysis.

A variation on the MCL algorithm, called RML-MCL [25], that improves MCL's scalability has been recently introduced. RML-MCL replaces the expansion operator with a regularization step. This step allows for a multi-level graph clustering approach to be combined with random walk flow simulations. However, the regularization parameter also leads the algorithm to produce larger clusterings, an effect confirmed in our experiments in Section 5.3.

Local clustering and graph partitioning can both be viewed as methods to reduce the complexity of the graph clustering problem, and allow for a faster, more scalable solution than having to do the global clustering. A different approach, and one that has many practical applications, is to return just the smaller, best scoring clusters of the entire graph. One recent paper has focused on finding the top-k maximal cliques in uncertain graphs [32]. However, this method of reducing the search space by finding cliques means many useful real world clusters may be missed, as they may connect strongly but be missing several edges between their nodes. Other methods for pruning by finding only the strongest clusters have not yet been explored fully, and this is the problem we focus upon in this paper.

3. CLUSTER STRENGTH AND SCORING

To return only the strongest or best clusters in a graph, we need the notion of a score quantifying the strength of a cluster. Many different cluster scoring methods have been introduced over the years, each with their own strengths and weaknesses. The specific choice

often depends on the particular application and type of clustering desired.

One popular measure of cluster quality is conductance, defined as the ratio of the number of links within a cluster to the number of links leaving that cluster [15]. Conductance has been used in many applications to good results, but it does have several drawbacks. A cluster containing several internally disconnected components within it may score higher than a similarly sized, better connected cluster of nodes. In addition, conductance as a cluster score does not work well when there are multiple slightly overlapping, but separate, clusters [23]. In this case, the edges leaving the cluster will penalize the score, and often cause the separate clusters to be merged together.

A different measure of score is to maximize how clique-like a cluster is. One example is the intra-cluster distance [11], defined as the ratio between the number of edges in a cluster to the number of possible edges within that cluster. Another related method is the ratio association [10], a score given to a set of clusters. The ratio association calculates the average link weight within each of its clusters, then sums these averages. A limitation of both the intra-cluster distance and the ratio association is that there is no preference given to larger cluster sizes (a clique of size 100, for example, is less probable by chance and intuitively should score higher than a clique of size 3).

In this paper, since the goal is to find strong, possibly overlapping clique-like clusters, a variation on both the intra-cluster distance and ratio association is used. This score has been found to correlate well with real world clusterings [22]. Here, a cluster's score is defined as the average link weight between all nodes in the cluster (including links with weight 0) multiplied by the square root of the cluster size (to allow for a bias towards larger cluster sizes).

$$Score\ of\ Cluster\ C = \sqrt{|C|}\ \frac{\sum_{v_i \in C} \sum_{v_j \in C} w_{i,j}}{|C| \cdot (|C| - 1)} \qquad (1)$$

This score is later shown to be related to the probability of TopGC discovering the cluster.

4. ALGORITHM OVERVIEW

To cluster the nodes in a graph, we wish to find those sets of nodes whose neighborhoods are most similar, and contain high edge weights between them.

Let $G = \{V, E, w\}$ be a weighted, directed graph where V are the set of vertices, $E = \{(v_1, v_2) \mid v_1, v_2 \in V\}$ are the set of directed edges, and $w(v_1, v_2)$ (abbreviated $w_{1,2}$) gives the weight of the edge going from vertex v_1 to vertex v_2. We assume that edge weights have been normalized, such that each link weight is a number between [0,1], and $\sum_{j \in V, j \neq i} w_{i,j} = 1\ \forall\ i \in V$. In this case, $w_{i,j} = 0$ if $(v_i, v_j) \notin E$. For simplicity in later calculations, let n be the number of vertices in the graph. Also, let the neighborhood N_i of vertex v_i be defined as:

Definition 1. $N_i = \{v_i\} \cup \{v_j \mid w_{i,j} > 0\}$

The TopGC algorithm we propose here makes use of Locality Sensitive Hashing (LSH) [12]. LSH has been used as a fast and efficient method for linear time similarity search in numerous practical applications [6].

4.1 LSH Applied To Graphs

If we first ignore edge weights, we can use each node's neighborhood set with a version of LSH that finds similarity based on the Jaccard index [6] to probabilistically find those nodes with similar neighborhoods. The Jaccard index of two nodes, v_i and v_j, is defined as $\frac{|N_i \cap N_j|}{|N_i \cup N_j|}$, and ranges between 0 (no overlap in their neighborhoods) to 1 (the neighborhoods are identical). In this technique, each node is represented by a small, length l, probabilistic "signature" created from its neighborhood. By generating multiple signatures for each node and hashing them, sets of nodes with matching signatures and similar neighborhoods may be found. In this section we give a brief overview of this technique. In the two sections following, we modify and extend the LSH algorithm so that it may be used quickly and effectively on graphs.

To create an LSH signature, there are two main steps. First, m random permutations, $\pi_1, \cdots, \pi_m$, of the nodes in the graph are generated and stored. From this, m "minhash" values, $mh_1, \cdots, mh_m$, are generated for every node. The value of mh_i for a vertex v_j is the element in its neighborhood N_j with the lowest ordering index in π_i. So at the end of this step, a node has m minhash values, each minhash consisting of one node from its neighborhood. It can be seen that the probability that two nodes, v_i and v_j, agree on their values for some mh_k is $\frac{|N_i \cap N_j|}{|N_i \cup N_j|}$, the Jaccard index. In practice, it is inefficient to generate and store the orderings for all nodes on the graph. Instead, a previously introduced family of hash functions, H, is used to approximate the random orderings [14]. Here, Pr[$h(v_i) = h(v_j)$] = Jaccard index, where h is a randomly chosen function of the form $h(x) = ax + b \bmod P$, where P is a prime number larger than n.

Next, a signature of length l is created. A series of l random numbers is generated, each number ranging from [1, m]. For a node, its signature will then consist of the concatenation of mh_{l_1}, $mh_{l_2}, \cdots, mh_{l_l}$ In this way, the probability of two nodes matching on the same minhash becomes $(\frac{|N_i \cap N_j|}{|N_i \cup N_j|})^l$.

There are several limitations with using this algorithm to cluster nodes in a graph. First, if edge weights are added using methods similar to previously proposed weighting solutions [13], this would lead to nodes being grouped based on how *similar* their neighborhood edge weights are, and not on the strength of the weights. In addition, another weakness is the low probability of all nodes from a cluster (or even a significant fraction of them) creating the exact same signature. As more nodes are added to a cluster, the probability of them all generating the same signature, unless they are a perfect clique, will decrease. This difficulty has led to different variants of LSH, such as multi-level LSH [31]. In this paper, we introduce a different solution, modification of the hashwords, which fits well with the graph clustering problem we are trying to solve.

4.2 Inclusion of Edge Weights

To address edge weights, we first define the weighted neighborhood of vertex. The weighted neighborhood, N_i^w, of a vertex v_i is a probabilistic subset of N_i where every vertex $v_j \in N_i$ is included inside N_i^w with a probability proportional to $w_{i,j}$. For example, consider a vertex v_i which has two edges leaving it, one to vertex v_j with weight $w_{i,j} = 0.6$ and the other to vertex v_k with weight $w_{i,k} = 0.4$. Vertex v_i will be included in the set always. Vertex v_j will be included in the set 60% of the time, and v_k will be in the set 40% of the time. This means Pr[$N_i^w = \{v_i, v_j, v_k\}$] = 0.24, Pr[$N_i^w = \{v_i, v_j\}$] = 0.36, Pr[$N_i^w = \{v_i, v_k\}$] = 0.16, and Pr[$N_i^w = \{v_i\}$] = 0.24. To approximate a weighted neighborhood in our algorithm, we draw multiple neighborhood instances from the weighted neighborhood probability distribution. (The number of neighborhood instances drawn is a parameter referred to later as *trials*.) From this, we can approximate using LSH with a weighted neighborhood by hashing as described in Section 4.1 with each of these drawn neighborhood instances.

Given this weighted neighborhood, we wish to relate the sim-

ilarity of two nodes, v_i and v_j, with their probability of hashing together. To do this, we calculate the probability they will pick the same minhash value. Raising this probability to the power of the signature length, l, will give their overall chance of hashing together.

For every vertex, $v_m \in \{N_i \cup N_j\}$, there is a $\frac{1}{|N_i \cup N_j|}$ chance that it will be first among $N_i \cup N_j$ in ordering, for a particular π. In addition, there is a $w_{i,m}w_{j,m}$ chance that it will be included in neighborhoods of both v_i and v_j. The probability that both these will occur and node v_m will therefore be chosen as the minhash for both v_i and v_j is:

$$Pr[minhash_{v_i,v_j} = v_m] = \frac{w_{i,m}w_{j,m}}{|N_i \cup N_j|} \quad (2)$$

This is the probability that any one, particular vertex v_m will be chosen as the minhash value for both v_i and v_j. There is also the possibility that another vertex in $N_i \cup N_j$ was first in ordering, but was not contained in either of the weighted neighborhood instances, leaving v_m to be second in ordering and chosen as the minhash value. And so on for a third, fourth, etc. minhash values. However, these following probabilities all involve multiplying both the likelihood of matching from Eqn. 2 with the likelihood of not matching, raised to the power of the level. These probabilities decrease at an exponential rate. Only the most significant term, listed in Eqn. 2, is kept here.

The overall probability of v_i and v_j choosing the same minhash (represented by $Pr[minhash_{v_i,v_j}]$) will be:

$$Pr[minhash_{v_i,v_j}] = \frac{\sum_{k \in (N_i \cup N_j)} w_{i,k}w_{j,k}}{|N_i \cup N_j|} \quad (3)$$

Note that we are taking the union in the summation, and therefore some weights may be zero. From Eqn. 3, we can already see intuitively that the probability of two nodes hashing together is maximized not only when their neighborhoods are similar (both $w_{i,k}$ and $w_{j,k}$ are nonzero for the same nodes v_k), but also when these weights are maximized. Extending Eqn. 2 to the probability for a cluster of nodes $C = \{v_1, v_2, \cdots, v_c\}$, gives the following probability for all nodes in C to pick a *particular* node, v_k, as a minhash:

$$Pr[minhash_C = v_k] = \frac{\prod_{i \in C} w_{i,k}}{|\bigcup_{i \in C} N_i|} \quad (4)$$

Equation 4 may be rearranged to solve for $\prod_{i \in C} w_{i,k}$, which gives the following equation:

$$\prod_{i \in C} w_{i,k} = Pr[minhash_C = v_k] \cdot |\bigcup_{i \in C} N_i| \quad (5)$$

The left side of this equation is taking the product of a series of numbers. This product can be related to the sum of the same series according to the relation observed in [16]:

OBSERVATION. *If we are given a positive real number k and a (fixed) positive integer n, then among all of the real factor sets of k that consist of n numbers, the real factor set that yields the minimal sum consists of n copies of $k^{1/n}$.*

This means Equation 5 may be converted to the following relation:

$$\sum_{i \in C} w_{i,k} \geq |C|(Pr[minhash_C = v_k] \cdot |\bigcup_{i \in C} N_i|)^{1/|C|} \quad (6)$$

Examining Equation 1, our equation for scoring a cluster, we can rearrange the summations in the numerator to form $\sum_{j \in C} \sum_{i \in C} w_{i,j}$. From this, we can substitute Equation 6 with the $\sum_{i \in C} w_{i,j}$ term in Equation 1. This gives the following bound between the probability of cluster C creating a common minhash, $v_j \in C$, and the score:

$$Score\ of\ C \geq \frac{\sqrt{|C|}}{(|C|-1)} \cdot |\bigcup_{i \in C} N_i|^{1/|C|} \cdot \sum_{v_j \in C} (Pr[minhash_C = v_j])^{1/|C|} \quad (7)$$

Because we are only interested in clusters with high score values, Eqn. 7 leads us to examine only those clusters, C, with a high probability of their signatures containing many vertices of C within them. In other words, those clusters where $Pr[minhash_C = v_j]$ is high for each vertex $v_j \in C$. This can be approximated by examining the hashtable obtained from running LSH on the weighted neighborhoods of each node. Let M be the set of nodes composing the signature. By finding those hashbuckets where $\frac{|C \cap M|}{|C|}$ is high among the frequent nodes, we can obtain a good estimate of $Pr[minhash_C = v_j]$, and therefore obtain clusters with high bounds on their scores.

A fortunate byproduct of the bound found in Eqn. 7 is that we are only concerned with finding clusters where $Pr[minhash_C \in C]$ is high. We do not need the extra ordering information contained within the LSH signatures to estimate this, which allows us to modify each signature to increase the probability of cluster nodes hashing to the same bucket. By sorting the nodes within the signature and removing duplicates, we can still find those buckets where both $\frac{|C \cap M|}{|C|}$, and therefore also $Pr[minash \in C]$, are large. The removal of this extra ordering information allows for an increased probability of cluster nodes hashing together, and is shown experimentally to lead to better clustering results. (For additional details, an analysis of this effect is included in Appendix 8.1).

Note that it is also possible for overlapping clusters to be found. After finding all clusters, a post-processing step is run where, if two clusters overlap more than λ, a given parameter, the cluster with lower score is removed. Overlapping clusterings are of real world importance [23, 28], and may lead to more accurate and useful clusterings. Additional details on the effects of λ may be found in Appendix 8.2.

Pseudocode for the overall hashing algorithm discussed in this section is contained in Appendix 8.4.

4.3 Pruning of Search Space

A weakness of the above hashing technique is the memory requirement. Every node in the graph will be stored, along with its hashword, w number of times. With each hashword of length l, the memory consumed will be $O(n \cdot w \cdot l)$. When working with very large graphs, it is desirable to decrease memory requirements to *below* $O(n)$. Since TopGC need search only for the top clusters in a graph, this allows for greater pruning of the search space. We can therefore further modify the LSH process, splitting it into two parts. First, a smaller low memory version of the modified LSH function is run on every node to pick out the top p nodes in the graph (nodes most likely to be part of the best clusters). Next, the full version of the algorithm described in the preceding sections is run on only these top p nodes. In this way the memory consumption is now restrained to $O(p \cdot w \cdot l)$.

This low memory version is based on the idea that a "promising" node is one most likely to increase the score of a high scoring cluster. To discover this likelihood, we first notice that, given a vertex v_a, the nodes of any cluster, $C = \{v_1, v_2, \cdots, v_c\}$ containing v_a can be divided into three subsets. Node v_a itself, the cluster nodes neighboring v_a in the set $S = C \cap N_a$, and the remaining set of nodes, R such that $\{v_a\} \cup S \cup R = C$. An illustration of this division is shown in Fig 2. For a high scoring cluster, $|S|$ and the

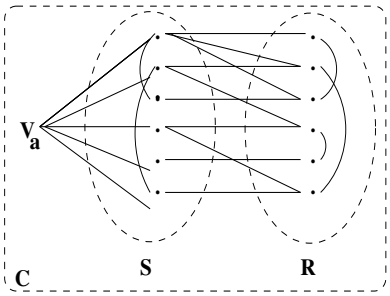


Figure 2: Illustration of subsets v_a, S, and R in cluster C

edge weights between v_a and S must be large. If they are too low, cluster C would have higher score without v_a and node v_a may be pruned. This divides the function of node v_a's likelihood score into four parts.

First, the set of edge weights connecting v_a to S (represented by VS, where $VS = \{w_{a,s}|s \in S\}$). Second, the set of edge weights connecting S to v_a ($SV = \{w_{s,a}|s \in S\}$). Third, the set of edge weights between members of S, represented here by SS. And fourth, the set of remaining edge weights within C (connecting the nodes R, as well as the nodes in S with R, represented by RR). We therefore have:

$$Score\ \alpha\ (VS + SV + SS + RR)$$

The larger each of these sets are, the better the overall cluster score. Maximizing all parts of this score would lead to obtaining clusters with maximum score, and therefore requires clustering the entire graph. To minimize time complexity, we choose to maximize just the first three terms as an approximation, obtaining a subscore related to the maximal scoring of the overall cluster. Experimental results confirm that this subscore is effective at choosing nodes likely to participate in strong clusterings, as discussed later in Section 5.1.

In addition, the list of possible nodes in S is limited to those connected to v_a with highest edge weights. Given that the maximum size of a cluster was a parameter S_{max}, we need only to consider some multiple of the S_{max} top nodes in N_a to get a notion of the likelihood of v_a being in a top cluster. (If the nodes in S are not contained within these top neighbors, then either a significant number of high weighted links leave the cluster and it is not well separated, or the links are all of low weight and v_a is not likely to participate in a high scoring cluster). This will give us a set, S_a, of nodes.

To maximize SV, we wish to find those nodes from S_a which have high edge weights to node v_a. To maximize SS, we wish to find those nodes within S_a which cluster well (having many links of high edge weight between them). Putting these two goals together we find that, in other words, we wish to find the best possible clusters from within the subgraph induced by the nodes in $\{v_a\} \cup S_a$. By applying the modified LSH algorithm described Subsection 4.2 to just this small subgraph, the set of best clusters within may be found in $O(S_{max})$ time, with $O(S_{max} \cdot w \cdot l)$ memory consumption. The cluster score of the best found cluster in this step may be used as an approximation to the score of v_a. Given that w and l are small constants, applying this function to every node in the graph will find the top p nodes in $O(n \cdot S_{max})$ time. In addition, given that our goal was to find dense, small subgraphs, S_{max} will be a small fixed constant relative to n, as well. This pruning step now allows the memory consumption to be limited to $O(p)$, as the individual hashtables for each node during pruning may be deleted after its score is found. Pseudocode for this process is shown in Appendix 8.4

To allow for a further speedup in the algorithm's running time, it can be seen that the pruning process is parallelizable. The score calculations of each node are separate. Thus, TopGC allows the user to split the pruning into any number of separate, parallel threads, giving it an approximate time complexity of $O(\frac{n}{t})$ (where t is the number of threads).

Table 1: List of parameter settings

Parameter	Setting
p	$min(0.3n, 50\text{k})$
S_{max}	20
S_{min}	5
λ	0.2
$trials$	S_{max}
m	$min(\alpha, 2S_{max})$
w	$2m$
l	2α

5. EXPERIMENTAL RESULTS

In this section, we report our experimental results obtained from implementing and running the TopGC program on a wide variety of different networks[1]. All experiments were run on an AMD Phenom 9750 Quad-Core Processor with 8GB of memory. We compare our performance to the state-of-the-art graph clustering program MLR-MCL [25], the scalable graph partitioning program Graclus [10], as well as to MCL [29], the graph clustering algorithm found to have the best clustering performance in several recent survey papers [3, 7]. To obtain the top scoring clusters from MLR-MCL, Graclus, and MCL, the programs were first run on the entire graph. Various parameter values were experimented with for all programs to obtain similarly sized clusters. Returned clusters of size s, $S_{min} \leq s \leq S_{max}$, were scored according to Eqn. 1 and the top scoring clusters output. Timing information reported here does not include the time taken to sort and output the clusters. TopGC was implemented in Java and all experiments were repeated five times, with the average results being reported here.

Table 1 lists the default settings used for the various TopGC parameters, unless otherwise specified. Here, α is shown to represent the average number of edges per node in the graph. The parameter values for the LSH hashing m, w, l, and *trials* were found to affect running time and cluster quality only slightly as long as they were within a reasonable threshold. Further details of their effect are discussed in Appendix 8.2. The values chosen in Table 1 were found experimentally to work well for a wide variety of graphs, and are the default values used in this paper.

5.1 Analysis of Pruning

To confirm the effectiveness of our pruning strategy from Section 4.3, the effect of varying the pruning parameter p is shown in Fig 3. The dataset used is a real world network from Amazon [19], containing ∼400,000 nodes and 3,000,000 edges. As can be seen in Fig 3, the scores for most values of p cluster tightly, meaning that using even low values of p (keeping less nodes and achieving better pruning) still gives clusters with high scores. In fact, from Fig 3 it can be seen that keeping just 5,000 nodes from pruning (about 1.2% of the graph) gives top scoring clusters of approxi-

[1]We have made our implementation of TopGC available for download at http://cs.ucsb.edu/∼kpm/TopGC .

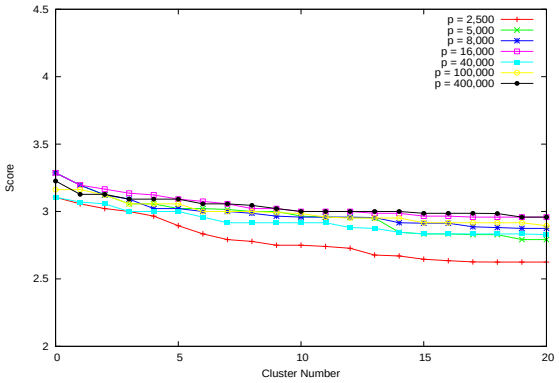


Figure 3: Varying p on a graph of ∼400,000 nodes

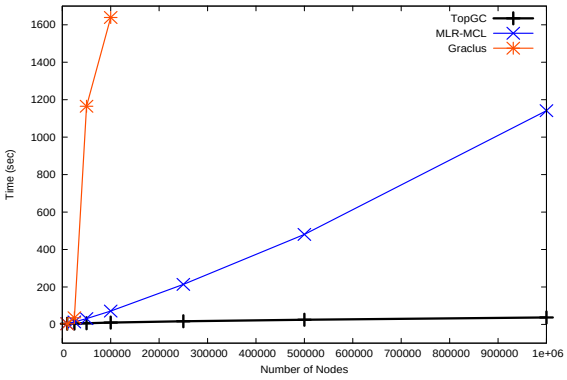


Figure 4: Running times on generated graphs

mately the same score as using the majority of the graph, confirming that our strategy and scoring method does indeed pick out those nodes that participate in strong clusters. The memory savings from using lower values of p is significant as well, with $p = 5,000$ leading to a ∼750MB maximum resident set size of memory, compared to ∼1.82GB for $p = 400,000$, a saving of almost 60% (and one that is magnified still further in larger datasets, as will be seen in Section 5.4).

5.2 Synthetically Generated Graphs

To illustrate scalability, a series of seven graphs of increasing size were generated using the Python Web Graph Generator [2]. This graph generator produces power law random, unweighted, undirected graphs using a variant of the R-MAT [8] algorithm. The average number of edges per node was kept at a constant of five, and the number of nodes increased from between 10,000 to 1,000,000. Parameters for the number of partitions in Graclus and coarsening in MLR-MCL were set so as to give similarly sized clusterings as TopGC (typically the number of nodes divided by 5). The running times for MCL are omitted as well, being several orders of magnitude slower (2618 seconds for 25,000 nodes). Additionally, results for Graclus are shown only to 100,000 nodes. Above this, the program began thrashing due to memory constraints.

Fig 4 compares the resulting running times of TopGC, MLR-MCL, and Graclus. It can be seen that TopGC clearly dominates the other algorithms, its running time varying from 3.7 seconds on a 10k node graph to ∼40 seconds on a 1,000k node graph. Graclus

[2]Available at http://pywebgraph.sourceforge.net/

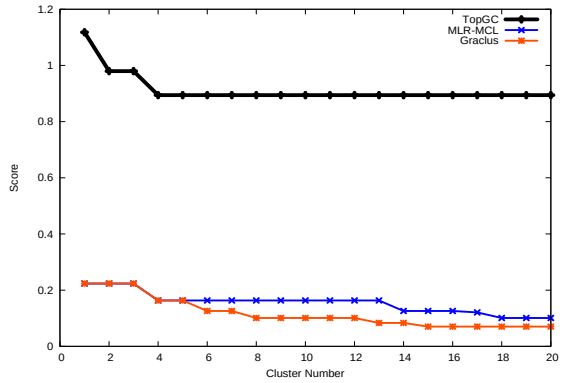


Figure 5: Top 20 cluster scores on a 100K node graph

performs well on the smaller graphs, but the number of partitions necessary to allow it to return similarly sized clusters on larger scale networks contributes to its poor performance in running time here. Fig 5 compares the scores for the top 20 clusters returned from the algorithms on the 100,000 node graph. A dramatic difference in the strength of returned clusters can be seen, with the clusters returned by both MLR-MCL and Graclus being only loosely connected. This emphasizes their role as graph partitioning algorithms focused on splitting a graph into a few number of larger sized partitions, making them unsuitable for those applications which need smaller, dense clusters. Figures for other synthetic graphs are omitted here, as they produced similar results.

5.3 Yeast Biological Network

We validated the quality of our clustering algorithm on a real world network dataset with available ground truth. This dataset, called YeastNet [17], is a weighted, undirected graph whose ∼6,000 nodes represent yeast genes, and ∼100,000 edges are weighted according to the likelihood that the two genes participate in the same biological function. It has been shown that by finding tight, clique-like clusters within such networks, new functional predictions for previously uncategorized genes may be found [22]. In addition, it is known that some genes participate in more than one biological function, emphasizing the need for a clustering solution that allows for slightly overlapping clusters.

To obtain a gold standard, Gene Ontology (GO) annotations [4] for yeast biological functions were obtained, and the GO Biological process terms used to identify sets of proteins annotated with the same terms. A list of the 295 significant GO biological process terms was used, as identified by Myers et al. [24] More details on this benchmark are contained in Appendix 8.3. To obtain a cluster "purity" score, the nodes in each cluster were associated with their GO annotations. For a single cluster, the purity is defined as the ratio of the maximum number of nodes annotated with the same GO process to the number of nodes in the cluster. The purity of a cluster therefore ranges from 1 (all nodes having the same annotation) to 0 (no matching annotations).

Results for the top scoring clusters returned by all algorithms are shown scored by the GO standard in Table 2, and the calculated score of the top 20 clusters in Fig 6. An inflation of 4.0 for MCL and partition size of 1000 for Graclus was found experimentally to give the highest purity results, and are used here. MLR-MCL had difficulty separating the clusters from this yeast dataset, and most parameter settings gave results with too few separate clusters to be compared. Only the extreme parameter values of 8.9 for g and

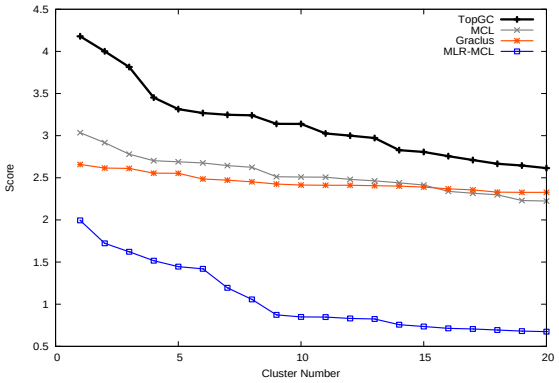


Figure 6: Comparing the top 20 cluster scores in YeastNet

5000 for c gave enough clusters for comparison, though this still resulted with one super-cluster containing the majority of nodes in the network. For all algorithms only those clusters of size $\geq$ five were kept, and an average size of 5-10 nodes per cluster for all techniques was found. Using the default parameters shown in Table 1, TopGC returned 67 clusters (with an average size of 9.1 and average score of 2.47).

Fig 6 shows that the calculated scores for the top clusters returned by TopGC again outrank the scores returned from the other algorithms. The difference is especially pronounced among the top few clusters. From Table 2, it can be seen that the clusters returned by TopGC have real world relevance as well, outperforming even the popular biological network clustering algorithm, MCL. In fact, 74% of the top fifty clusters returned by TopGC contained nodes with over 0.9 purity when compared to the GO gold standard, an increase of 11% over MCL. TopGC's method of weighted similarity search can be seen to effectively find those clusters that are strongly connected, leading to both high calculated and real world scores. The lower purity results for MLR-MCL and Graclus emphasize their bias towards finding large graph partitions, making them inappropriate for finding smaller, tight-knit groupings of nodes. Both MLR-MCL and Graclus performed faster on this small yeast dataset than TopGC but, as shall be shown in the next section, as graph sizes increase, both algorithms scale poorly in comparison.

5.4 Large Scale Real World Networks

To confirm that TopGC produces strong and fast results on large real world networks, five directed, unweighted, real world graphs were obtained from the SNAP graph library [18] and TopGC used to find the highest scoring clusters. These graphs include a social network graph from LiveJournal [21] consisting of $\sim$5 million nodes and $\sim$70 million edges, a Wikipedia talk network [20] between interacting individuals on Wikipedia's talk pages, a webgraph representing links between berkely.edu and stanford.edu domains [21], a product co-purchasing network from Amazon [19], and a social graph from Slashdot's zoo feature [20]. A comparison of the results from TopGC and MLR-MCL, with scores from the top 50 returned clusters, are shown in Table 3. A coarsening value of 5000 was used MLR-MCL to allow for the return of similarly sized clusters. The results from both MCL and Graclus are not included in this table as they were unable to complete on the majority of graphs due to time, and memory constraints.

From Table 3, TopGC's results again confirm its scalability and strong clustering ability. TopGC consistently performs faster and returns similar or higher scoring clusters than MLR-MCL on all

Table 2: Comparison of YeastNet results to GO gold standard

Cluster Purity	TopGC 8.3 sec.	MCL 139 sec.	Graclus 4.90 sec.	MLR-MCL 1.3 sec.
0.9	74%	63%	36%	13%
0.8	88%	83%	48%	21%
0.7	98%	92%	58%	25%
0.6	100%	96%	66%	33%
0.5	100%	98%	82%	50%
0.25	100%	100%	100%	100%

datasets, the difference becoming especially clear as graph sizes increase. On the webgraph dataset, TopGC's running time is over 3.5 times faster than MLR-MCL, while also returning clusters of much higher score. For the two largest datasets, the results for MLR-MCL could not be obtained, due to it running out of memory. TopGC, conversely, with the use of its pruning process, needed roughly the same amount of resident set size memory to cluster each of these graphs (needing, for example, only 1.1GB of memory to cluster the webgraph dataset, and 1.3GB of memory to cluster the LiveJournal dataset). This highlights again the need for both time and memory scalability, and further emphasizes TopGC's ability to efficiently find strong clusters within the massive graphs available today.

6. CONCLUSIONS

Graph clustering is an important tool for the mining and visualization of graphs, especially in those massive graphs that are most difficult to handle. Though finding all clusters in a massive graph may take an inordinate amount of time and space, it is possible to prune the search space by examining just for those clusters with highest scores. Since many real world applications on large graphs need only the subset of most strongly connected clusters, this is a solution that can produce both interesting and relevant results.

In this paper we have introduced and made available a new tool, TopGC, which allows for the probabilistic search of a graph for its top scoring clusters in linear time. We also show that the probability of TopGC finding a cluster is related to a lower bound on the cluster score. TopGC works with directed or undirected, weighted graphs, and finds clusters that overlap to a given percentage. It has a limited memory consumption, and testing TopGC against other state-of-the-art graph clustering techniques shows TopGC consistently performing faster, scaling well with graph size, and performing up to 70% faster on real world datasets. When scaled to massive real world graphs of up to 5 million nodes and 70 million edges, all three other previous graph clustering techniques were unable to run to completion, due to their time and memory constraints. However, TopGC is able to return high scoring clusterings on these massive datasets with just $\sim$1GB-2GB of resident set memory and completes in a running time of a few minutes. Finally, not only is TopGC shown to be scalable with respect to running time and memory, but the top clusters returned by the TopGC algorithm consistently have higher scores than other current clustering techniques, when compared to both calculated scores and real world benchmarks. TopGC is shown to return up to 11% more clusters with higher real world biological relevance than MCL, a popular graph clustering algorithm in bioinformatics.

These results lead us to conclude that TopGC's technique for pruning the search space on large graphs to find top scoring clusters is an effective solution to this interesting problem, and can allow the TopGC algorithm to be both useful and relevant as a scalable clustering solution on massive real world graphs.

Table 3: Summary of experiments on large real world networks

			TopGC			MLR-MCL		
Graph	Nodes	Edges	Time (sec)	Avg. Score	Avg. Size	Time (sec)	Avg. Score	Avg. Size
LiveJournal	4,847,571	68,993,773	199	3.1	13.9	–	–	–
Wikipedia	2,394,385	5,021,410	37.3	1.4	6.6	–	–	–
Webgraph	685,230	7,600,595	28.6	2.6	12	102	1.5	9
Amazon	410,236	3,356,824	22.2	2.7	9.1	71.8	2.8	13
Slashdot	82,168	948,464	11.0	2.0	7.7	19.9	0.78	6

7. REFERENCES

[1] A. Abourjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *IPDPS*, pages 16–575, 2006.

[2] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, Washington, DC, USA, 2006. IEEE Computer Society.

[3] B. Andreopoulos, A. An, X. Wang, and M. Schroeder. A roadmap of clustering algorithms: finding a match for a biomedical application. *Brief Bioinform*, 10(4):297–314, May 2009.

[4] M. Ashburner and et. al. Gene ontology: tool for the unification of biology. The Gene Ontology Consortium. *Nat Genet*, 25(1):25–29, May 2000.

[5] G. D. Bader and C. W. V. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4(2), 2003.

[6] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, 1997.

[7] S. Brohee and J. van Helden. Evaluation of clustering algorithms for protein-protein interaction networks. *BMC Bioinformatics*, 7:488, November 2006.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *In SDM*, 2004.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. pages 137–150.

[10] I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors: A multilevel approach. *TPAMI*, 29(11):1944–1957, 2007.

[11] S. Fortunato. Community detection in graphs. *Physics Reports*, 486:75–174, Feb. 2010.

[12] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[13] T. H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web (extended abstract). In *WebDB*, 2000.

[14] P. Indyk. A small approximately min-wise independent family of hash functions. In *SODA '99*, pages 454–456, 1999.

[15] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515, May 2004.

[16] E. F. Krause. Maximizing the product of summands; minimizing the sum of factors. *Mathematics Magazine*, 69(4):270–278, 1996.

[17] I. Lee, Z. Li, and E. M. Marcotte. An improved, bias-reduced probabilistic functional gene network of baker's yeast, saccharomyces cerevisiae. *PloS one*, 2(10):e988+, October 2007.

[18] J. Leskovec. Stanford network analysis package (snap). `http://snap.stanford.edu/`.

[19] J. Leskovec, L. A. Adamic, and B. A. Huberman. The dynamics of viral marketing. In *EC*, pages 228–237, New York, NY, USA, 2006. ACM.

[20] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Signed networks in social media. In *CHI*, 2010.

[21] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Technical Report arXiv:0810.1355, 2008.

[22] K. Macropol, T. Can, and A. Singh. RRW: repeated random walks on genome-scale protein networks for local cluster discovery. *BMC Bioinformatics*, 10(1):283, 2009.

[23] N. Mishra, R. Schreiber, I. Stanton, and R. E. Tarjan. Clustering social networks. In *WAW*, pages 56–67, 2007.

[24] C. L. Myers, D. R. Barrett, M. A. Hibbs, C. Huttenhower, and O. G. Troyanskaya. Finding function: evaluation methods for functional genomic data. *BMC genomics*, 7:187, 2006 2006.

[25] V. Satuluri and S. Parthasarathy. Scalable graph clustering using stochastic flows: applications to community discovery. In *KDD*, pages 737–746, New York, NY, USA, 2009. ACM.

[26] J. Shi and J. Malik. Normalized cuts and image segmentation. *TPAMI*, 22:888–905, 1997.

[27] D. A. Spielman and S.-H. Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. *CoRR*, abs/0809.3232, 2008.

[28] A. P. Streich, M. Frank, D. Basin, and J. M. Buhmann. Multi-assignment clustering for boolean data. In *ICML*, pages 969–976, New York, NY, USA, 2009. ACM.

[29] S. van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, Utrecht, May 2000.

[30] W.-S. Yang, J.-B. Dia, H.-C. Cheng, and H.-T. Lin. Mining social networks for targeted advertising. In *HICSS*, page 137.1, 2006.

[31] Y. Yu, M. Crucianu, V. Oria, and L. Chen. Local summarization and multi-level LSH for retrieving multi-variant audio tracks. In *ACM Multimedia*, pages 341–350, 2009.

[32] Z. Zou, J. Li, H. Gao, and S. Zhang. Finding top-k maximal cliques in an uncertain graph. In *ICDE*, pages 625–636, 2010.

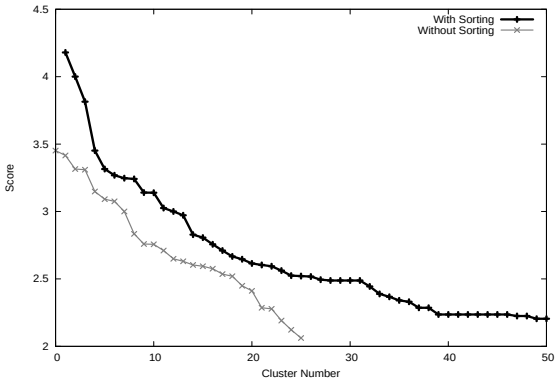


Figure 7: Comparing the top 50 cluster scores with and without signature modification

Table 5: Effect of varying λ

λ	Avg. Score
0	2.4
0.1	2.5
0.2	2.6
0.3	2.6
0.4	2.6

Table 6: Effect of varying S_{max}

S_{max}	Time	Avg. Score	Avg. Size
5	25.5	1.3	5.0
10	27.9	2.5	9.8
20	28.6	2.6	12.0
50	31.5	2.6	12.0
100	34.1	2.7	12.3

8. APPENDICES

8.1 Modification of Signature

Fig 7 shows the effect of the modification of hashword signatures introduced in Section 4.2. Results shown are from the YeastNet network. The bottom line displays the results obtained from using the original, unmodified signatures, while the top line represents the results obtained from our modification. As expected, there are far less clusters returned from the original signatures, with on average approximately 26 clusters being returned, as compared with over 65 clusters found by the full TopGC algorithm. In addition, the scores of the resulting clusters are lower, as well, due to the difficulty of finding clusters using the full signatures. Similar results were found on other datasets. This helps demonstrate the effectiveness of the signature modification introduced in the TopGC technique from Section 4.2.

8.2 Parameter Analysis

In this section we examine the various TopGC parameters, and how their values effect obtained results. In Table 4, the parameters m (number of minhashes), w (number of signatures), l (length of a signature), and $trials$ (number of weighted neighborhood instances to draw) are varied and the resulting statistics compared. All experiments in this table were performed on the Amazon dataset described in Section 5.4. It can be seen from Table 4 that the results stay reasonably consistent and high for all values tested here, emphasizing the consistency and robustness of TopGC as a clustering method. The parameter l can be seen to have the most consistent and significant effect. As the signature length increases, the average clustering score tends to increase. This agrees with our scoring method and bound found in Section 4.2. By increasing the length of the signature, we are able to obtain a better approximation for Pr[$minhash_C = v_j$], and therefore return higher scoring clusters.

The effect of varying the overlap threshold parameter, λ, on the YeastNet dataset can be seen in Table 5. With a λ of 0.2, on average there were three sets of clusters, with average size 9, that overlapped in one or two nodes. From Table 5, it can be seen that increasing λ gives a small but definite boost to the average score of returned clusters. In addition, this allows slightly overlapping but still significant clusters to be returned. No appreciable time or size differences between the returned clusters were observed.

Table 6 examines the effect of changing the maximum cluster size, S_{max}. As S_{max} increases, the average size of the top scoring clusters, as well as their score, increase as well. This continues until a maximum is reached, and at this point, no higher scoring larger clusters within the graph can be found. It is worthwhile to note that the increase of S_{max} increases running time only slightly, allowing for larger values of S_{max} to be used if necessary on a graph.

8.3 Biological Gold Standard

The yeast biological process annotations used for the Gold Standard were downloaded (May 23, 2008 version), and those annotations belonging to the 295 significant GO biological process terms [24] kept. Only those terms annotated to at least 5 proteins were kept, leaving 158 terms. We accounted for hierarchical information among the terms by allowing genes with an annotation lower in the tree to match with their parent annotations. These 158 terms and their associated genes were then used as the gold standard.

8.4 Pseudocode

Algorithms 1 and 2 show the pseudocode for the basic TopGC algorithm, as introduced in Sections 4.2 and 4.3

9. ACKNOWLEDGMENTS

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

Work was also partially supported by the National Science Foundation under grants IIS-0917149 and IIS-0612327.

Table 4: Effect of varying parameters

	m			w			l			$trials$		
	Time (sec)	Avg. Score	Avg. Size	Time (sec.)	Avg. Score	Avg. Size	Time (sec.)	Avg. Score	Avg. Size	Time (sec.)	Avg. Score	Avg. Size
2	21.2	2.7	9.8	18.5	2.9	11.2	22.1	2.8	9.6	20.5	2.7	9.5
3	16.6	2.7	9.4	18.8	2.9	11.2	21.5	2.7	9.4	23.5	2.7	9.8
4	20.3	2.7	9.4	17.4	3.0	10.7	21.7	2.7	9.7	20.1	2.7	9.2
5	21.2	2.8	9.8	19.3	2.7	9.5	22.5	2.9	11.1	22.5	2.7	9.1
6	23.9	2.8	9.8	19.5	2.8	10.5	20.1	3.0	11.2	21.6	2.7	9.6
8	23.2	2.8	10.3	18.9	2.7	9.3	19.6	3.0	11.1	22.1	2.7	9.4
10	26.8	2.7	9.3	22.6	2.7	9.7	23.7	3.0	11.5	21.1	2.8	9.8

Algorithm 1 Graph Hashing

Require: Set of nodes $\mathbf{p}$, m, l, w, k
Ensure: Set of Clusters C
  for $i = 1$ to m **do**
    Initialize minhashes $mh_i()$ with random values for a, b, and p
  end for
  Initialize s[i][j] {Signature Orderings}
  for $i = 1$ to w **do**
    for $j = 1$ to l **do**
      s[i][j] $\leftarrow$ (random value between 1 and m)
    end for
  end for
  Initialize Hashtable H {Create and store signature}
  for $n \in \mathbf{p}$ **do**
    for $i = 1$ to $trials$ **do**
      c $\leftarrow$ (random value between 0 and 1)
      $\mathbf{N} \leftarrow$ Neighborhood of n where $w_{n,v} \geq c$
      for $j = 1$ to m **do**
        minhash[j] $\leftarrow mh_j(\mathbf{N})$
      end for
      for $j = 1$ to w **do**
        Initialize T {Signature}
        for $k = 1$ to l **do**
          $T \leftarrow T$ + "minhash[$s[j][k]$]"
        end for
        $T \leftarrow$ sort items in T and remove duplicates
        Store (T, n) in H
      end for
    end for
  end for
  Initialize set C {Retrieve clusters from Hashtable}
  $threshold \leftarrow 1$
  $found \leftarrow 0$
  while $found \leq k$ **do**
    for Bin b in Hashtable H **do**
      $Score \leftarrow \frac{b(T) \cap b(nodes)}{|b(nodes)|}$
      if $Score \geq threshold$ **then**
        Add b to C
        $found \leftarrow found + 1$
      end if
    end for
    $threshold \leftarrow threshold - 0.1$
  end while
  return C

Algorithm 2 Graph Pruning

Require: Graph $\mathbf{G}$, m, l, w, k, p, S_{max}
Ensure: Set of nodes, S, of size p
  for $i = 1$ to m **do**
    Initialize minhashes $mh_i()$ with random values for a, b, and p
  end for
  Initialize s[i][j] {Signature Orderings}
  for $i = 1$ to w **do**
    for $j = 1$ to l **do**
      s[i][j] $\leftarrow$ (random value between 1 and m)
    end for
  end for
  Initialize Heap $heap$
  for $n \in \mathbf{G}$ **do**
    Initialize Hashtable H
    Add (n, N_n) to set $nodes$
    Set $topN \leftarrow$ top S_{max} neighbors in $\mathbf{N}$
    for $i \in TopN$ **do**
      Add(i, N_i) to set $nodes$
    end for
    for $a \in nodes$ **do**
      for $i = 1$ to $trials$ **do**
        c $\leftarrow$ (random value between 0 and 1)
        $\mathbf{N} \leftarrow$ Neighborhood of a where $w_{a,v} \geq c$
        for $j = 1$ to m **do**
          minhash[j] $\leftarrow mh_j(\mathbf{N})$
        end for
        for $j = 1$ to w **do**
          Initialize T {Signature}
          for $b = 1$ to l **do**
            $T \leftarrow T$ + "minhash[$s[j][b]$]"
          end for
          $T \leftarrow$ sort items in T and remove duplicates
          Store (T, n) in H
        end for
      end for
    end for
    Initialize $MaxScore$ {Go through hashtable to find top score}
    for Bin b in Hashtable H **do**
      $Score \leftarrow \frac{b(T) \cap b(nodes)}{|b(nodes)|}$
      if $Score \geq MaxScore$ **then**
        $MaxScore \leftarrow Score$
      end if
    end for
    $heap \leftarrow (MaxScore, n)$
  end for
  $S \leftarrow$ Top p nodes from $heap$
  return S